Notas e Exercícios de Complexidade

Armando Matos
Nelma Moreira

Departamento de Ciência de Computadores
Faculdade de Ciências, Universidade do Porto
email: {acm,nam}@ncc.up.pt

Índice

Lista de Figuras

Capítulo 1

Computabilidade e Teoria da Recursão

1.1 Máquinas de Turing

Bibliografia [HU79, Cap. 7], [Pap94, Cap. 2],[LP81, Cap. p4][Dew89, Cap. 28]

Começamos por definir um tipo básico de máquina de Turing. Como veremos, existem muitos outros modelos, todos equivalentes a este.

Uma máquina de Turing é constituída por um controlo finito (conjunto finito de estados), uma fita dividida em células, e uma cabeça de leitura/escrita (RW) que actua sobre uma célula da fita de cada vez. Supomos que a fita tem uma célula mais à esquerda mas é infinita para a direita. No início supomos que as n células mais à esquerda contém a sequência de entrada ("dados") e as restantes o caracter de fita "branco".

a_1	a_2	$\ldots$	a_i	$\ldots$	a_n	b	$\ldots$

$$\uparrow$$

$$\boxed{\text{controlo finito}}$$

Num movimento, dependendo do símbolo lido na fita pela cabeça e do estado do controlo finito, a máquina

1. muda de estado
2. escreve um símbolo na célula que está debaixo da cabeça
3. move a cabeça para a esquerda ou para a direita

Formalmente, uma máquina de Turing (TM) é um tuplo

$$M = (S, \Sigma, \Gamma, \delta, s_0, b, F)$$

onde

S é um conjunto finito de estados

Γ é o conjunto finito de *símbolos da fita*

Σ é um subconjunto de Γ que não incluí b, é o conjunto dos *símbolos de entrada*

δ é a *função movimento seguinte*, função parcial de $S \times \Gamma$ em $S \times \Gamma \times \{l, r\}$

s_0 é o estado inicial

b é um símbolo de Γ, designado por *branco*

$F \subseteq S$ é o conjunto de estados finais

Uma *descrição instantânea* (ID) ou *configuração* é representada por xsy, onde s é o estado corrente, $x \in \Gamma^\star$ é a sequência de símbolos à esquerda da cabeça, sendo o último símbolo de x o que está imediatamente à esquerda da cabeça, e $y \in \Gamma^\star$ é a sequência à direita da cabeça (o primeiro símbolo de y é o que está debaixo da cabeça). Supomos que x e y não contêm "brancos" desnecessários, mas y pode ser b.

A relação $\underset{M}{\longrightarrow}$, um *movimento* (passo), entre duas configurações é definida da seguinte forma. Seja $X_1 \cdots X_{i-1} s X_i \cdots X_n$ uma configuração. Se $\delta(s, X_i) = (s', Y, D)$ e $i = 1$ então $D = r$ obrigatoriamente. Se $i > 1$ e $D = l$ então,

$$X_1 \cdots X_{i-1} s X_i \cdots X_n \underset{M}{\longrightarrow} X_1 \cdots X_{i-2} s' X_{i-1} Y X_{i+1} \cdots X_n$$

Analogamente se $D = r$ tem-se que

$$X_1 \cdots X_{i-1} s X_i \cdots X_n \underset{M}{\longrightarrow} X_1 \cdots X_{i-1} Y s' X_{i+1} \cdots X_n$$

Diz-se que $xsy \underset{M}{\longrightarrow}^k x's'y'$ se $x's'y'$ resulta de xsy em k movimentos e $xsy \underset{M}{\longrightarrow}^\star x's'y'$ se existe $k > 0$ tal que $xsy \underset{M}{\longrightarrow}^k x's'y'$.

A *linguagem aceite por* M, é

$$L(M) = \{x \mid \ x \in \Sigma^\star \text{ e } s_0 x \underset{M}{\longrightarrow}^\star x_1 s x_2 \text{ para algum } s \in F, \text{ e } x_1, x_2 \in \Gamma^\star\}$$

Exemplo 1 *Sendo* $L = \{0^n 1^n \mid \ n \geq 1\}$ *definimos a seguinte TM,*

$$M = (\{s_0, s_1, s_2, s_3, s_4\}, \{0, 1\}, \{0, 1, X, Y, b\}, \delta, s_0, b, \{s_4\})$$

onde,

$$\begin{array}{ll}
\delta(s_0, 0) = (s_1, X, r) & \delta(s_0, Y) = (s_3, Y, r) \\
\delta(s_1, 0) = (s_1, 0, r) & \delta(s_1, 1) = (s_2, Y, l) \\
\delta(s_1, Y) = (s_1, Y, r) & \delta(s_2, 0) = (s_2, 0, l) \\
\delta(s_2, X) = (s_0, X, r) & \delta(s_2, Y) = (s_2, Y, l) \\
\delta(s_3, Y) = (s_3, Y, r) & \delta(s_3, b) = (s_4, b, r)
\end{array}$$

Exercício 1.1.1 Descreva máquinas de Turing que aceitem cada uma das seguintes linguagens:

a) $\{0^n 1 0^n \mid \ n \geq 1\}$

b) o conjunto de sequências com igual número de 1s e de 0s

c) $L = \{ww^r \mid w \in \{0, 1\}^\star\}$ onde w^r corresponde ao inverso de w

Se $x \in L(M)$ então M pára quando atinge um estado final. Caso contrário, M pode não parar. Uma linguagem diz-se *recursivamente enumerável* (r.e) se é aceite por uma máquina de Turing. Neste caso, consideramos TMs como *aceitadoras* de linguagens.

Note-se que sendo M uma TM, o problema $x \in L(M)$? não é computável. Se for verdade, sendo x a sequência de entrada (dados) de M, esta pára num número finito de movimentos (passos). Mas se $x \notin L(M)$, M pode não parar e portanto não se obtém resposta.

Designam-se por *recursivas* as linguagens para as quais existe pelo menos uma TM que pára para todos os dados.

Podemos, também, considerar TMs *reconhecedoras* de linguagens, isto é, máquinas

$$M = (S, \Sigma, \Gamma, b, \delta, s_0, b, s_a, s_r)$$

onde há só dois estados finais, um estado de aceitação, s_a e um estado de rejeição, s_r. Neste caso, as máquinas param para quaisquer dados. Como veremos, isto equivale a TM calcular uma função total de $\Sigma^\star$ em $\{0,1\}$.

Exercício 1.1.2 Descreva uma máquina de Turing que determine se um $x \in \{0,1\}^\star$ é um palíndromo.

1.1.1 Máquinas de Turing calculadoras de funções parciais

Uma máquina de Turing pode ser vista como uma calculadora de funções parciais dos inteiros nos inteiros:

$$f : \mathbb{N}^k \longrightarrow_p \mathbb{N}$$

Suponhamos que os inteiros estão codificados em *unário*. Então, um inteiro $i \geq 0$ é representado por 1^i. Se uma função tiver k argumentos $i_1, \ldots, i_k$, a sequência de entrada pode ser $1^{i_1}0 \ldots 01^{i_k}$. Se a TM pára com a sequência 1^m na fita, então $f(i_1, \ldots, i_k) = m$. Se a TM pára para quaisquer argumentos, então f é uma função recursiva total. Note-se que as funções recursivas totais correspondem a linguagens recursivas.

Exemplo 2 *Seja*

$$m \dot{-} n = \begin{cases} m - n & \textit{se } m \geq n \\ 0 & \textit{caso contrário} \end{cases}$$

A TM $M = (\{s_0, s_1, \ldots, s_6\}, \{0,1\}, \{0,1,b\}, \delta, s_0, b, \emptyset)$*, onde*

$$\begin{array}{lll}
\delta(s_0,1) = (s_1,b,r) & \delta(s_1,1) = (s_1,1,r) & \delta(s_1,0) = (s_2,0,r) \\
\delta(s_2,0) = (s_2,0,r) & \delta(s_2,1) = (s_3,0,l) & \delta(s_3,1) = (s_3,1,l) \\
\delta(s_3,0) = (s_3,0,l) & \delta(s_3,b) = (s_0,b,r) & \delta(s_2,b) = (s_4,b,l) \\
\delta(s_4,0) = (s_4,b,l) & \delta(s_4,1) = (s_4,1,l) & \delta(s_4,b) = (s_6,0,r) \\
\delta(s_0,0) = (s_5,b,r) & \delta(s_5,0) = (s_5,b,r) & \delta(s_5,1) = (s_5,b,r) \\
\delta(s_5,b) = (s_6,b,r) & &
\end{array}$$

calcula $1^m01^n \underset{M}{\longrightarrow^\star} 1^{m \dot{-} n}$. A máquina vai sucessivamente apagando um 1 da representação de m e substituindo um 1 por 0 na representação de n, até que uma das seguintes condições ocorre:

- quando procura um 1 na representação de n, não o encontra. Neste caso, $m > n$. Então substitui $n+1$ $0's$ por um 1 e por n símbolos b, deixando $m \dot{-} n$ 1s na fita.
- quando começa um ciclo não encontra nenhum 1. Neste caso, $m < n$. Então apaga toda a fita.

Exercício 1.1.3 *Escreva máquinas de Turing que permitam calcular as seguintes funções, supondo os inteiros codificados em "unário":*

(i) $f(n) = n + 1$

(ii) $f(n) = n^2$

(iii) $f(n, m) = n + m$

Dum modo geral uma TM M calcula uma função parcial f de $\Sigma^\star$ em $\Sigma^\star$ se dada como sequência de entrada $x \in \Sigma^\star$ a configuração final de M é $f(x)sb$, para um estado final s.

1.1.2 Variações sobre as Máquinas de Turing

Várias modificações da TM básica são possíveis sem que o modelo de computação obtido seja nem mais nem menos poderoso.

Entre elas considerem-se as seguintes restrições/extensões:

1. Alfabeto.
   (a) Codificação de um alfabeto noutro com pelo menos dois símbolos. Dê exemplos e conclua que o alfabeto não reduz a potência.
2. Aridade das funções
   (a) Codificação de pares de inteiros em inteiros. Considere a função bijectiva p de $\mathbf{N}_0 \times \mathbf{N}_0$ em $\mathbf{N}$
   $$p(x, y) = \frac{(x+y)(x+y+1)}{2} + x$$
   (b) Codificações bijectivas de $\mathbf{N}^n$ em $\mathbf{N}$ usando a função p definida atrás.
   (c) Conclusão: As funções de aridade maior que 1 podem ser codificadas em funções de aridade 1.
3. Movimentos da cabeça
   (a) Máquinas com movimentos $\{l, r\}$ (esquerda/direita)
   (b) Máquinas com movimentos $\{l, n, r\}$ (esquerda/não mexe/direita)
4. Tamanho da fita
   (a) Máquinas com uma fita semi-infinita
   (b) Máquinas com uma fita duplamente infinita
5. Número e tipo de fitas
   (a) Máquinas com 1 fita RW
   (b) Máquinas com k fitas RW
   (c) Máquinas com fita de escrita de saída (W) e uma fita RW de trabalho.

Vamos apenas analisar a equivalência dos seguintes modelos: 3a e 3b;4a e4b; 5a e 5b.

Máquinas com movimentos $\{l, n, r\}$ Exceptuando a terceira componente da definição de δ estas máquinas são definidas como as TM básicas.

Se M_1 é uma TM com movimentos $\{l, r\}$ também é uma TM com movimentos $\{l, n, r\}$. Então, suponhamos que $M_2 = (S_2, \Sigma_2, \Gamma_2, \delta_2, s_2, b, F_2)$ é uma máquina com movimentos $\{l, n, r\}$. Vamos construir uma TM apenas com movimentos $\{l, r\}$, M_1, tal que se L é aceite por M_2 então, L é aceite por M_1. Quando a cabeça de M_2 fica parada, M_1 deve entrar num estado especial e movimentar-se uma célula para direita e outra para a esquerda, voltando à mesma célula de tal modo que simule o facto de M_2 não se mexer. Para tal, os estados de M_1 vão ser da forma $[s, X]$ ou $[s, Y]$ para cada estado $s \in S_2$, sendo os estados $[s, X]$ usados sempre que a cabeça de M_2 se movimente e os estados $[s, Y]$ usados no caso da cabeça de M_2 não se mova. Formalmente, tem-se $M_1 = (S_1, \Sigma_1, \Gamma_1, \delta_1, [s_2, X], b, F_1)$ onde

$S_1 = \{[s, X], [s, Y] \mid s \in S_2\}$, $F_1 = F_2 \times \{X, Y\}$

$\Sigma_1 = \Sigma_2$, $\Gamma_1 = \Gamma_2$

δ_1 define-se por:

1. $\forall \alpha \in \Gamma_1$,

$$\delta_1([s, X], \alpha) = ([s', X], \alpha', r) \qquad \text{se } \delta_2(s, \alpha) = (s', \alpha', r)$$

2. $\forall \alpha \in \Gamma_1$,

$$\delta_1([s, X], \alpha) = ([s', X], \alpha', l) \qquad \text{se } \delta_2(s, \alpha) = (s', \alpha', l)$$

3. $\forall \alpha \in \Gamma_1$,

$$\delta_1([s, X], \alpha) = ([s', Y], \alpha', r) \qquad \text{se } \delta_2(s, \alpha) = (s', \alpha', n)$$

4. $\forall \alpha \in \Gamma_1, \forall s \in S_2$,

$$\delta_1([s, Y], \alpha) = ([s, X], \alpha, l)$$

Máquinas com fita duplamente infinita Uma TM com uma fita duplamente infinita é definida como a TM básica com a excepção de que não existe nenhuma célula mais à esquerda e portanto δ não está limitada nos seus movimentos para a esquerda. A noção de configuração e da relação $\underset{M}{\longrightarrow}$ são definidas análogamente, mas neste caso também se supõe a existência de uma infinidade de "brancos" à esquerda, que podem ser omitidos. No início a sequência de entrada pode começar em qualquer célula.

Exercício 1.1.4 Formalize as noções associadas a uma TM com fita duplamente infinita.

Teorema 1.1.1 *L é uma linguagem aceite por uma TM com uma fita duplamente infinita se e só se é aceite por uma TM com uma fita semi-infinita.*

Dem. ($\Leftarrow$) Uma TM M_2 com uma fita duplamente infinita pode simular facilmente uma TM com uma fita semi-infinita, M_1. Basta que M_2 marque a célula à esquerda da cabeça no início e depois simule a TM M_1.

($\Rightarrow$) Seja $M_2 = (S_2, \Sigma_2, \Gamma_2, \delta_2, s_2, b, F_2)$ uma TM com fita duplamente infinita. Vamos construir M_1 com uma fita semi-infinita para a direita. Podemos imaginar que a fita de M_1

é dividida ao meio em duas pistas, de modo a que cada célula pode conter dois símbolos. Uma pista (S) corresponde às células da fita de M_2 à direita da célula onde inicialmente se encontra a cabeça e a outra (I) às células à esquerda dessa célula. Por exemplo:

Fita de M_2:

...	a_{-2}	a_{-1}	a_0	a_1	a_2	...

Fita de M_1:

a_0	a_1	a_2	...
\$	a_{-1}	a_{-2},	...

onde a_0 é a célula que inicialmente está debaixo da cabeça de M_2. A primeira célula da fita de M_1 contém o símbolo \$, na "pista" inferior, para indicar que é a célula mais à esquerda. O controlo finito de M_1 indicará se a cabeça de M_2 está debaixo duma célula correspondente à parte superior ou inferior da fita de M_1. Enquanto a cabeça de M_2 estiver à direita da célula inicial, a cabeça de M_1 "trabalhará" na parte superior da sua fita e movimentar-se-á como a de M_2. Se a cabeça de M_2 estiver à esquerda da célula inicial, a cabeça de M_1 actuará na parte infeior da fita e movimentar-se-á em sentido contrário ao da cabeça de M_2. Note-se que a noção de pistas é apenas conceptual, não havendo nenhuma alteração na definição de TM básica.

Formalmente seja $M_1 = (S_1, \Sigma_1, \Gamma_1, \delta_1, s_1, [b, b], F_1)$ onde,

$$S_1 = \{[s, S], [s, I] \mid s \in S_2\} \cup \{s_1\}$$
$$\Sigma_1 = \{[\alpha, b] \mid \alpha \in \Sigma_2\}$$
$$\Gamma_1 = \{[X, Y] \mid X \in \Gamma_2, Y \in \Gamma_2 \cup \{\$\}\}$$
$$F_1 = \{[s, S], [s, I] \mid s \in F_2\}$$

A função $\delta_1 : S_1 \times \Gamma_1 \longrightarrow_p S_1 \times \Gamma_1 \times \{l, r\}$ é definida por:

1. Para cada $\alpha \in \Sigma_1 \cup \{b\}$,

$$\delta_1(s_1, [\alpha, b]) = ([s, S], [X, \$], r) \qquad \text{se } \delta_2(s_2, \alpha) = (s, X, r)$$

2. Para cada $\alpha \in \Sigma_1 \cup \{b\}$,

$$\delta_1(s_1, [\alpha, b]) = ([s, I], [X, \$], r) \qquad \text{se } \delta_2(s_2, \alpha) = (s, X, l)$$

3. Para cada $[X, Y] \in \Gamma_1$, com $Y \neq \$$ e $m = r$ ou $m = l$,

$$\delta_1([s, S], [X, Y]) = ([s', S], [Z, Y], m) \qquad \text{se } \delta_2(s, X) = (s', Z, m)$$

4. Para cada $[X, Y] \in \Gamma_1$, com $Y \neq \$$ e se $m = r$ então $\bar{m} = l$ e se $m = l$ então $\bar{m} = r$,

$$\delta_1([s, I], [X, Y]) = ([s', I], [X, Z], m) \qquad \text{se } \delta_2(s, Y) = (s', Z, \bar{m})$$

5.

$$\begin{aligned} \delta_1([s, S], [X, \$]) = \delta_1([s, I], [X, \$]) &= ([s', P], [Y, \$], r) \\ &\quad \text{se } \delta_2(s, X) = (s', Y, m) \end{aligned}$$

onde $P = S$ se $m = r$ e $P = I$ se $m = l$

□

Máquinas com k fitas RW Uma máquina de Turing com k fitas, $k > 1$, consiste num controlo finito com k cabeças, uma para cada uma das k fitas. Vamos considerar que cada fita é duplamente infinita. Num movimento, dependendo do estado do controlo finito e dos símbolos lidos em cada uma das k fitas pelas k cabeças, a máquina

1. muda de estado
2. escreve um símbolo em cada uma das células que estão debaixo de cada cabeça
3. move cada cabeça, independentemente, para a esquerda ou para a direita ou não mexe

Inicialmente a sequência de entrada encontra-se na primeira fita e todas as outras estão em branco. Os restantes conceitos são análogos aos duma TM com uma só fita.

Exercício 1.1.5 Descreva formalmente uma TM com k fitas e as noções de configuração, relação $\underset{M}{\longrightarrow}$ e de linguagem aceite por uma TM de k fitas.

Teorema 1.1.2 *Se uma linguagem L é aceite por uma TM com k fitas, então L é aceite por uma TM com uma única fita.*

Dem. Suponhamos que L é aceite por uma TM com k fitas M_1. Podemos construir uma TM, M_2 com uma só fita e com $2k$ pistas, duas pistas para cada fita. A primeira pista contém o conteúdo da fita correspondente de M_1 e a segunda está em branco excepto para a célula actualmente lida pela cabeça correspondente de M_1. Esta célula contém um indicador, que supomos ser o símbolo X. Por exemplo se $k = 2$, a fita de M_2 podia ser

	b	X	$\dots$	b	
	a_1	a_2	$\dots$	a_m	
	b	b	$\dots$	X	
	b_1	b_2	$\dots$	b_m	

Formalmente, ter-se-á $\Sigma_2 \subseteq (\{X, b\} \times \Sigma_1)^k$.

O controlo finito de M_2 guardará para além do estado de M_1 o número de indicadores à direita da cabeça de M_2 (n_i). Cada movimento de M_1 é simulado percorrendo a fita de M_2 da esquerda para a direita, para saber quais os símbolos actualmente lidos pelas k cabeças de M_1 e depois da direita para a esquerda, para actualizar esses valores e movimentar convenientemente os "indicadores". Inicialmente supõe-se que a cabeça de M_2 está na célula mais à esquerda que contém um indicador. Então, a cabeça de M_2 percorre a fita para a direita, e para cada célula com indicador "memoriza" o símbolo lido e diminui n_i. Esta memorização pode ser feita acrescentado a M_2 novos estados, cada um correspondendo a uma combinação de um estado de M_1 e a k-símbolos de Γ_1. Quando $n_i = 0$ então M_2 tem informação suficiente para determinar o movimento de M_1. Então, a cabeça de M_2 percorre a fita da esquerda para a direita, e sempre que encontra um indicador modifica o símbolo lido e movimenta o indicador de acordo com o movimento da cabeça correspondente em M_1. Quando já actualizou todos os indicadores, M_2 muda o estado correspondente a M_1. Se o novo estado de M_1 é um estado final, o de M_2 também o é. □

Exercício 1.1.6 Para $k = 2$ formaliza a máquina M_2 construída na demonstração anterior.

Exercício 1.1.7 Estima o número de movimentos efectuados por M_2 para cada movimento de M_1. Analisa a eficiência do uso de TMs com k fitas, $k > 1$, em relação a TMs só com uma fita.

Exercício 1.1.8 Descreve uma TM com 2 fitas que aceite a seguinte linguagem:

$$L = \{ww^r \mid w \in \{0,1\}^\star\}$$

e compara-a com a descrita no exercício 1.1.1.c)

1.1.3 Máquinas de Turing não determinísticas

Uma máquina de Turing é *não determinística* se dado um estado e um símbolo lido, existirem várias hipóteses para o movimento seguinte, isto é,

$$\delta \subseteq (S \times \Gamma) \times (S \times \Gamma \times \{l, r\})$$

Uma TM não determinística aceita a sequência de entrada se existe uma sequência de escolhas de movimentos que conduza a um estado de aceitação (análogamente se definem os conceitos de função calculada ou linguagem reconhecida). Em contraste, as TM definidas anteriormente denominam-se *determinísticas*.

Contudo a adição de não determinismo não torna estas TMs um modelo de computação mais poderoso que as TM determinísticas.

Teorema 1.1.3 *Se uma linguagem é aceite por uma máquina de Turing não determinística, M_1, então L é aceite por alguma máquina de Turing determinística, M_2.*

Dem. Para cada estado e símbolo de fita de M_1, existe um número finito de escolhas para o movimento seguinte. Estas escolhas podem ser numeradas 1, 2, ... Seja r o número máximo de escolhas de um par estado-símbolo. Então qualquer sequência finita de escolhas pode ser representado por uma sequência de dígitos de 1 a r.

Vamos construir M_2 com 3 fitas. A primeira contém a sequência de entrada. A segunda, gera sequências de dígitos de 1 a r dum modo sistemático. Por exemplo, por ordem crescente de comprimento e por ordem lexicográfica se de igual comprimento. Para cada sequência gerada na segunda fita, M_2 copia a sequência de entrada para a terceira fita e simula M_1 na terceira fita, usando a sequência da segunda fita para decidir os movimentos de M_1. Se M_1 atinge um estado de aceitação, então M_2 aceita. Se nenhuma sequência de escolhas conduz a M_1 aceitar, M_2 também não aceita. □

Exercício 1.1.9 Formaliza a geração de sequências de dígitos entre 1 e r, usada na demonstração do teorema anterior. Mostra como é que esta sequência pode codificar os movimentos de M_1.

1.2 Linguagem FOR($\mathbb{N}$)

Bibliografia [SvW88, Cap. 3], [Wei87, Cap. 1]

Considere a linguagem FOR($\mathbb{N}$) cuja sintaxe é definida pela seguinte gramática:

$$\begin{array}{lcl}
< \texttt{Prog} > & \rightarrow & \{(n, p, < \texttt{Insts} >) \mid \texttt{n} \geq \texttt{0},\ \texttt{p} \geq \texttt{n}+\texttt{1}\} \\
< \texttt{Insts} > & \rightarrow & < \texttt{Inst} >^* \\
< \texttt{Inst} > & \rightarrow & < \texttt{Var} > ++ \\
 & \rightarrow & \mid\ < \texttt{Var} > -- \\
 & \rightarrow & \mid \texttt{for}\ < \texttt{Var} >\ \{ < \texttt{Insts} > \}
\end{array}$$

Onde,

– considera-se que $\texttt{0} -- = \texttt{0}$.

– supõe-se que os valores das variáveis são inteiros não negativos.

– uma <Var> é uma letra eventualmente seguida de 1 ou mais letras ou símbolos.

– inicialmente todas as variáveis contêm 0 excepto as que correspondem aos valores dos dados. Estas são designadas por `x1`, `x2`,...`xn`. O número total de variáveis dum programa é p.

– a variável que define o resultado é `y`.

– o "<Insts>" dentro de uma instrução "`for`" é executado um número de vezes que é igual ao valor *inicialmente* contido na variável.

O parâmetro $\mathbb{N}$ pode ser omitido supondo-se que o valor das variáveis pertence a $\mathbb{N}$.
Um programa em FOR com n variáveis de entrada, calcula uma função $f : \mathbb{N}^n \longrightarrow \mathbb{N}$.

Exercício 1.2.1 Mostre que as seguintes funções podem ser implementadas em linguagem FOR.

1. Para cada valor k a função constante $f(x) = k$
2. A função $x1 + x2$.
3. A função $x1 - x2$ definindo-se $x1 - x2 = 0$ se $x2 > x1$.
4. A função $x1 * x2$.
5. A função $x1^{x2}$.

Exercício 1.2.2 Escreva um programa que implemente a instrução condicional genérica

$$\texttt{if}\ < \texttt{Var} >\ \texttt{then}\ < \texttt{Insts} >\ \texttt{else}\ < \texttt{Insts} >$$

supondo que o primeira sequência de instruções é executada se o valor da variável é diferente de 0.

Para formalizar a semântica operacional dum programa em FOR considere-se que cada programa é formado por um conjunto de linhas e cada linha contém:

- uma instrução de incremento ou decremento
- `for` < `Var` > {
- }

Supõe-se ainda que uma linha com o início duma instrução `for` tem à direita o número da linha da instrução que *fecha* o ciclo e que essa instrução tem ao número da linha do início do ciclo. Por exemplo:

```
1: for x1{   :3
2: y++
3: }         :1
```

Definição 1.2.1 *Seja $Q = (n, p, P)$ um programa em* FOR, *com linhas numeradas de 1 a t. Um* estado *de Q é um tuplo (l, x), $1 \leq l \leq t + 1$ e $x \in \mathbb{N}^p$. Para cada $1 \leq i \leq p$ x_i representa o valor da variável xi. O* estado *é inicial se $l = 1$ e $x_{n+1} = \ldots = x_p = 0$. O estado é final se $l = t + 1$.*

Definição 1.2.2 *Seja $Q = (n, p, P)$ um programa em* FOR, *com linhas numeradas de 1 a t. Seja S_Q o conjunto de todos os seus estados. Um estado $s_1 = (l, x)$ é transformado num passo de computação num estado $s_2 = (l', x')$, e denota-se $s_1 \underset{Q}{\longrightarrow} s_2$ se e só se:*

1. *l : $\mathtt{xj}--$ ou l : $\mathtt{xj}++$, $l' = l + 1$, $x'_i = x_i$ para todos $1 \leq i \leq p$ e $i \neq j$ sendo $x'_j = x_j - 1$ ou $x'_j = x_j + 1$, respectivamente*
2. *l : $\mathtt{for\ xj}\ \{\ \ : l_1$, então $l' = l + 1$ se $x_j \neq 0$ e $l' = l_1 + 1$ caso contrário. E $x' = x$.*
3. *l :$\}$: l_1 então $l' = l_1$ e $x'_j = x_j - 1$ sendo x_j a variável de ciclo da linha l_1*

Definem-se como habitualmente $\underset{Q}{\longrightarrow}^{+}$ e $\underset{Q}{\longrightarrow}^{\star}$.

Definição 1.2.3 *Seja $Q = (n, p, P)$ um programa em* FOR, *com linhas numeradas de 1 a t. Seja S_Q o conjunto de todos os seus estados. A função* estado seguinte *$F_Q : \mathbb{N}^{p+1} \longrightarrow \mathbb{N}^{p+1}$ é definida por:*

$$F_Q(s) = \begin{cases} s_1 & \text{se } s \underset{Q}{\longrightarrow} s_1 \\ s & \text{se } s \notin S_Q \text{ ou não existe tal } s_1 \end{cases}$$

Finalmente,

Definição 1.2.4 *Seja $Q = (n, p, P)$ um programa em* FOR, *com linhas numeradas de 1 a t. A semântica de Q é uma função $M_Q : \mathbb{N}^p \longrightarrow \mathbb{N}^p$ tal que $M_Q(x) = y$ se existe $(1, x) \underset{Q}{\longrightarrow}^{\star} (l, y)$ e $l = t + 1$, senão M_Q não está definida.*

Exercício 1.2.3 Defina uma expressão para a função calculada por um programa Q, usando a função M_Q.

Exercício 1.2.4 Mostre que qualquer programa Q em FOR termina sempre a sua execução, isto é que M_Q é uma função total.

Exercício 1.2.5 Mostre que, se $f(x)$ e $g(x)$ são implementáveis em FOR também o são as seguintes funções.

1. $f(x) + g(x)$, $f(x) * g(x)$ e $f(x)^{g(x)}$.
2. $g(f(x))$, a função composta.

Exercício 1.2.6 Considere de novo o programa que escreveu para o cálculo de $x1^{x2}$. Supondo que cada incremento ou decremento (isto é, instruções da forma $\mathtt{x}++$ ou $\mathtt{x}--$) demora 1 nanosegundo a ser executado e, não contabilizando os restantes testes, etc., calcule o tempo (em unidades apropriadas ao seu tamanho) que demoraria a calcular 100^{100}.

1.3 Funções Recursivas

Bibliografia [Rog67], [Phi92], [LP81, Cap. 5], [Yas71, Cap. 4, 6]

Vamos estudar as funções de $\mathbb{N}^n$ em $\mathbb{N}$.

1.3.1 Funções Primitivas Recursivas

Tentativa de englobar todas as funções totais num mesmo método de definição.

Definição 1.3.1 *1. Funções de base:*

- Função zero: $0 : \mathbb{N} \to \mathbb{N}$ *definida por*

$$0(x) = 0 \text{ para todo o } x \in \mathbb{N}$$

- Função sucessor: $s : \mathbb{N} \to \mathbb{N}$ *definida por*

$$s(x) = x + 1 \text{ para todo o } x \in \mathbb{N}$$

- Funções projecção: $\pi^n_m : \mathbb{N}^n \to \mathbb{N}$ *com* $n \geq m \geq 1$*) definida por*

$$\pi^n_m(x_1, \cdots, x_n) = x_m$$

2. Regras para formação de funções primitivas recursivas.

- Composição: *Dadas as* $n + 1$ *funções* $f : \mathbb{N}^n \to \mathbb{N}$ *e*

$$g_i : \mathbb{N}^m \to \mathbb{N} \text{ para } i = 1, \cdots, n$$

definimos a função composta $h : \mathbb{N}^m \to \mathbb{N}$ *por*

$$h(\mathbf{x}) = f(g_1(\mathbf{x}), \cdots, g_n(\mathbf{x}))$$

onde $\mathbf{x}$ *designa o "vector" de variáveis* $x_1, \cdots, x_m$*. A função* h *pode ser mais explicitamente designada por* $comp^n_m(f, g_1, \cdots, g_n)$*.*

- Recursão primitiva: *Dadas as duas funções com* $n \geq 1$

$$f : \mathbb{N}^n \to \mathbb{N}$$
$$g : \mathbb{N}^{n+2} \to \mathbb{N}$$

definimos por recursão primitiva a função $h : \mathbb{N}^{n+1} \to \mathbb{N}$ *por (definição indutiva)*

$$\begin{cases} h(\mathbf{x}, 0) = f(\mathbf{x}) \\ h(\mathbf{x}, y + 1) = g(\mathbf{x}, y, h(\mathbf{x}, y)) \end{cases}$$

A função h *pode ser mais explicitamente designada por* $prim^n(f, g)$*.*

A classe das funções primitivas recursivas (PR) é o menor conjunto de funções de $\mathbb{N}^n$ *em* $\mathbb{N}$ *que inclui as funções de base e que é fechado relativamente à composição e à recursão primitiva.*

É fácil verificar que todas as funções desta classse são totais.

Exercício 1.3.1 Mostra que as seguintes funções são primitivas recursivas. Dá uma justificação detalhada.

1. $f(x) = x$, a função identidade.
2. $f(x) = s(s(x))$.
3. $d(x) = 2$, função constante 2.
4. $f(x,y) = g(y,x)$ supondo que g é PR.

   Sugestão: Utiliza a composição.
5. $f(x,y) = x + y$.
6. $f(x,y) = x \times y$.
7. $f(x,y) = x^y$.
8. A função *não zero*, $sg(0) = 0$, $sg(x) = 1$ para $x \geq 1$.
   Sugestão: $sg(x+1) = 1$.
9. A função *teste zero*, $\overline{sg}(0) = 1$, $\overline{sg}(x) = 0$ para $x \geq 1$.
10. A função $f(x,y)$ cujo valor é igual a $g(x)$

    $$f(x,y) = g(x) \text{ para todo o } x \text{ e } y$$

    supondo que g é PR.
11. A função assim definida

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

    supondo que f e g são PR.

    Sugestão: Utiliza o resultado da alínea anterior e a recursão primitiva.
12. A função assim definida

    $$f(x,y) = \begin{cases} g(x) & \text{se } x \geq y \\ h(x) & \text{se } x < y \end{cases}$$

    supondo que f e g são PR.

    Sugestão: Utiliza o resultado da alínea anterior e a função $f'(x,y) = f(x, x - y + 1)$
13. A função assim definida

    $$f(x,y) = \begin{cases} g(x) & \text{se } x = y \\ h(x) & \text{se } x \neq y \end{cases}$$

    supondo que f e g são PR.

    Sugestão: Utiliza a função $f'(x,y) = f(x, (x-y) + (y-x))$
14. $f(x,y) : \mathbf{N}^2 \to \mathbf{N}$ sendo $f(0,y) = g(y)$ onde g é primitiva recursiva e $f(1,y) = h(y)$ onde h é primitiva recursiva. O valor de $f(x,y)$ para os restantes valores de x deve ser escolhido por forma que f seja primitiva recursiva. Sugestão: considere a função $x\,h(x) + (1-x)f(x)$
15. $f(x,y) = y - 1$, antecessor de y com um argumento extra x. Define-se $f(x,0) = 0$ para todo o x.
16. $f(x) = x - 1$, antecessor de y, definindo-se $f(0) = 0$.

    Sugestão: Utiliza a alínea anterior e a composição de funções.

17. $f(x, y) = x - y$, definindo-se $x - y = 0$ para $y \geq x$.

    Sugestão: Utiliza a alínea anterior.

18. $f(x, y) = x \, \mathtt{mod} \, y$ (resto da divisão inteira, definindo $a\mathbf{mod}0 = 0$ para todo o a).

    Sugestão: Utiliza o resultado da alínea anterior a recursão primitiva e a composição.

19. $f(x, y) = x/y$ (divisão inteira, definindo $a/0 = 0$ para todo o a).

    Sugestão: Utiliza o resultado da alínea anterior.

Exercício 1.3.2 Sendo $f : \mathbb{N} \longrightarrow \mathbb{N}$ define-se $f^\star : \mathbb{N}^2 \longrightarrow \mathbb{N}$, *exponenciação* de f, por:

$$f^\star(\mathbf{x}, y) = \underbrace{f(f(\cdots f(\mathbf{x}) \cdots))}_{y\text{vezes}}$$

Mostra que se f for PR então $f^\star$ é PR.

Exercício 1.3.3 (i) Mostra que a classe `FOR` de funções de $\mathbf{N}^n$ em $\mathbf{N}$ contém a classe PR.
Sugestão: Usa indução no número de regras utilizadas para a definição de uma função PR.

(ii) [$\star$] Mostra que para qualquer programa em `FOR` $Q = (n, p, P)$, M_Q corresponde a uma função primitiva recursiva. Concluí que todas as funções calculadas por um programa em *FOR* são PR. Sugestão: Define $T_Q(x)$ como o número de estados necessários para terminar uma computação que se inicie no estado $(1, x)$.

Exercício 1.3.4 Seja `P` um programa `FOR` que implementa uma função primitiva recursiva. Mostra que o programa `FOR`

$$\texttt{for}\ v\ \{\texttt{P}\}$$

onde v é uma variável que não ocorre em `P` (esta restrição não é importante) também implementa uma função primitiva recursiva. Sugestão: Considera uma função com mais um argumento (v) definida por recursão primitiva em v.

Função de Ackermann

[eAP95]

A função de Ackermann é uma função total computável que não é primitiva recursiva.

Definição 1.3.2

$$\begin{aligned} A(0, y) &= y + 1 \\ A(x + 1, 0) &= A(x, 1) \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)) \end{aligned}$$

Exercício 1.3.5 (i) Mostra que $A(x, y)$ está bem definida.

(ii) Calcula $A(1, 2)$, $A(1, 3)$, $A(5, 3)$. Podes fazer um programa para este fim.

(iii) Mostra que $A(n, m)$ é uma função total. Sugestão: Considera o seguinte predicado

$$\begin{array}{lll} P(x, y) & sse & A(x', y') \text{ termina para todo } (x', y') \text{ tal que} \\ & & x' < x \text{ ou } x' = x \text{ e } y' < y \end{array}$$

e prova $\forall\ x,\ y \in \mathbb{N},\ P(x, y)$ por dupla indução.

A classe das funções primitivas recursivas é muito vasta e inclui quase todas as funções de interesse prático. Mas, ao contrário daquilo que se pretendia não inclui todas as funções totais.

Pode demonstrar-se, por exemplo, que a função de Ackermann não é primitiva recursiva. Mais concretamente, existe um n tal que para todo o x é $f(x) < A(n,x)$. Nenhuma função primitiva recursiva cresce tão rapidamente como a função de Ackermann!

Exercício 1.3.6 Mostra que existem funções computáveis totais que não pertencem à classe PR.

1. Demonstrando que dada uma qualquer classe enumerável de funções totais computáveis é sempre possível definir uma função total computável não pertencente a essa classe.
2. Demonstrando que a função de Ackermann não é primitiva recursiva.

1.3.2 Funções Recursivas

O que falta às funções primitivas recursivas? Todas as funções são totais (análogo aos programas `FOR`). Vamos introduzir a *parcialidade* das funções. Se f é uma função parcial, designámos $f(x)\downarrow$ se f está definida para x e por $f(x)\uparrow$ caso contrário.

Definição 1.3.3 (Operador de minimização) *Seja* $f : \mathbb{N}^{n+1} \longrightarrow_p \mathbb{N}$. *Define-se* $g : \mathbb{N}^n \longrightarrow_p \mathbb{N}$ *por* minimização *como:*

$$g(\mathbf{x}) = menor\ y\ tal\ que\ f(\mathbf{x},y) = 0$$

Designa-se $g = \mu^n(f)$.

Dado que f pode não estar definido para alguns valores, tem-se que:

$$\begin{array}{rl} \mu^n(f)(\mathbf{x}) = y & \text{se } f(\mathbf{x},y) = 0 \text{ e} \\ & \forall z,\ 0 \leq z < y,\ f(\mathbf{x},z)\downarrow \text{ e } f(\mathbf{x},z) \neq 0 \\ \mu^n(f)(\mathbf{x})\uparrow & \text{caso tal } y \text{ não exista} \end{array}$$

Definição 1.3.4 *A classe das funções (parciais) recursivas (R) é o menor conjunto de funções de* $\mathbb{N}^n$ *em* $\mathbb{N}$ *que inclui as funções de base e que é fechado relativamente à composição, à recursão primitiva e à minimização. Uma função é total recursiva se pertence a R e é total.*

Exercício 1.3.7 Mostra que as seguintes funções são recursivas (R). Podes usar os resultados dos problemas anteriores.

1. $f(x,y) =$ menor y (se existir) tal que $f(x,y) = 2$ onde f é R.
2. $f(x,y) =$ menor y (se existir) tal que $f(x,y) = 0$ ou $g(x,y) = 0$ onde f e g são PR. Sugestão: Considera $\mu^1(f(x,y) \times g(x,y))$

Exercício 1.3.8 Mostra que as seguintes funções são recursivas.

(i) Qualquer função primitiva recursiva.

(ii) A função não definida em nenhum ponto, $\uparrow : \mathbf{N} \to \mathbf{N}$.

(iii) A função $f(x)$ não definida para $x \leq 4$ e com valor 0 nos restantes pontos.

Exercício 1.3.9 Considera uma operação de minimização cuja única diferença para μ^n é a seguinte: em vez de se testar $f(\overline{x},y) = 0$ testa-se $f(\overline{x},y) = 2$. Mostra que esta operação é equivalente a μ.

1.4 Linguagem WHILE(ℕ)

Bibliografia [SvW88, Cap. 3, 4], [Wei87, Cap. 1]

Considere a linguagem WHILE(ℕ) cuja sintaxe é definida pela seguinte gramática:

$$
\begin{array}{lcl}
<\texttt{Prog}> & \to & \{(n, p, <\texttt{Insts}>) \mid \texttt{n} \geq \texttt{0},\ \texttt{p} \geq \texttt{n}+\texttt{1}\} \\
<\texttt{Insts}> & \to & <\texttt{Inst}>^{\star} \\
<\texttt{Inst}> & \to & <\texttt{Var}>++ \\
 & & \mid <\texttt{Var}>-- \\
 & & \mid \texttt{if } <\texttt{Var}> \texttt{ then } <\texttt{Insts}> \texttt{ else } <\texttt{Insts}> \\
 & & \mid \texttt{while } <\texttt{Var}> \{<\texttt{Insts}>\}
\end{array}
$$

Onde,

- considera-se que $\texttt{0}-- = \texttt{0}$.
- supõe-se que os valores das variáveis (registos) são inteiros não negativos.
- uma $<\texttt{Var}>$ é da forma x0, x1, x2, ...
- inicialmente todas as variáveis (registos) contêm 0 excepto as que correspondem aos valores dos dados. Estas são designadas por x1, x2,...xn. O número total de variáveis do programa é p.
- a variável (registo) que define o resultado é $\texttt{x}_0$.
- o "$<\texttt{Insts}>$" dentro de uma instrução "while" é executado enquanto o valor da variável $<\texttt{Var}>$ for diferente de 0.

O parâmetro ℕ pode ser omitido supondo-se que o valor das variáveis pertence a ℕ.

Um programa em WHILE com n variáveis de entrada, calcula uma função $f : \mathbb{N}^n \longrightarrow \mathbb{N}$. Designámos essa classe de funções como a classe WHILE.

Para formalizar a semântica operacional dum programa em WHILE considere-se que cada programa é formado por um conjunto de linhas. A instrução condicional vai ser omitida, pois pode ser implementada com as restantes (Verifica!). Cada linha contém:

- uma instrução de incremento ou decremento
- while $<\texttt{Var}>$ {
- }

Supõe-se ainda que uma linha com o início duma instrução while tem à direita o número da linha da instrução que *fecha* o ciclo e que essa instrução, por sua vez, tem à direita o número da linha do início do ciclo. Por exemplo:

```
1:  while x1{   :3
2:  x1--
3:  }           :1
```

Definição 1.4.1 *Seja $Q = (n, p, P)$ um programa em* WHILE*, com linhas numeradas de* 1 *a* t*. Um* estado *de Q é um tuplo (l, x), $1 \leq l \leq t+1$ e $x \in \mathbb{N}^p$. Para cada $0 \leq i \leq p-1$ x_i representa o valor da variável xi. O* estado *é inicial se $l = 1$ e $x_0 = x_{n+1} = \ldots = x_{p-1} = 0$. O estado é final se $l = t+1$.*

Definição 1.4.2 *Seja $Q = (n, p, P)$ um programa em* `WHILE`*, com linhas numeradas de 1 a t. Seja S_Q o conjunto de todos os seus estados. Um estado $s_1 = (l, x)$ é transformado num passo de computação num estado $s_2 = (l', x')$, e denota-se $s_1 \underset{Q}{\longrightarrow} s_2$ se e só se:*

1. *l : `xj` $--$ ou l : `xj` $++$, $l' = l + 1$, $x'_i = x_i$ para todos $1 \leq i \leq p$ e $i \neq j$ sendo $x'_j = x_j - 1$ ou $x'_j = x_j + 1$, respectivamente*
2. *nos restantes casos $x' = x$ e se*
   (a) *l : `while xj {` $: l_1$, então $l' = l + 1$ se $x_j \neq 0$ e $l' = l_1 + 1$ caso contrário.*
   (b) *l :} : l_1 e l_1 : `while xj {` : l ocorre em Q (e $l_1 < l$), então $l' = l_1 + 1$ se $x_j \neq 0$ e $l' = l + 1$ caso contrário.*

Definem-se como habitualmente $\underset{Q}{\longrightarrow}^{+}$ e $\underset{Q}{\longrightarrow}^{\star}$.

Definição 1.4.3 *Seja $Q = (n, p, P)$ um programa em* `WHILE`*, com linhas numeradas de 1 a t. Seja S_Q o conjunto de todos os seus estados. A função* estado seguinte *$F_Q : \mathbb{N}^{p+1} \longrightarrow \mathbb{N}^{p+1}$ é definida por:*

$$F_Q(s) = \begin{cases} s_1 & \text{se } s \underset{Q}{\longrightarrow} s_1 \\ s & \text{se } s \notin S_Q \text{ ou não existe tal } s_1 \end{cases}$$

Uma computação dum programa Q pode ser identificada com a a sequência de estados $s_0, s_1, \ldots$ tal que $F_Q(s_i) = s_{i+1}\ \forall\, i \geq 0$. A computação termina no estado s_f, se $f = min\{j \mid F_Q(s_j) = s_j\}$. O comprimento da computação é definido por $T_Q(x) = f$, e é designado por complexidade temporal.

Definição 1.4.4 *Seja $Q = (n, p, P)$ um programa em* `WHILE`*, com linhas numeradas de 1 a t. A semântica de Q é uma função $M_Q : \mathbb{N}^p \longrightarrow \mathbb{N}^p$ tal que $M_Q(x) = y$ se existe $(1, x) \underset{Q}{\longrightarrow}^{\star} (t + 1, y)$, senão $M_Q(x) \uparrow$ (não está definida). A função calculada por Q é $f_Q : \mathbb{N}^n \longrightarrow \mathbb{N}$ definida por $f_Q(x) = \pi_1^p(M_Q(x))$.*

Teorema 1.4.1 *A classe de funções* `WHILE` *é equivalente à classe R.*

Dem. ($\Leftarrow$) Prova-se por indução na estrutura de R.

($\Rightarrow$) Seja $Q = (n, p, P)$ um programa `WHILE`. Pretende-se provar que $f_Q \in R$. Em primeiro lugar é necessário codificar os uplos de inteiros (estados) em inteiros, de tal modo que uma função recursiva nos números codificados corresponda à função F_Q. Iremos considerar a codificação de "Cantor" de $\mathbb{N}^k$ em $\mathbb{N}$, p^k e as correspondentes funções de projecção (que são PR).

Para facilitar a exposição, considerem-se as funções:

$$\begin{array}{rll} \iota : \mathbb{N} \longrightarrow \mathbb{N}^{p+1} & \text{tal que} & \iota < l, x_0, x_1, \ldots, x_{p-1} >= (l, x_0, x_1, \ldots, x_{p-1} \\ \lambda : \mathbb{N} \longrightarrow \mathbb{N} & \text{tal que} & \lambda < l, x_0, x_1, \ldots, x_{p-1} >=< x_0, x_1, \ldots, x_{p-1} > \\ \tau : \mathbb{N} \longrightarrow \mathbb{N}^p & \text{tal que} & \tau < x_0, x_1, \ldots, x_{p-1} >= (x_0, x_1, \ldots, x_{p-1}) \\ \sigma : \mathbb{N}^n \longrightarrow \mathbb{N} & \text{tal que} & \sigma(x_1, \ldots, x_n) =< 0, 0, x_1, \ldots, x_n, 0, \ldots, 0 > \\ & & \quad \text{e } (0, 0, x_1, \ldots, x_n, 0, \ldots, 0) \in \mathbb{N}^{p+1} \end{array}$$

Vamos definir uma função $sf : \mathbb{N} \longrightarrow \mathbb{N}$ tal que se $\iota(z) = s$ então

$$\iota(sf(z)) = F_Q(\iota(z)) \tag{1.1}$$

Para Q, com linhas numeradas de 1 a t, a função F_Q pode ser escrita:

$$F_Q(l, x_0, \ldots, x_{p-1}) = \begin{cases} (l, x_0, \ldots, x_{p-1}) & \text{se } l = 0 \\ F_1(l, x_0, \ldots, x_{p-1}) & \text{se } l = 1 \\ \vdots & \\ F_t(l, x_0, \ldots, x_{p-1}) & \text{se } l = t \\ (l, x_0, \ldots, x_{p-1}) & \text{se } l > t \end{cases}$$

onde, F_i é a transformação (passo de computação) correspondente à linha i do programa, de acordo com a definição de $\underset{Q}{\longrightarrow}$. Sejam $f_i < l, x_0, \ldots, x_{p-1} > = < l', x'_0, \ldots, x'_{p-1} >$ as funções corresponentes a cada F_i. Define-se para $z = < l, x_0, \ldots, x_{p-1} >$

$$sf(z) = z.\overline{sg}(l) + f_1(z).\overline{sg}(|l-1|) + \ldots + f_t(z).\overline{sg}(|l-t|) + z.\overline{sg}(t+1-l)$$

onde $\overline{sg}(0) = 1$, $\overline{sg}(x) = 0$ para $x \geq 1$. Se $l = 1$ então $sf(z) = f_1$, ..., se $l > t$ então $sf(z) = z$. É fácil verificar que sf é PR e satisfaz (1.1). E, também sf^y é PR , $\forall y \geq 0$. Seja, $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$ tal que $g(z, 0) = z$ e $g(z, y+1) = sf(g(z, y))$. Então, g é PR e $g(z, y) = sf^y(z)$. Defina-se $h : \mathbb{N}^2 \longrightarrow \mathbb{N}$ por:

$$h(z, y) = t + 1 - \pi_1^{p+1}(g(z, y))$$

Então, $T_Q(\tau(\lambda(z))) = \mu^2(h)(z)$, se tal valor existe. Finalmente,

$$M_Q(\tau(\lambda(z)) = \tau(g(z, \mu^2(h)(z)))$$

se z corresponder ao estado inicial isto é $z = < 0, 0, x_1, \ldots, x_n, 0, \ldots, 0 >$ e se $x \in \mathbb{N}^n$,

$$f_Q(x) = \pi_1^p(\tau(g(\sigma(x), \mu^2(h)(\sigma(x)))))$$

e portanto $f_Q \in R$. $\square$

Exercício 1.4.1 Mostra cuidadosamente que sf é PR.

Exercício 1.4.2 Supõe que o programa `P` implementa uma função recursiva com resultado em `x0` e dados em `x1` e `x2`. Mostra que o seguinte programa também implementa uma função recursiva (com resultado em `x0` e dados em `x1`)

```
P
x2 <- 0;
while x0 {
   P
   x2 <- x2+1;
}
x0 <- x2;
```

Exercício 1.4.3 (i) Mostra que a classe `WHILE` de funções de $\mathbf{N}^n$ em $\mathbf{N}$ contém a classe R. Sugestão: Usa indução no número de regras utilizadas para a definição de uma função R.

(ii) [$\star$] Mostra que a classe R contém a classe `WHILE`. Define uma semântica M_Q para um programa Q em `WHILE` análoga à definida para programas `FOR`. Define uma função $T_Q(x)$ que seja igual a l, o menor número de estados s_l tal que $F_Q(s_l) = s_l$ se tal valor existir sendo $s_0 = (1, x)$

Exercício 1.4.4 [$\star$] Mostra que para qualquer programa P escrito em `WHILE` existe um programa P1 equivalente a P que tem no máximo um ciclo `while`. Sugestão: Utiliza a indução na estrutura de P. É possível guardar em variáveis adicionais de P1 informação sobre o estado de execução dos "whiles" internos em P (que, pela hipótese indutiva não têm outros "whiles" dentro deles) e executar (em P1) um só ciclo externo.

Nota Este resultado pode traduzir-se em termos das funções recursivas no facto de só ser preciso aplicar uma vez a minimização. Está relacionado com o teorema da forma normal de Kleene.

1.5 Predicados recursivos

Bibliografia [Min74], [LP81, Cap. 5], [Hof79], [Yas71, Cap. 3,7], [HU79, Cap. 8]

Definição 1.5.1 *Dado um predicado (relação) $P \subseteq \mathbb{N}^k$, a função característica de P define-se por*[1]*:*

$$ch_P(\mathbf{x}) = \begin{cases} 1 & se\ P(\mathbf{x}),\ isto\ é,\ se\ P\ é\ verdade\ para\ \mathbf{x} \\ 0 & se\ não\ P(\mathbf{x}) \end{cases}$$

Se ch_P é recursiva (primitiva recursiva) P diz-se recursivo (primitivo recursivo). Se ch_P é total, P diz-se decidível.

Exercício 1.5.1 Mostre que as seguintes funções são PR:

(i) A função *não zero*, $sg(0) = 0$, $sg(x) = 1$ para $x \geq 1$.
Sugestão: $sg(x + 1) = 1$.

(ii) A função *zero*, $\overline{sg}(0) = 1$, $\overline{sg}(x) = 0$ para $x \geq 1$. Sugestão: $\overline{sg}(0) = s(0); \overline{sg}(x + 1) = 0(\cdots)$.

Exemplo 3 *A relação = é PR, dado que a sua função característica é*

$$ch_=(x, y) = 1 - sg((x - y) + (y - x))$$

Análogamente, a relação < é PR porque $ch_<(x, y) = \overline{sg}(y - x)$.

Lema 1.5.1 *Se Q e P são predicados PR também são PR os predicados $\neg P$, $Q \wedge P$ e $Q \vee P$.*

Exercício 1.5.2 Prova o lema anterior, mostrando que dados ch_P e ch_Q então $ch_{\neg P} = 1 - ch_P$, $ch_{Q \wedge P} = ch_Q \cdot ch_P$ e $ch_{Q \vee P} = sg(ch_Q + ch_P)$ e são PR se ch_P e ch_Q o forem.

Exercício 1.5.3 Mostra que se $g_1 \ldots g_m$ são $k + 1$-árias e PR, então

$$f(\mathbf{x}, m) = \Pi_{i=1}^{m} g(\mathbf{x}, i) = g(\mathbf{x}, 0) \cdot \ldots \cdot g(\mathbf{x}, m)$$

também o é. Sugestão: $f(\mathbf{x}, 0) = g(\mathbf{x}, 0)$ e $f(\mathbf{x}, m + 1) = f(\mathbf{x}, m) \cdot g(\mathbf{x}, m + 1)$

Definição 1.5.2 *Dado um predicado $Q \subseteq \mathbb{N}^k$ a quantificação existencial limitada de Q é qualquer predicado da forma $\exists i \leq m\, Q(\mathbf{x}m)$, e $\mathbf{x} \in \mathbb{N}^{k-1}$. Analogamente se define quantificação universal limitada.*

Lema 1.5.2 *Se Q é um predicado PR, então também o são as suas quantificações universais e existenciais.*

Exercício 1.5.4 Mostra o lema anterior. Sugestão: Use o exercício anterior.

Exercício 1.5.5 Mostra que se $g_1 \ldots g_l$ são funções k-árias e PR, e $P_1 \ldots P_l$ são predicados PR tal que para cada $\mathbf{x} \in \mathbb{N}^k$ exactamente um e só um $P_i(\mathbf{x})$ é verdade, então a função f definida por:

$$f(\mathbf{x}) = \begin{cases} g_1(\mathbf{x}) & se\ \ P_1(\mathbf{x}) \\ \vdots & \\ g_l(\mathbf{x}) & se\ \ P_l(\mathbf{x} \end{cases}$$

também é PR. Sugestão: Usa as funções ch_{P_i}.

[1] Esta definição está de acordo com associar 1 ao valor lógico verdade

Definição 1.5.3 *Dado $k \geq 0$ e Q um predicado $k+1$-ário, a minimização limitada de Q é uma função $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ tal que:*

$$f(\mathbf{x}, m) = \begin{cases} \textit{menor } p,\ 0 \leq p \leq m \textit{ tal que } Q(\mathbf{x}m) & \textit{se existir} \\ 0 & \textit{caso contrário} \end{cases}$$

Escreve-se $f(\mathbf{x}, m) = \mu p \leq m[Q(\mathbf{x}p)]$.

Lema 1.5.3 *Se Q é um predicado PR, então a sua minimização limitada também o é.*

Exercício 1.5.6 Prova o lema anterior. Sugestão: Considera a função h:

$$h(\mathbf{x}, m, p) = \begin{cases} p & \text{se } \exists i \leq m[Q(\mathbf{x}i)] \\ m+1 & \text{se } Q(\mathbf{x}m+1) \text{ e é falso } \exists i \leq m[Q(\mathbf{x}i)] \\ 0 & \text{é falso } \exists i \leq m[Q(\mathbf{x}i])\end{cases}$$

Mostra que h é PR e f pode ser obtida de h por recursão primitiva.

1.6 Codificações e Numerações de Gödel

Para codificar k-tuplos de números inteiros em inteiros pode-se usar as funções $p^k : \mathbb{N}^k \longrightarrow \mathbb{N}$ (Cantor):

$$\begin{aligned} p^1(x) &= x \\ p^{k+1}(x_1, \ldots, x_{k+1}) &= p(x_1, p^k(x_2, \ldots, x_{k+1})) \end{aligned}$$

onde $p(x, y) = \frac{(x+y)(x+y+1)}{2} + x$

Lema 1.6.1 *Para todo o k:*

1. *p^k é bijectiva.*
2. *p^k é PR*
3. *$\forall i,\ 1 \leq i \leq k,\ p^k_i = \pi^k_i (p^k)^{-1}$*

Escreve-se $p^k(x_1, \ldots, x_k) =< x_1, \ldots, x_k >$ e $(< x_1, \ldots, x_k >)_i = x_i$.

Exercício 1.6.1 Prova o lema 1.6.1. Sugestão: Nota que $p^{-1}(z) = (u(z), d(z) - u(z))$ onde $d(z) = (\mu y \leq z[p(0, y) > z]) - 1$ e $u(z) = z - p(0, d(z))$.

Usando esta codificação uma função $f : \mathbb{N}^n \longrightarrow \mathbb{N}^k$ pode ser codificada como uma função $h : \mathbb{N} \longrightarrow \mathbb{N}$. Tem-se:

$$\begin{aligned} cod(f)(x) &= p^k(f((p^n)^{-1}(x))) \\ cod^{-1}(h)(x) &= = (p^k)^{-1}(h(p^n(x))) \end{aligned}$$

Estas codificações permitem enumerar todas os k-tuplos de números inteiros.

Exercício 1.6.2 Define uma função que codifique uma qualquer sequência finita de inteiros em inteiros. Sugestão: Define $L(x)$ a função comprimento de $x \in \mathbb{N}^\star$.

Genericamente sendo Σ um conjunto finito ou numerável de símbolos, funções bijectivas computáveis de $\Sigma^\star$ em $\mathbb{N}$ (ou f injectiva de $\Sigma^\star$ em $\mathbb{N}$ e f^{-1} sobrejectiva, ou de domínio restrito ao contradomínio de f, também computável) denominam-se *numerações de Gödel*.[2]

Vamos dar alguns exemplos:

1. Seja $\Sigma = \{d_1, \ldots, d_{n-1}\}$. Então qualquer $w \in \Sigma^\star$, $w = d_{i_1} \ldots d_{i_k}$ pode ser representado como um número inteiro na base $n = \mid \Sigma \mid +1$:

$$c(w) = \sum_{j=1}^{k} n^{k-j} i_j$$

   Por exemplo, se $\Sigma = \{a, b, c\}$ então

$$c(aabb) = 4^3 \times 1 + 4^2 \times 1 + 4^1 \times 2 + 2 = 90$$

   Verifica que nesta codificação nem todos os inteiros são códigos válidos!

2. Outra hipótese é ordenar os elementos de $\Sigma^\star$ pelo comprimento e se tiverem o mesmo comprimento por ordem lexicográfica. Dado $\Sigma = \{d_1, \ldots, d_n\}$ vem

$$\begin{aligned} c(\epsilon) &= 0 & \\ c(d_i) &= i & \text{para todo } 1 \leq i \leq n \\ c(wd_i) &= nc(w) + i & \text{para todo } w \in \Sigma^\star \text{ e } i,\ 1 \leq i \leq n \end{aligned}$$

   Então, $c(d_{i_1} \ldots d_{i_k}) = n^{k-1} i_{i_1} + \ldots + n i_{k-1} + i_k$. No caso do exemplo anterior ter-se-ia:

0	1	2	3	4	5	6	...	44	...
ϵ	a	b	c	aa	bb	cc	...	aabb	...

3. Baseada na decomposição única em factores primos[3] de qualquer $n > 1$, tem-se a seguinte codificação $c : \Sigma^\star \longrightarrow \mathbb{N}$, sendo $\Sigma = \{d_0, d_1, \ldots\}$:

$$\begin{aligned} c(d_i) &= p_i & \text{onde } p_i \text{ é o } i\text{-ésimo primo, sendo } p_0 = 2 \\ c(d_{i_0} d_{i_1} \ldots d_{i_k}) &= 2^{c(d_{i_0})} 3^{c(d_{i_1})} \ldots p_k^{c(d_{i_k})} & \end{aligned}$$

   Para o exemplo anterior tem-se que $c(aabb) = 2^2 3^2 5^3 7^3 = 1543500$

   Nota que esta codificação não é sobrejectiva em $\mathbb{N}$.

Exercício 1.6.3 Define uma numeração de Gödel para programas $WHILE$, tal que g represente o programa P_g que calcula f_g. Isso permite em particular enumerar a classe R de funções. Escreve um programa em $WHILE$ que dado g e $x \in \mathbb{N}^n$ simule o programa P_g.

Exercício 1.6.4 [Máquina Universal de Turing]

a) Inventa uma descrição genérica $m : TM \to \{0, 1\}^*$ de máquinas de Turing utilizando apenas o alfabeto $\{0, 1\}$.

b) Descreve em linguagem C uma função $f(m, x)$ onde m é um "string" que representa uma máquina de Turing M (isto é, $m = d(M)$), x é o "string" dos dados e

[2] Pelo facto de Kurt Gödel (1930) ter usado pela primeira vez estas codificações para aritematizar a teoria axiomática dos números, isto é, codificar em inteiros proposições, teoremas e demonstrações

[3] Esta é semelhante à usada por Kurt Gödel.

(a) f retorna SIM se M aceita x.

(b) f retorna NAO se M rejeita x.

(c) f entra em ciclo infinito se M com dados x entra em ciclo infinito.

c) Dadas duas sequências x e y define uma função $c(x, y)$ cujo resultado é uma sequência de 0s e 1s a partir da qual se pode extrair x e y. Obviamente a concatenação xy não satisfaz.

d) Repete a questão 5 utilizando uma função de um só argumento, $g(c(m, x))$ onde c é a função descrita em c).

e) Mostra que existe uma máquina de Turing U que faz o "mesmo" da função g da alínea anterior (inicialmente a fita contém $c(m, x)$).

f) Define a máquina de Turing U.

g) Mostra que U imita qualquer máquina com quaisquer dados no sentido seguinte

(a) M/x termina em s_1 sse $U/c(d(M, x))$ termina em s_1.

(b) M/x termina em s_2 sse $U/c(d(M, x))$ termina em s_2.

(c) M/x não termina sse $U/c(d(M, x))$ não termina.

Nos problemas seguintes supõe-se que as instâncias de máquinas de Turing são codificadas em palavras com símbolos de um alfabeto Σ. É evidentemente decidível se um determinado $x \in \Sigma$ representa ou não a codificação de uma máquina de Turing. Essa decidibilidade corresponde a uma função (total) $t : \Sigma^* \to \{F, V\}$.

Exercício 1.6.5 Mostra que existe uma função total $f: \mathbf{N} \to \Sigma^*$ cujo resultado é a máquina de Turing de ordem n (na ordem lexicográfica).
Nota: descrever uma implementação de f numa linguagem de alto nível onde se supõe a existência da função d referida.

Exercício 1.6.6 Mostra que existe uma função total $f: \Sigma^* \to \mathbf{N} \cup \{-1\}$ cujo resultado é n se x representa a máquina de ordem n e -1 se x não representa nenhuma máquina de Turing.

Exercício 1.6.7 Mostre que é possível codificar o conjunto das máquinas de Turing por forma que o contradomínio seja Σ^*, isto é, por forma que todo $x \in \Sigma^*$ seja a representação de uma máquina de Turing.
Sugestão: Seja y a codificação usual da máquina. Seja n o seu número. A codificação sobrejectiva z é a sequência cuja ordem lexicográfica é n.

Exercício 1.6.8 Mostra que não se perde generalidade se numa codificação supusermos $||\Sigma|| = 2$ ou $||\Sigma|| = 1$.

1.7 Computabilidade: Sumário de Alguns Resultados

Bibliografia [Pap94, Cap. 3], [HU79, Cap. 8], [LP81, Cap. 6], [Phi92]

1.7.1 Nota sobre a enumerabilidade dos programas

Um modelo de computação tem, normalmente, uma infinidade de instâncias possíveis: programas de uma linguagem, casos particulares de máquinas de Turing (TM), etc.

Modelo de computação	**Instâncias**
Autómatos finitos	Descrição de um FA particular
Máquinas de Turing	Descrição de uma TM particular
Linguagem "While"	Programa particular
...	...

Fazem-se sempre as seguintes hipóteses mais que razoáveis:

1. Cada instância de um modelo é codificada num alfabeto finito Σ. Por exemplo: $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, |, (,), \mathtt{0}, \mathtt{1}, \mathtt{2}\}$, etc. Note, em particular, que isto implica que a descrição de qualquer instância é *finita*.

2. Para cada modelo é decidível se uma dada sequência $x \in \Sigma^\star$ representa ou não uma instância desse modelo. Isto corresponde, para as linguagens de programação a dizer se um dado texto representa um programa o que pode corresponder a dizer que não existem erros sintácticos. Neste caso, a decisão pode ser efectuada pela parte de análise sintáctica de um compilador.

Uma consequência das hipóteses anteriores é o seguinte

Teorema 1.7.1 *O conjunto das instâncias de um determinado modelo é lexicograficamente enumerável.*

A demonstração pode basear-se na existência de uma DTM que enumera lexicograficamente (na fita) todas as sequências do alfabeto (por exemplo, para $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ temos λ, 0, 1, 00,...). Após gerar cada uma dessas sequências x, aplica-se o suposto método de decisão. Se a resposta é sim (isto é, se x é a descrição de uma instância) escreve-se x (na parte da fita destinada ao "output"); se a resposta é não, não se escreve x. $\Box$.

Os seguintes resultados são fáceis de demonstrar.

Corolário 1.7.1 *Existe uma bijecção n entre o conjunto* **D** *das descrições de instâncias de um certo modelo e* **N**. *Tanto n como n^{-1} são computáveis.*

Por outras palavras, dado um inteiro i podemos saber efectivamente qual a descrição que lhe corresponde ($n^{-1}(i)$) e, reciprocamente, dada uma descrição $x \in \Sigma^*$ podemos saber a sua ordem na enumeração (lexicográfica), $n(x)$.

Cada instância P dum modelo de computação pode ser visto como uma função $f : \mathbb{N} \longrightarrow \mathbb{N}$. Normalmente f será uma função parcial, porque P pode não terminar para todos os dados.

1.7.2 Modelos universais

Tese 1.7.1 Tese de Church–Turing
Uma função parcial é efectivamente computável (isto é, computável por um qualquer método algorítmico, humano ou não) se for computável por uma máquina de Turing determinística (DTM).

Obviamente esta tese não se pode demonstrar. As razões pelas quais a quase totalidade dos investigadores pensa que é verdadeira são de vários tipos.

1. As DTMs são tão gerais que parecem poder simular qualquer possível computação.

2. Nunca ninguém descobriu um modelo "algorítmico" mais geral que as DTMs.

3. Vários investigadores, ao estudar modelos de computação suficientemente gerais, têm sempre chegado a "algo" que nunca é mais gera que as DTMs e, muitas vezes, *é equivalente* às DTMs. Ver o Teorema 1.7.2.

Um modelo de computação diz-se *universal* se toda a função efectivamente computável o for nesse modelo. Um modelo de computação diz-se *total* se todas as funções computáveis nesse modelo são totais.

Teorema 1.7.2 Modelos universais
Os seguintes modelos são equivalentes no sentido em que qualquer função que possa ser caracterizada num deles pode ser caracterizada em qualquer dos outros. Qualquer destes modelos é pois universal.

- *DTM – Máquinas de Turing determinísticas com uma fita RW.*

- *DTM – Máquinas de Turing determinísticas com uma fita RW e alfabeto de fita $\{b, 1\}$.*

- *DTM – Máquinas de Turing determinísticas com uma fita RW, uma fita R (dados) e uma fita W (resultados).*

- *DTM – Máquinas de Turing determinísticas com um número fixo de fitas RW.*

- *NDTM – Máquinas de Turing não determinísticas com uma fita RW.*

- *Programas na linguagem WHILE.*

- *Funções recursivas.*

A demonstração é trabalhosa embora não excessivamente complicada. No curso apenas foi apresentada parte.

Existem outros modelos de computação (ou sistemas formais) que poderiam ser incluídos na lista anterior.

1.7.3 Modelos totais

Teorema 1.7.3 *Os seguintes modelos (definidos no curso) são equivalentes no sentido em que qualquer função que possa ser caracterizada num deles pode ser caracterizada em qualquer dos outros. Todas essas funções são totais.*

- *Programas na linguagem FOR.*

- *Funções primitivas recursivas.*

Teorema 1.7.4 *Nenhum modelo total inclui todas as funções totais computáveis.*

Para modelos particulares a demonstração pode ser efectuada exibindo uma função total computável que não caracterizável no modelo (exemplo: a função de Ackermann para os dois modelos referidos acima). Para o caso geral utiliza-se um um argumento "diagonal" baseado na enumerabilidade das instâncias do modelo.

Dem. [(Caso geral)] Seja Σ um alfabeto finito e $L \subseteq \Sigma^\star$ a linguagem em que são codificadas as instâncias (que designaremos por programas) desse modelo. Então podemos enumerar todos os seus programas P_0, P_1, ... Suponhamos que L é universal e que todos os seus programas são totais. Seja g_n a função total calculada por P_n. Utilizamos uma *construção diagonal* para obter uma contradição.

Seja $f : \mathbb{N} \longrightarrow \mathbb{N}$ definida por $f(n) = g_n(n) + 1$. Então f é total se g_n o for. E f é computável: dado n gera-se a lista de programas até P_n; em seguida, calcula-se $P_n(n)$ e adiciona-se 1 ao resultado. Então, existe m tal que P_m calcula f, isto é, $f(m) = g_m(m)$. Contradição (que resultou de se supor que L é total e calcula todas as funções totais).

As funções g_n podem ser tabeladas:

$$\begin{array}{cccc}
g_0(0) & g_0(1) & g_0(2) & \ldots \\
g_1(0) & g_1(1) & g_1(2) & \ldots \\
g_2(0) & g_2(1) & g_2(2) & \ldots \\
\ldots & \ldots & \ldots & \ldots
\end{array}$$

Para todo o n, o valor de f é diferente do da função enumerada pela diagonal principal. Logo não pode corresponder a nenhuma linha da tabela. Então a enumeração das funções totais não é completa :existe uma função total computável que não é computável por L. □

Em conclusão:

- um modelo de computação é universal mas admite funções parciais computáveis nesse modelo.
- um modelo de computação é total mas existem funções totais que não são computáveis pelo modelo.

1.7.4 Existência de funções não computáveis

Teorema 1.7.5 *Existem funções não computáveis.*

Dem. Utiliza-se também uma construção diagonal. Seja L a linguagem associada a um modelo de computação universal, P_1,...uma enumeração das suas instâncias e g_1, ...as funções associadas a cada instância. Pelo teorema 1.7.4 sabemos que g_i podem não ser totais. Defina-se f por:

$$f(n) = \begin{cases} g_n(n) + 1 & \text{se } g_n \text{ está definida para } n \\ 0 & \text{caso contrário} \end{cases}$$

Suponhamos que f é computável. Então, $f = g_m$ para algum m. Mas considerando $f(m)$ obtém-se uma contradição. □

1.7.5 Forma Normal de Kleene

Dado um modelo de computação e uma instância I com dados $\mathbf{x}$ pode-se associar um predicado T tal que:

$$T(I, \mathbf{x}, y) \quad \leftrightarrow \quad y \text{ é uma computação completa de } I \text{ com dados } x$$

e uma função $U(y)$ que é resultado da computação.

Resulta da demonstração do teorema fundamental 1.7.2, que através das codificações (de Gödel) de linguagens, instâncias[4] e computações em inteiros é possível considerar T um predicado definido num tuplo de inteiros.

Teorema 1.7.6 Forma Normal de Kleene *Existe um predicado PR T_m e uma função PR U tal que se $f : \mathbb{N}^m \longrightarrow_p \mathbb{N}$ é computável então existe n tal que para todo o* $\mathbf{x}$

$$\begin{aligned} f(\mathbf{x}) \downarrow \quad &\leftrightarrow \quad \exists y\, T_m(n, \mathbf{x}, y) \\ &\qquad e\ f(\mathbf{x}) = U(\mu y . T_m(n, \mathbf{x}, y)) \end{aligned}$$

Em particular, tem-se que qualquer função recursiva pode ser definida usando o operador de minimização apenas uma vez.

Dado um modelo de computação e uma codificação, n é o índice (código) duma instância. Por exemplo, dada uma codificação para máquinas de Turing n é o número associado a uma TM particular, ou eventualmente, a nenhuma (note-se que neste caso T está definido e U é a função parcial totalmente indefinida).

Para cada $n \in \mathbb{N}$, $\{n\}_m(\mathbf{x})$ denota a função m-ária $U(\mu y . T_m(n, \mathbf{x}, y)$. Temos assim uma enumeração das funções recursivas m-árias. Note-se que $\{n\}$ também pode ser interpretada como a máquina de Turing de índice n, para uma dada codificação ...

1.7.6 Máquinas de Turing universais e programas universais

Teorema 1.7.7 *Para cada m existe uma função computável Ψ_m tal que $\{n\}_m(\mathbf{x}) = \Psi_m(n, \mathbf{x})$*

A demonstração deste teorema é imediata pelo teorema 1.7.6. Basta considerar cada $\{n\}_m(\mathbf{x})$ como função computável de m e $\mathbf{x}$.

Em termos de máquinas de Turing, o teorema garante a existência de uma máquina U que tendo como dados uma descrição duma máquina M e $\mathbf{x}$, simula M e tem como resultado o resultado de M com dados $\mathbf{x}$. U denomina-se uma máquina universal de Turing (UTM). Ver, por exemplo, [Dew89, Cap. 48] para a descrição duma UTM.

1.7.7 Linguagens, funções características e problemas

Vimos que as máquinas de Turing, para além de outras variantes, podiam diferir quanto à finalidade: *reconhecer* (ou decidir) linguagens, *aceitar* linguagens ou *calcular* funções. No primeiro caso dizia-se que a linguagem reconhecida (decidida) era *recursiva* e no segundo que era *recursivamente enumerável*. Associada a qualquer linguagem $L \subseteq \Sigma^\star$, para algum alfabeto finito Σ, podemos definir uma função (total) característica de L, $ch_L : \Sigma^\star \longrightarrow \{0,1\}$:

$$ch_L(x) = \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$$

[4] Máquinas de Turing, se o modelo é TM, programas se o modelo é a linguagem `WHILE` ou outra linguagem de programação universal ou funções se o modelo forem as funções recursivas.

Se uma linguagem L for reconhecida (decidida) por uma máquina de Turing M, então também existe uma máquina de Turing M' que calcula a sua função característica (e vice-versa). Se uma linguagem L for aceite por uma máquina de Turing M, então também existe uma máquina de Turing M' que calcula a sua função (parcial) semi-característica, s_L (e vice-versa) onde, s_L é definida por:

$$ch_L(x) = \begin{cases} 1 & x \in L \\ \uparrow & x \notin L \end{cases}$$

isto é, M' pára com o valor 1 se M parar e M' não pára se M não parar.

Então, recordando o que foi estudado para predicados (secção 1.5), para uma dada linguagem L (ou predicado P) associa-se o seguinte problema de *decisão* : $x \in L$? ($P(x)$ é verdade). Um problema de decisão é *solúvel* ou *decidível* se L é recursiva e *insolúvel* ou *indecidível*, caso contrário. Por abuso de notação, podemos, para *linguagens* considerar equivalentes as noções de recursividade, decidibilidade ou solubilidade.

Por outro lado, um problema $x \in L$? *indecidível* será semi-decidível se a linguagem L for recursivamente enumerável.

Finalmente, e mais uma vez, dada a correspondência entre $\Sigma^\star$ e $\mathbb{N}$ podemos restringir-nos a estudar funções em $\mathbb{N}$ (ou $\mathbb{N}^m$) e subconjuntos de $\mathbb{N}$.

1.7.8 Indecidibilidade do problema da paragem ("halting problem")

A existência de problemas indecidíveis é uma consequência do teorema 1.7.4. O problema da paragem consiste em saber se uma máquina de Turing M pára com dados x, isto é, dados n e m $\boxed{\{n\}(m)\downarrow}$?

Teorema 1.7.8 Problema da paragem
O problema $\boxed{\{n\}(m)\downarrow}$ *é indecidível.*

Dem. Seja $H = \{p(n,m) \mid \{n\}(m)\downarrow\}$, onde p é a função de codificação de Cantor. Suponhamos por absurdo que existe uma máquina de Turing M_H, com índice h que calcula a função característica de H. Modifique-se M_H de modo a obter uma máquina D, de índice d, que se comporte do seguinte modo: com dados n simule M_H com dados $p(n,n)$ até que M_H pare (que acontece sempre). Se M_H parou com o valor 0, então D entra num estado que move sempre a cabeça para a direita, logo não termina. Se M_H parou com o valor 0, então D pára. Isto é:

$$\begin{array}{lcl} \{d\}(n)\uparrow & \leftrightarrow & \{h\}(p(n,n)) = 1, \text{ isto é, } \{n\}(n)\downarrow \\ \{d\}(n)\downarrow & \leftrightarrow & \{h\}(p(n,n)) = 0, \text{ isto é, } \{n\}(n)\uparrow \end{array}$$

Qual é o valor de $\{d\}(d)$? Se $\{d\}(d)\uparrow$, pela definição de D, M_H com dados $p(d,d)$ pára com o valor 1 isto é, $d(d)\downarrow$. Por outro lado, se $\{d\}(d)\downarrow$, pela definição de D, M_H com dados $p(d,d)$ pára com o valor 0 isto é, $d(d)\uparrow$. Temos então uma contradição, que resultou de supor que H era reconhecido por alguma TM. $\square$

No curso, foi demonstrado que o problema mais restrito $\boxed{\{n\}(n)\downarrow}$, também é indecidível.

Teorema 1.7.9 Problema da auto-paragem
O problema $\boxed{\{n\}(n)\downarrow}$ *é indecidível.*

Assim, podia-se ter usado este resultado para mostrar o teorema 1.7.8 (compara as duas demonstrações!).

1.7.9 Conjuntos recursivos e recursivamente enumeráveis

As definições seguintes correspondem às que foram dadas para linguagens na secção 1.7.7.

Para os resultados que são enunciados sem demonstração esta ou foi feita no curso ou é deixada como exercício.

Definição 1.7.1 *Um conjunto $A \subseteq \mathbb{N}$ diz-se* recursivo *se existe uma função recursiva total $ch_A : \mathbb{N} \longrightarrow \mathbb{N}$ tal que:*

$$ch_A(n) = \begin{cases} 1 & n \in A \\ 0 & n \notin A \end{cases}$$

ch_A denomina-se função característica de A.

Definição 1.7.2 *Um conjunto $A \subseteq \mathbb{N}$ diz-se* recursivamente enumerável (r.e.) *se a sua função semi-característica s_A:*

$$s_A(n) = \begin{cases} 1 & n \in A \\ \uparrow & n \notin A \end{cases}$$

é (parcial) recursiva.

Proposição 1.7.1 *Se $A \subseteq \mathbb{N}$ é recursivo então A é r.e.*

Proposição 1.7.2 *O conjunto $H = \{p(n,m) \mid \{n\}(m)\downarrow\}$ é r.e.*

Dem. É necessário construir um máquina de Turing que calcule s_H. Basta considerar uma máquina de Turing (universal) U_H que dado $p(n,m)$, uma descrição da máquina de Turing M de índice n e dados m, simule o comportamento de M com dados m. Se M parar, U_H pára com o valor 1. Se M não parar, U_H não pára. □

Análogamente,

Proposição 1.7.3 *O conjunto $K = \{n \mid \{n\}(n)\downarrow\}$ é r.e.*

Teorema 1.7.10 *O conjunto $A \subseteq \mathbb{N}$ é r.e. se e só se A é o domínio duma função recursiva parcial.*

Dem. ($\Rightarrow$) Basta notar de s_A é uma função parcial recursiva.

($\Leftarrow$) Seja f tal que $Dom(f) = A$ e f recursiva parcial. É fácil calcular s_A usando f. Dado $n \in \mathbb{N}$ se $f(n)\downarrow$ então $s_A(n) = 1$. Pela Tese de Church s_A é recursiva. □

Teorema 1.7.11 *O conjunto $\emptyset \neq A \subseteq \mathbb{N}$ é r.e. se e só se é o contradomínio de uma função recursiva total.*

Dem. ($\Rightarrow$) Seja $\{n\}$ uma função unária recursiva parcial e D_n o seu domínio. Pelo Teorema 1.7.10, existe $\{n\}$ tal que $A = D_n$. Pela Forma normal de Kleene, $x \in A$ sse $\exists y T(n,x,y)$. Tome-se um qualquer $k \in A$. Define-se $f : \mathbb{N} \longrightarrow \mathbb{N}$ por

$$f(z) = \begin{cases} x & z = p(x,y) \text{ e } T(n,x,y) \\ k & \text{caso contrário} \end{cases}$$

e $f(0) = 0$. A função f é recursiva e total. Se $x \in CD_f$ então $\exists y T(n,x,y)$, logo $x \in A$. Se $x \in A$ então existe algum y tal que $T(n,x,y)$ e $f(p(x,y)) = x$. Donde $A = CD_f$.

($\Leftarrow$) Seja $f : \mathbb{N} \longrightarrow \mathbb{N}$ função recursiva total e $A = CD_f$. Defina-se $g(x) = \mu z(f(z) = x)$. A função g é recursiva (porquê?). E, $g(x)\downarrow$ sse $x \in A$. Pelo Teorema 1.7.10, A é r.e..

□

A denominação "recursivamente enumeráveis" provém do resultado seguinte. Uma máquina de Turing M enumera uma linguagem $L \subseteq \Sigma^\star$, para algum alfabeto Σ, se durante a computação de M com a fita inicialmente vazia, os elementos de L são escritos na fita. Formalmente, se $\forall x \in L,\ s_0\epsilon \underset{M}{\longrightarrow}^\star ybxsb$, para algum estado s e sequência y. Neste caso designa-se $L = E(M)$.

Teorema 1.7.12 *O conjunto $A \subseteq \mathbb{N}$ é r.e. se e só se existe uma máquina de Turing M que enumera os seus elementos, isto é, $A = E(M)$.*

Dem. ($\Leftarrow$) Suponhamos que $A = E(M)$, para alguma TM M. Então, A é aceite por uma máquina de Turing M' que com dados n simula M com a fita inicialmente vazia. Se durante a computação de M, n é enumerado, então M' pára com o valor 1. Caso contrário M' não termina. (Nota que se $n \in A$ então n tem de ser enumerado na fita de M).

($\Rightarrow$) Suponhamos que A é r.e. Então, existe uma máquina de Turing M que calcula a sua função semi-característica. Pretende-se obter uma máquina M' tal que $A = E(M')$, isto é, que enumere os elementos de A. M' não pode simular directamente M para cada um dos $n \in \mathbb{N}$, porque se $n \notin A$, M não pára. (Imagina que $0 \notin A$ e $1 \in A$). A técnica que será usada denomina-se "dovetailing"[5] – computações intercaladas. M' com a fita inicialmente vazia opera da seguinte forma: para cada $n = 0, 1, 2, \ldots$ simula M com dados n, intercaladamente e cada um para n passos de computação. Por exemplo: simula o 1º passo de computação de M para 0; simula o 1º passo de computação de M para 1; simula 2º passo de computação para 0; simula o 1º passo de computação de M para 2; ... Pode-se obter a seguinte tabela, supondo que é f a função calculada e que os números da tabela são as computações de M':

	1	2	3	4	...
f(0)	1	3	6	10	...
f(1)	2	5	9	...	
f(2)	4	8	...		
f(3)	7	...			
⋮					

Se M pára para algum destes dados, por exemplo n, então M' escreve n na fita (seguido de um b ("branco")) e continua a simulação. Isto garante que M' só enumera elementos em A. Por outro lado,d se $n \in A$, então M com dados n pára ao fim de um número finito de computações, logo n será enumerado por M'. □

Proposição 1.7.4 *Se $A, B \subseteq \mathbb{N}$ recursivos (r.e.) então $A \cup B$ e $A \cap B$ são recursivos (r.e.)*

Teorema 1.7.13 Teorema de Post
Um conjunto $A \subseteq \mathbb{N}$ é recursivo se e só se A e $\overline{A}$ são recursivamente enumeráveis.

Dem. ($\Rightarrow$) Se A é recursiva então podemos modificar a máquina de Turing M que calcula ch_A para calcular s_A. Analogamente podemos modificar M para calcular $s_{\overline{A}}$. Logo, A e $\overline{A}$ são r.e..

($\Leftarrow$) Suponhamos que A e $\overline{A}$ são r.e., e que M e $\overline{M}$ são as máquinas de Turing que calculam respectivamente s_A e $s_{\overline{A}}$. Podemos construir uma máquina M' que calcula ch_A. A máquina M' simula em paralelo M e $\overline{M}$. Com dados n, M' executa alternadamente um passo de cada uma até uma delas parar. Se M parar o resultado de M' é 1, e se $\overline{M}$ parar o resultado de M' é 0. É garantido que para cada n uma e uma só das máquinas M e $\overline{M}$ pára (porquê?). □

[5] A tradução literal de "dove tail" é "cauda de andorinha"

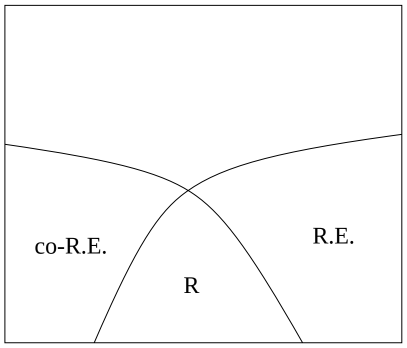


Figura 1.1: Classes de Conjuntos

Aplicação: $\overline{H}$ e $\overline{K}$ são não r.e.

Um conjunto A designa-se por $co-r.e$ se não for recursivo e $\overline{A}$ for r.e (isto é, A é o complementar dum conjunto r.e.).
Temos então que dado $A \subseteq \mathbb{N}$ ou:

(i) A e $\overline{A}$ são recursivos

(ii) Nem A nem $\overline{A}$ são r.e.

(iii) A é r.e e não recursivo e $\overline{A}$ é não r.e. (é co-r.e.)

(iv) $\overline{A}$ é r.e e não recursivo e A é não r.e. (é co-r.e.)

Na figura 1.1 apresenta-se um diagrama dos subconjuntos de $\mathbb{N}$, para esta classificação (equivalentemente das classes de linguagens).

Teorema 1.7.14 *O contra-domínio duma função total recursiva estritamente crescente é recursivo.*

Dem. Seja $f : \mathbb{N} \longrightarrow \mathbb{N}$ como enunciado e $A = CD_f$. Defina-se

$$h(n) \quad = \quad \mu z.f(z) \geq n$$

A função h é recursiva e é total porque f é total e estritamente crescente. E $n \in A$ sse $f(h(n)) = n$ o que é um predicado recursivo. □

Corolário 1.7.2 *Se A é r.e e pode ser enumerado por uma função estritamente crescente então A é recursivo.*

O teorema seguinte é usado em demonstrações de indecidibilidade e afirma apenas que uma função computável $f(a, b)$, para cada a pode ser vista como uma função computável de b, $f_a(b)$:

Teorema 1.7.15 Teorema $S - m - n$
Sejam m, $n \geq 1$. Existe uma função primitiva recursiva S_m^m tal que para qualquer índice e, $\mathbf{x} \in \mathbb{N}^m$ e $\mathbf{y} \in \mathbb{N}^n$,

$$\{e\}_{m+n}(\mathbf{x}, \mathbf{y}) \quad = \quad \{S_n^m(e, \mathbf{x})\}(\mathbf{y})$$

Dem. Seja P uma máquina que calcula $\{e\}$ dados $\mathbf{x}$ e $\mathbf{y}$. Dado $\mathbf{x}$ podemos modificar P para uma máquina Q tal que Q com dados $\mathbf{y}$ produza $\{e\}(\mathbf{x}, \mathbf{y})$. Em particular, Q dado $\mathbf{y}$ pode gerar $\mathbf{x}$ e simular P com dados $\mathbf{x}$ e $\mathbf{y}$. Obviamente Q depende de $\mathbf{x}$. □

1.7.10 Teorema de Rice e mais problemas indecidíveis

Seja C_1 o conjunto das funções parciais computáveis unárias. O teorema seguinte afirma que problemas à cerca de propriedades não triviais de funções de C_1 (genéricamente, acerca do funcionamento de máquinas de Turing ou programas em geral) são indecidíveis. Uma propriedade pode ser definida como o conjunto dos elementos que a verificam. Exemplos de propriedades em C_1 são: se duas funções (parciais) são iguais; as funções que estão definidas (convergem) para 0; etc.

Teorema 1.7.16 Teorema de Rice
Seja $B \subseteq C_1$ e $B \neq \emptyset, C_1$. Então, o conjunto $\{n \in \mathbb{N} \mid \{n\} \in B\}$ não é recursivo.

Dem. Pelo enunciado $B \neq \emptyset$ e $\overline{B} \neq \emptyset$. A função totalmente indefinida $\uparrow$ ou está em B ou em $\overline{B}$. Seja h uma função que não está no mesmo conjunto que $\uparrow$. Sem falta de generalidade suponhamos que $\uparrow \notin B$. Defina-se

$$f(x,y) = \begin{cases} h(y) & \{x\}(x)\downarrow \\ \uparrow & \text{caso contrário} \end{cases}$$

A função f é computável (verifique!). Pelo Teorema $S-n-m$, existe uma função primitiva recursiva g tal que $f(x,y) = \{g(x)\}(y)$. Vem que

$$\begin{aligned} \forall y\ \{g(x)\}(y) = h(y) \quad & se \quad \{x\}(x)\downarrow \\ \forall y\ \{g(x)\}(y)\uparrow \quad & se \quad \{x\}(x)\uparrow \end{aligned}$$

então

$$\{x\}(x)\downarrow \quad sse \quad \{g(x)\} \in B$$

Isto é reduzimos o problema da auto-paragem ao problema de saber se uma dada função está em B. Como o primeiro é indecidível o problema $\boxed{\{x\} \in B}$ é indecidível.

□

Exemplo 4 *Os seguintes problemas são indecidíveis:*

(i) $\boxed{\{n\}(0)\downarrow}$

(ii) $\boxed{\{n\}\ \textit{é total}}$

(iii) $\boxed{\{n\} = \uparrow}$

(iv) $\boxed{\{n\} = 0}$

(v) $\boxed{\{n\} = \{m\}}$, *isto é, $\{n\}$ e $\{m\}$ têm o mesmo domínio e para cada valor do domínio são iguais.*

Serão semi-decidíveis ? O problema (i) é semi-decidível. Os restantes problemas não são semi-decidíveis. O problema (ii) pode ser reduzido ao problema complementar da auto-paragem. Seja

$$f(x,y) = \begin{cases} 1 & \text{se } \{x\}(x) \text{ não pára em menos de } y \text{ passos} \\ \uparrow & \text{caso contrário} \end{cases}$$

A função f é computável. Pelo Teorema $S-m-n$ existe $g \in PR$ tal que $f(x,y) = \{g(x)\}(y)$. Se $\{x\}(x)\uparrow$ então $\{g(x)\}$ é total e se $\{x\}(x)\downarrow$ então $\{g(x)\}$ não está definida a partir dum dado valor. Isto é

$$\{g(x)\} \text{ é total} \quad sse \quad \{x\}(x)\uparrow$$

Então se $\boxed{\{n\} \text{ é total}}$ é semi-decidível também o é o problema complementar da auto-paragem. Para o problema (iii) a justificação é semelhante, usando-se para um dado n a função:

$$f(n,m) = \begin{cases} \uparrow \text{ se } \{n\}(n)\uparrow \\ 0 \text{ no caso contrário} \end{cases}$$

Aplicando o Teorema $S-m-n$ existe $g \in PR$ tal que $f(n,m) = \{g(n)\}(m)$. Note que $\{g(n)\} =\uparrow$ se e só se $\{n\}(n)\uparrow$. Para o problema (iv), a justificação é semelhante à do problema (ii), considerando:

$$f(n,t) = \begin{cases} \uparrow \text{se } \{n\}(n) \text{ dá o resultado 0 em} \leq t \text{ passos} \\ 0 \text{ no caso contrário} \end{cases}$$

Para o problema (v), tome-se $\{m\}$ a função $0(k)$ e aplique-se o resultado do problema (iv).

Serão complementares de semi-decidíveis? Os problemas (ii), (iv) e (v) não, mas (iii) é. Para a justificação do problema (ii), isto é, que o problema $\boxed{\{n\} \text{ não é total}}$ não é semi-decidível, pode-se usar a função (computável):

$$f(n,m) = \begin{cases} \uparrow & \text{se } \{n\}(n)\uparrow \\ 1 & \text{se } \{n\}(n)\downarrow \end{cases}$$

e aplicar o Teorema $S-n-m$: $f(n,m) = \{g(n)\}(m)$ e $\{g(n)\}$ é não total sse $\{n\}(n)\uparrow$. Para o problema $\boxed{\{n\} \neq\uparrow}$ é fácil provar que é r.e., usando a técnica de "dovetailing".

1.7.11 Redução $\leq_m$

Definição 1.7.3 *Dados $A, B \subseteq \mathbb{N}$, A é redutível a B (muitos para um), $A \leq_m B$, se e só se existe uma função total recursiva f tal que:*

$$n \in A \quad \leftrightarrow \quad f(n) \in B$$

Pode-se dizer que se $A \leq_m B$, então resolver A não é mais "difícil" que resolver B. Em particular:

Proposição 1.7.5 *Dados $A, B \subseteq \mathbb{N}$, $A \leq_m B$ e B recursivo então A é recursivo.*

Por exemplo, prova-se que $K \leq_m H$ considerando $f(n) = p(n,n)$. Então, K não é mais "difícil" que H, e que se K não é recursivo então H também não o pode ser[6].
Se $A \leq_m B$ e $B \leq_m A$ diz-se que $A \equiv_m B$.

Proposição 1.7.6 *A relação $\leq_m$ é reflexiva e transitiva.*

Proposição 1.7.7 *Dados $A, B \subseteq \mathbb{N}$, se A recursivo e $B \neq \emptyset, \mathbb{N}$ então $A \leq_m B$.*

Corolário 1.7.3 *Dados $A, B \subseteq \mathbb{N}$ recursivos então $A \equiv_m B$.*

Definição 1.7.4 *Seja A um conjunto r.e., A é* r.e. completo *se e só se para todo B r.e $B \leq_m A$.*

[6] Temos assim uma demonstração que H não é recursivo, sabendo que K o não é.

Isto é, os conjuntos r.e completos, são os mais "difíceis" da classe (Porquê?).

Teorema 1.7.17 *K é r.e. completo.*

Dem. A demonstração é semelhante à do teorema de Rice. Seja A um conjunto r.e. Então $x \in A$ sse $\exists n \{n\}(x) \downarrow$. Defina-se

$$f(x,y) = \begin{cases} 1 & \text{se } \{n\}(x) \downarrow \\ \uparrow & \text{se } \{n\}(x) \uparrow \end{cases}$$

A função f é computável e pelo Teorema $S-n-m$, $f(x,y) = \{g(x)\}(y)$ e $g \in PR$. Se $\{n\}(x) \downarrow$ então $\forall y\ \{g(x)\}(y) = 1$ e se $\{n\}(x) \uparrow$ então $\forall y\ \{g(x)\}(y) \uparrow$. Então $x \in A$ sse $\{g(x)\}(g(x)) \downarrow$ sse $g(x) \in K$. $\square$

1.8 Exercícios de Revisão sobre Decidibilidade e Recursividade

Bibliografia [Phi92], [LP81, Cap. 6], [HU79, Cap. 8], [Pap94, Cap. 3], [Yas71, Cap. 1, 2, 5, 7]

Exercício 1.8.1 Supõe que numa determinada linguagem de programação consegues escrever uma função booleana (f é também uma função booleana)

$$bool\ termina(f, x)$$

que termina sempre, dando o valor TRUE se a computação $f(x)$ termina e FALSE caso contrário. Considera a nova função $h(f)$ assim definida

$$h(f) \ = \ if\ termina(f, f)\ then\ \neg f(f)\ else\ TRUE$$

isto é, se f aplicado ao próprio f termina, o resultado é a negação do resultado respectivo; se não termina, o resultado é TRUE.

a) Mostra que $h(f)$ termina para todo a função f.

b) Mostra que a computação de $h(h)$ leva a uma contradição inescapável.

c) O que está errado? Qual a hipótese falsa? Enuncia como teorema a conclusão obtida.

Exercício 1.8.2 Numa determinada linguagem de programação podem definir-se funções cujos argumentos podem ser do tipo função, inteiro ou booleano. Um exemplo

$$\begin{aligned} mais(x, y) \quad &= \quad \texttt{if}\ y = 0\ \texttt{then}\ x \\ &\quad\quad \texttt{else}\ suc(mais(x, ant(y))) \end{aligned}$$

Mostra (directamente, sem invocar a indecibilidade do problema do paragem) que é impossível programar uma função total

$$termina(f, x)$$

que tem o valor lógico *Verdade* se a aplicação da função f (unária) ao argumento x é uma computação que termina e *Falso* no caso contrário. <u>Sugestão</u>: Considera a função

$$\begin{aligned} pescada(f) \quad &= \quad \texttt{if}\ termina(f, f)\ \texttt{then}\ \neg f(f) \\ &\quad\quad \texttt{else}\ Verdade \end{aligned}$$

aplicada a si mesma.

Exercício 1.8.3 Considera as seguintes características possíveis de um modelo formal de computação de funções de inteiros em inteiros (por exemplo, uma linguagem, máquina ou autómato)

(α) Todas as funções computáveis são totais.

(β) As funções aritméticas usuais como adição, multiplicação, etc, são computáveis.

(γ) é possível simular o funcionamento da "máquina" de número n com um qualquer dado m.

a) Dá um exemplo de um modelo que satisfaça as condições (α) e (β) mas não a condição (γ).

b) Dá um exemplo de um modelo que satisfaça as condições (β) e (γ) mas não a condição (α).

c) Mostra que é impossível a existência de um modelo que possua as três características mencionadas. Sugestão: Supondo que tal era possível define uma função que permita obter uma contradição.

Exercício 1.8.4 Considera as seguintes questões.

1. $\boxed{\{n\}(n) = 0}$? (para n inteiro)
2. $\boxed{\{n\}(m) = 0}$? (para n e m inteiros)
3. $\boxed{\{n\}(m) = x}$? (para n, m e x inteiros)
4. $\boxed{\{n\}(n) \downarrow}$? (para n inteiro)
5. $\boxed{\{n\}(n) \uparrow}$? (para n inteiro)
6. $\boxed{\{n\}(n+1) \downarrow}$? (para n inteiro)
7. $\boxed{\{n\}(m) \downarrow \text{ para todo o } m \leq n}$? (para n e m inteiros)

a) Descreve os correspondentes subconjuntos de $\mathbf{N}$.

b) Considerando a correspondência entre funções parciais e máquinas de Turing, descreve os problemas em termos de máquinas de Turing.

c) Diz se cada conjunto é recursivo, recursivamente enumerável e se o seu complementar é recursivamente enumerável.

Exercício 1.8.5 Mostra cuidadosamente que o problema $\boxed{\{n\}(m) \downarrow \text{em} \leq t \text{ passos}}$ é decidível.

Exercício 1.8.6 Mostra que um conjunto é recursivamente enumerável sse for o domínio de uma função recursiva (em geral parcial).

Exercício 1.8.7 Mostra que um conjunto $A \subseteq \mathbf{N}$ não vazio é recursivamente enumerável sse puder ser enumerado como o contra-domínio de função recursiva total.

Exercício 1.8.8 Averigua se existem máquinas de Turing específicas M (de número m) tais que:

1. O problema $\boxed{\{m\}(n) \downarrow}$ (isto é "a máquina converge para o argumento n?" — Note que m é fixo e n é o parâmetro do problema) é decidível.
2. O problema $\boxed{\{m\}(n) \downarrow}$ é indecidível (mas semi-decidível).
3. O problema $\boxed{\{m\}(2) \downarrow}$ é indecidível.

Exercício 1.8.9 Se A corresponde à propriedade P_A, isto é

$$A = \{x \mid x \in \mathbf{N} \land P_A(x)\}$$

e se B corresponde à propriedade P_B, a que propriedades correspondem $A \cup B$, $A \cap B$ e $\overline{A}$?

1. Que podes dizer de $A \cup B$ e de $A \cap B$ se A e B forem recursivos?

2. Que podes dizer de $A \cup B$ e de $A \cap B$ se A e B forem recursivamente enumeráveis?

3. Que podes dizer de $A \cup B$ e de $A \cap B$ se A for recursivo e e B for recursivamente enumerável?

4. Que podes dizer de $\overline{A}$ se A for recursivo?

5. Que podes dizer de $\overline{A}$ se A for recursivamente enumerável?

Exercício 1.8.10 Dá um exemplo de uma propriedade P tal que nem P nem $\neg P$ sejam semi-decidíveis. Sugestão: Considera propriedades relativas ao problema da paragem para *duas* máquinas de Turing.

Exercício 1.8.11 Usando o Teorema de Rice estuda a decidibilidade dos problemas seguintes. Para cada caso de indecibilidade averigua se o problema é semi-decidível.

1. $\boxed{\{x\}(0)\downarrow}$? (equivale ao problema da paragem para uma TM que comece com a fita vazia)

2. $\boxed{\{x\} \text{ é total}}$?

3. $\boxed{\{x\} =\uparrow}$ (função não definida em nenhum ponto) ?

4. $\boxed{\{x\} = 0}$ (função total 0(x))?

5. $\boxed{\{x\} = \{y\}}$?

Exercício 1.8.12 Provou-se no curso (e no exercício anterior ...) que o problema de se saber se uma função é total não é decidível nem sequer semi-decidível. Mostre que também não é

1. O complementar de um problema decidível.

2. O complementar de um problema semi-decidível, isto é, que a questão "f não é total" não é semi-decidível. Sugestão: considera a seguinte função

$$f(n,m) = \begin{cases} \uparrow & \text{se } \{n\}(n)\uparrow \\ 1 & \text{se } \{n\}(n)\downarrow \end{cases}$$

Exercício 1.8.13 Enuncia o Teorema de Rice em termos do conjunto de todas as linguagens recursivamente numeráveis.

Exercício 1.8.14 Dizemos que $P \leq_T Q$ (redução de Turing) quando existe uma máquina de Turing que decide P usando uma rotina (oráculo) para decidir Q. Nota, por exemplo, que a afirmação $\{x\}(x)\uparrow \leq_T \{x\}(x)\downarrow$ é verdadeira enquanto a afirmação $\{x\}(x)\uparrow \leq_m \{x\}(x)\downarrow$ é falsa.

1. Mostra que se Q é recursivo (r.e.) e se $P \leq_m Q$ então Q também é recursivo (r.e.).

2. Mostra que $P \leq_m Q$ implica $P \leq_T Q$ (o contrário é falso).

3. Mostra que, na classe dos problemas, as relações $\leq_m$ e $\leq_T$ são relações binárias de pré-ordem, isto é, que são reflexivas e transitivas mas que não são relações de ordem.

4. Seja D um problema decidível, S um problema não decidível mas semi-decidível, C um problema não decidível mas que é o complementar de um problema semi-decidível e N um problema que não pertence a nenhuma das classes mencionadas.

   Para todos os casos $X, Y \in \{D, S, C, N\}$ e $r \in \{\leq_T, \leq_m\}$ quais das afirmações XrY são verdadeiras e quais as falsas?

Exercício 1.8.15 [$\star$]**Sistemas de Reescrita (Semi-Thue)**
Um sistema de reescrita num alfabeto finito é um conjunto de pares de palavras a que se chama regras de reescrita

$$S = \{(x_1, y_1), \cdots, (x_n, y_n)\}$$

onde os x_i e os y_i são membros deΣ^*. é usual escrever uma regra de reescrita (x, y) na forma $x \to y$.

Diz-se que a palavra w se pode reescrever directamente na palavra w' se, para algum i, w for da forma tx_iz e w' for da forma ty_iz; por outras palavras, se pudermos aplicar uma regra de reescrita a uma qualquer sub-palavra de w. Diz-se que w se pode reescrever na palavra w' e escreve-se

$$w \to^* w'$$

se, por uma aplicação sucessiva de $n \geq 0$ regras de reescrita, w se puder transformar em w'.

1. Mostra que a relação $\to^*$ definida em Σ^* é reflexiva e transitiva.
2. Dê um exemplo de um sistemas de reescrita um em que a relação $\to^*$ seja de ordem e outro em que seja apenas de pré-ordem (não antissimétrica).
3. Considera o seguinte problema RE: dado um sistema de reescrita e duas palavras w e z, é $w \to^* z$?
   (a) Mostra que RE é semi-decidível.
   (b) Mostra que RE não é decidível. <u>Sugestão</u>: reduz o problema de paragem de uma máquina de Turing com fita vazia no início e no fim (se houver) da computação a este problema. Admitindo, por exemplo, que o alfabeto da fita é $\{a, b\}$, faz corresponder a w a configuração inicial da computação, $\#s\#b\#$ (onde s codifica o estado inicial no alfabeto $\{a, b\}$ e b é o símbolo debaixo da cabeça) e a z a configuração final, $\#f\#b\#$ (onde f codifica o estado final que, sem perda de generalidade, se pode supor único). Além disso, para cada regra da máquina de Turing, por exemplo,

   $$(s, a) \to (s', b, +1)$$

   corresponde uma ou mais regras de reescrita, neste caso, se admitirmos, por exemplo que, abb codifica s e bbb codifica s' ($\#$ é um símbolo auxiliar) teríamos as seguintes regras de reescrita correspondentes ao facto do símbolo à direita da cabeça ser respectivamente a ou b

   $$\#\mathtt{abb}\#\mathtt{a}\#\mathtt{a} \to \mathtt{b}\#\mathtt{bbb}\#\mathtt{a}\#$$

   e

   $$\#\mathtt{abb}\#\mathtt{a}\#\mathtt{b} \to \mathtt{b}\#\mathtt{bbb}\#\mathtt{b}\#$$

Exercício 1.8.16 [$\star\star$] **Problema da correspondência de Post**
Considera o seguinte problema (PCP): é dado um conjunto finito de pares de palavras (todas escritas com símbolos de um mesmo alfabeto finito Σ)

$$S = \{(x_1, y_1), \cdots, (x_n, y_n)\}$$

Pergunta-se se existe uma sequência não nula de x_is (com possível repetição) que iguala a correspondente sequência de ys. Por outras palavras, pergunta-se se existe uma sequência de inteiros $i_1, \ldots, i_n$, $(n \geq 1)$ todos $\leq n$, tais que

$$x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n}$$

1. Verifica se existe solução para os seguintes PCP (de [Man74, Cap. 1.5]); Em ambos os casos é $\Sigma = \{a, b\}$:
   (a) $\{(b, bbb), (babbb, ba), (ba, a)\}$
   (b)) $\{(ab, abb), (b, ba), (b, bb)\}$
   Sugestão: considera o número de bs dos x e dos ys.
2. Mostra que o problema PCP é semi-decidível; utiliza uma linguagem de alto nível para descrever um algoritmo de semi-decidibilidade e invoca a tese de Church-Turing.
3. Mostra que o problema PCP é indecidível. Ver uma solução, por exemplo, em [HU79, Cap 8] ou [LP81, Cap 6].

Exercício 1.8.17 [$\star\star$] **A hierarquia aritmética** (ver [Pap94], Cap. 8. [FB94])
A relação $R \subseteq (\Sigma^*)^k$ diz-se recursiva se a linguagem

$$L = \{x_1; x_2; \cdots; x_k \mid (x_1, x_2, \cdots, x_k) \in R\}$$

for recursiva. Defina-se Σ_k $(k \geq 0)$ como sendo a classe de todas as linguagens L para as quais existe uma relação R recursiva de aridade $k+1$ tal que

$$L = \{x \mid \exists x_1 \forall x_2 \cdots Q_k x_k \; (x_1, x_2, \cdots, x_k, x) \in R\}$$

onde os quantificadores alternam, isto é, Q_k é $\exists$ se k é ímpar e $\forall$ se k é par. Defina-se ainda $\Pi_k = \mathtt{co}\Sigma_k$. À reunião de todos os Σ_i para $i \geq 0$ chama-se hierarquia aritmética.

a) Mostre que para todo o $k \geq 0$, Π_k é a linguagem L tal que existe uma relação recursiva R tal que

$$L = \{x \mid \forall x_1 \exists x_2 \cdots Q_k x_k \; (x_1, x_2, \cdots, x_k, x) \in R\}$$

b) Mostre que $\Sigma_0 = \Pi_0 = \mathbf{R}$ onde $\mathbf{R}$ é o conjunto das linguagens recursivas.

c) Mostre que, para todo o $i \geq 0$ é $\Sigma_i \subseteq \Sigma_{i+1}$ e $\Pi_i \subseteq \Sigma_{i+1}$

d) Mostre que Σ_2 é a classe das linguagens que podem ser aceites (não decididas) por máquinas de Turing onde existe a seguinte possibilidade de consulta ao "oráculo" da paragem:
Se a máquina estiver no estado $q?$ o próximo estado vai ser q_{sim} ou $q_{\text{não}}$ conforme, respectivamente, o conteúdo x da fita (desde a cabeça até ao próximo branco) for ou não a codificação de uma máquina de Turing para a qual a resposta ao problema da auto-paragem seja sim.

e) Estenda o resultado da alínea anterior para os casos $i > 2$.

f) Mostre que, para todo o i é $\Sigma_{i+1} \neq \Sigma_i$, generalizando o caso $i = 0$.

g) Mostre que $\{n \mid L_{\{n\}} = \Sigma^*\}$, o conjunto de todos os números de máquinas que aceitam todas as sequências, pertence a $\Pi_2 - \Pi_1$.
Sugestão: Mostre que está incluído em Π_2 usando a alínea (d); mostre que não pertence a Π_1 verificando que todas as linguagens de Π_2 se reduzem a ele e utilize o resultado da alínea e).

h) Posicione na hierarquia aritmética, tão rigorosamente quanto possível, a linguagem

$$\{M \mid L_M \text{ é infinita}\}$$

1.9 Sugestões para a Resolução de Exercícios da Secção 1.8

Exercício 1.8.4

Utilizamos a correspondência entre funções parciais e máquinas de Turing e a bijecção $p(x,y)$ de $\mathbf{N}^2$ em $\mathbf{N}$ caracterizada nas aulas.

1. $\boxed{\{n\}(n)=0}$: Conjunto dos inteiros n tais que a função parcial $\{n\}$ converge e vale 0 para o argumento n.

   Conjunto não recursivo: se fosse recursivo existia uma função algoritmica (total) f tal que $f(n)=1$ se $\{n\}(n)=0$ e $f(n)=0$ se $\{n\}(n)$ não convergisse ou fosse não nulo. Seja k o índice de uma máquina de Turing que calcula esta função. Considerando o valor de $f(k)$ (isto é $\{k\}(k)$) chegamos a uma contradição: se $f(k)=1$ deveria ser $f(k)=0$ (por definição de f e de k) e vice-versa.

   O conjunto é recursivamente enumerável: considere-se a simulação do cálculo de $\{n\}(n)$; se houver convergência, "provoca-se" a saída 1.

2. $\boxed{\{n\}(m)=0}$: Conjunto dos inteiros da forma $p(m,n)$ tais que a função parcial $\{n\}$ converge e vale 0 para o argumento m.

   Conjunto não recursivo. Suponhamos que é recursivo e seja f a correspondente função; então o conjunto da alínea (a) também era recursivo. Considere-se com efeito a função total

   $$n \longrightarrow p(n,n) \longrightarrow f(p(n,n))$$

3. $\boxed{\{n\}(m)=x}$:

   Resposta idêntica à do exercício anterior, considerando o emparelhamento $p(p(n,m),x)$.

4. $\boxed{\{n\}(n)\downarrow}$:

   Linguagem correspondente ao problema (restrito) do "halting". Recursivamente enumerável, não recursivo.

5. $\boxed{\{n\}(n)\uparrow}$: Complementar do conjunto anterior. Pelo teorema de Post nem é recursivo nem recursivamente enumerável (mas o seu complementar é recursivamente enumerável).

6. $\boxed{\{n\}(n+1)\downarrow}$? (para n inteiro) É fácil ver-se que o conjunto respectivo é recursivamente enumerável considerando uma máquina de Turing que, a partir do dado n, simula a máquina $\{n\}$ com a entrada $n+1$ e que, se houver convergência, dá o resultado 1.

   Pode provar-se que o conjunto não é recursivo por absurdo. Suponhamos que era recursivo, sendo f a função correspondente, implementada por uma máquina de índice k. Construimos uma outra máquina que faz o seguinte:

   (a) Com entrada x aplica $x-1$ a M_k, isto é, calcula $f(x)$.
   (b) Se o resultado for 1 entra em ciclo infinito.
   (c) Se for 0 pára e dá resultado 0.

   Seja p o índice desta máquina. A contradição resulta da análise da aplicação de $p+1$ a esta máquina. Se houver convergência, M_k deve dar 1 mas então haverá divergência (por construção de M_p) e vice-versa.

   O método utilizado nesta demonstração é válido para qualquer dado que substitua $p+1$, como, por exemplo, 1, 2 ou p^2.

7. $\boxed{\{n\}(m)\downarrow \text{ para todo o } m \leq n}$? (para n e m inteiros) Conjunto recursivamente enumerável (usar a simulação "paralela" para $m = 1, \cdots, n$) mas não recursivo (o caso $m = n$ é o problema do "halting"). Os detalhes ficam para o leitor.

Exercício 1.8.8

1. Seja M de índice m uma máquina de Turing correspondente a uma função constante, p.e. 1. Para qualquer dados M apaga os dados e escreve 1 na fita. Então, $\boxed{\{m\}(n)\downarrow}$ é decídivel. Nota que para uma máquina qualquer fixa, o problema é indecidível.
2. Seja M de índice m a máquina de Turing que aceita o problema da auto-paragem.
3. Dada qualquer máquina de Turing M de índice m, o problema $\boxed{\{m\}(2)\downarrow}$, para m fixo, é decidível, pois só tem uma instância. Então, é resolvido por uma TM que termine com 1 para todos os dados ou uma TM que termine com 0 para todos os dados. Nota que, qualquer conjunto finito é recursivo.

Exercício 1.8.9

É fácil ver-se que $A \cup B$, $A \cap B$ e $\overline{A}$ correspondem respectivamente a $P_A \vee P_B$, $P_A \wedge P_B$ e $\neg P_A$.

1. São conjuntos recursivos. Por exemplo, para mostrar que $A \cup B$ é recursivo, sejam f_A e f_B as funções (totais) correspondentes respectivamente à recursividade de A e de B e consideremos uma máquina de Turing M que a partir de um dado x
   (a) "Calcula" $f_A(x)$.
   (b) Se $f_A(x) = 1$ (x pertence a A), M dá resultado 1 e pára.
   (c) Se $f_A(x) = 0$ (x não pertence a A), M "calcula" $f_B(x)$ e dá como resultado o valor deste cálculo (se x não pertence a A então pertence a $A \cup B$ sse pertence a B.
2. São recursivamente enumeráveis. A justificação pode basear-se numa máquina de Turing que simula em passos alternados A e B.
3. São conjuntos recursivamente enumeráveis. A prova pode basear-se numa máquina de Turing que começa por decidir se x pertence a A.
4. É recursivo. Utilizar ou o teorema de Post ou uma função que dá 0 se $f_A(x) = 1$ e que dá 1 se $f_A(x) = 0$.
5. O conjunto $\overline{A}$ pode ser ou não recursivamente enumerável.

Exercício 1.8.10

Considere-se um inteiro x e sejam k e m tais que $p(k, m) = x$. Consideremos a propriedade $P(x)$

$$\boxed{(\{k\}(k)\downarrow) \ \wedge \ (\{m\}(m)\uparrow)}\ ?$$

Sejam c e d índices de máquinas de Turing (fixas) tais que $\{c\}(c)\downarrow$ e $\{d\}(d)\uparrow$ (por exemplo, c pode corresponder a uma máquina que pára imediatamente para qualquer dado e d a uma máquina que se mantém sempre no estado inicial). Então se P fosse semi-decidível poderíamos "semi-decidir" o problema do "halting" para uma qualquer máquina m considerando $P(x)$ com $k = c$. E analogamente, como $\neg P$ é

$$\boxed{(\{k\}(k)\uparrow) \ \vee \ (\{m\}(m)\downarrow)}\ ?$$

se considerarmos $m = d$ (de modo que o segundo termo da disjunção é falso) vemos que $\neg P$ não pode ser semi-decidível.

Exercício 1.8.11

Seja C_1 a classe das funções parciais de aridade 1. Excepto para a primeira alínea apenas damos uma ideia da justificação.

1. $\boxed{\{x\}(0)\downarrow}$? Existem funções recursivas que convergem para o argumento 0 (por exemplo, $0(x)$) e existem funções que não convergem para o argumento 0 (por exemplo, a função $\uparrow(x)$, não definida em nenhum valor). Aplicando o teorma de Rice concluí-se que o problema é indecidível. Por outro lado, para saber se $\{x\}(0)\downarrow$ basta fazer uma simulação da máquina $\{x\}$ com 0 inicialmente na fita. Se houver convergência, a saída é 1. Portanto, o problema é semi-decidível.

2. $\boxed{\{x\}\text{ é total}}$?

   Seja B a classe das funções totais e B' a classe das funções não totais (que divergem pelo menos num ponto). Nem B nem B' são o conjunto vazio – é fácil encontrar funções totais (por exemplo $f(x) = 0$) e não totais (por exemplo $f(x) =\uparrow$) e, por outro lado é $C_1 = B \cup B'$. Logo aplica-se o teorema de Rice e B não é recursivo.

   Vamos mostrar que a semi-decidibilidade da questão considerada teria como consequência a semi-decidibilidade do problema restrito do "halting". Consideremos a seguinte função de dois argumentos

   $$f(n,t) = \begin{cases} 1 & \text{se } \{n\}(n) \text{ não converge em } \leq t \text{ passos} \\ \uparrow & \text{no caso contrário} \end{cases}$$

   Para um valor dado de n a função $f(n,t)$ é uma função $g(t)$ de um só argumento (invocação implícita do teorema S-mn). Esta função é total se e só se $\{n\}(n)$ divergir. Então, se o problema dado for semi-decidível, podemos construir (pelo método esquematizado) uma máquina de Turing correspondente à semi-decidibilidade do problema "$\boxed{\{n\}(n)\uparrow}$?" o que, como sabemos, não é possível.

3. $\boxed{\{x\} =\uparrow}$ (função não definida em nenhum ponto) ?

   Não é decidível.

   Não é semi-decidível. Breve justificação: considerar para um dado n a função

   $$f(m) = \begin{cases} \uparrow & \text{se } \{n\}(n)\uparrow \\ 0 & \text{no caso contrário} \end{cases}$$

   Note que $f(m) =\uparrow$ se e só se $\{n\}(n)\uparrow$.

4. $\boxed{\{x\} = 0}$ (função total 0(x))?

   Não é decidível.

   Não é semi-decidível. Justificação semelhante à da alínea (a), utilizar a função

   $$f(n,t) = \begin{cases} \uparrow & \text{se } \{n\}(n) \text{ dá o resultado 0 em } \leq t \text{ passos} \\ 0 & \text{no caso contrário} \end{cases}$$

5. $\boxed{\{x\} = \{y\}}$?

Não decidível; tome-se um valor particular de y – por exemplo correspondente à função $0(x)$ – e aplique-se o teorema de Rice.

Não é semi-decidível. Tome-se o valor sugerido para y e aplique-se o resultado da alínea anterior.

Exercício 1.8.15 Ver por exemplo [FB94][Cap. 7.10].
Exercício 1.8.16 Ver por exemplo [FB94][Cap. 7.11] ou [RS93].
Exercício 1.8.17 Ver por exemplo [FB94][Cap. 8.5].

Capítulo 2

Teoria dos problemas NP-completos

Bibliografia [GJ79, Cap. 1–3,5,7.1,7.2], [Pap94, Cap. 8, 9, 10 e 17] e [BC94, Cap. 4, 5 e 7]

2.1 Classes de complexidade

O conjunto das linguagens decidíveis pode ser dividida em classes de complexidade, que caracterizam os limites dos recursos computacionais usados para as decidir. Uma *classe de complexidade* é especificada pelo modelo de computação – vamos considerar máquinas de Turing com k-fitas –, pelo modo de computação – determinístico ou não determinístico –, pelo um recurso – p.e., tempo ou espaço – e por um limite, isto é, uma função de $\mathbb{N}$ em $\mathbb{N}$. Uma classe de complexidade é então um conjunto $C(f(n))$ definido por:

$$C(f(n)) = \{L \mid L \text{ é decidida por uma máquina de Turing } M \text{ do modo adequado, tal que para qualquer } x,\text{ e } |x| = n,\ M \text{ gasta no máximo } f(|x|) \text{ unidades do respectivo recurso}\}$$

Uma função $f(n)$ para ser limite de complexidade tem de ser total e não decrescente. No caso, de funções de complexidade temporal deve ainda ser tal que exista uma TM M_f de k-fitas que, para quaisquer dados x, pare em exactamente $f(|x|)$ passos. Esta característica permite a simulação de "relógios" em máquinas de Turing. Restrições semelhantes se impõem para funções de complexidade de espaço.

2.1.1 Classe P

Seja $M = (S, \gamma, \Sigma, s_0, \delta, F)$ uma máquina de Turing (determinística) e $x \in \Sigma^\star$, o tempo requerido por M em x, $T_M(x)$, é o número de passos da computação de M para dados x.[1] Então, a *complexidade temporal* de M é dada por:

$$T_M(n) = max\{t \mid \exists x \in \Sigma^\star,\ |x| = n \text{ e } t = T_M(x)\}$$

Se $T_M(n)$ é $O(f(n))$, diz-se que M opera em tempo limitado por $f(n)$. Se $f(n)$ é um polinómio dizemos que M é polinomial.

[1] Se a computação não parar $T_M(x) = \infty$. Nesta secção vamos considerar apenas máquinas de Turing que param sempre.

Recorde que, se uma máquina de Turing M decidir uma linguagem L, então $F = \{s_y, s_n\}$. Dada uma função $f(n)$, define-se a classe de complexidade $TIME(f(n)$ por:

$$TIME(f(n)) \quad = \quad \{L \mid L \text{ é decidível por uma TM } M \text{ tal que } T_M(n) \text{ é } O(f(n))\}$$

A classe **P** é definida como o conjunto de linguagens que são decidíveis em tempo polinomial por uma TM, isto é:

$$\mathbf{P} \quad = \quad \bigcup_{k \geq 1} TIME(n^k)$$

2.1.2 Classe NP

Seja $N = (S, \gamma, \Sigma, s_0, \Delta, F)$ uma máquina de Turing não- determinística (NDTM), onde

$$\Delta \subseteq (S \times \Gamma) \times (S \times \Gamma \times \{l, r\})$$

Recorde que um passo de computação entre duas configurações é definido por uma relação $xsy \underset{M}{\longrightarrow} x's'y'$ se e só se existe $((s, \alpha), s', \alpha', d) \in \Delta$ que torne a transição possível. Então uma computação de N (em n passos, $\underset{N}{\longrightarrow}^n$) com dados $x \in \Sigma^\star$ pode ser vista como uma árvore de computação. Cada caminho da árvore iniciado na raiz é um *caminho de computação*. Diz-se que N aceita $x \in \Sigma^\star$, se a árvore de computação de N com dados x incluí pelo menos um caminho de computação que termina num estado de aceitação. Isto é, N aceita x se existe uma sequência de escolhas não-determinísticas que leva a um estado de aceitação. Note-se que x só é rejeitado se *todas* as sequências de escolhas (caminhos de computação) conduzirem a um estado de rejeição. Note ainda a assimetria entre a aceitação e a rejeição por estas máquinas, que não existia nas máquinas de Turing determinísticas.

Seja $L(N) = \{x \in \Sigma^\star \mid N \text{ aceita } x\}$. Se $L \subseteq \Sigma^\star$, N decide L se $L = L(N)$.

Se N aceita x, o tempo associado é definido pelo número de passos do menor caminho de computação que leva a um estado de aceitação, isto é:

$$T_N(x) \quad = \quad min\{t \mid N \text{ aceita } x \text{ com um caminho de computação em } t \text{ passos}\}$$

e, a complexidade temporal de N é a função:

$$T_N(n) \quad = \quad max\{\{1\} \cup \{m \mid \exists\, x \in L(N),\ |x| = n \text{ e } m = T_N(x)\}\}$$

Note-se, que $T_M(n) = 1$ se não há nenhum x, com comprimento n que seja aceite por N.

Se $T_N(n)$ é $O(f(n))$, diz-se que N opera em tempo limitado por $f(n)$ e se $f(n)$ é um polinómio dizemos que N é polinomial.

Seja,

$$NTIME(f(n)) \quad = \quad \{L \mid L \text{ é decidível por uma DTM } N \text{ tal que } T_N(n) \text{ é } O(f(n))\}$$

Teorema 2.1.1 *Se $L \in NTIME(f(n))$ então existe $c > 1$ tal que $L \in TIME(c^{f(n)})$.*

A classe **NP** é definida com o conjunto de linguagens que são decidíveis em tempo polinomial por uma NDTM, isto é:

$$\mathbf{NP} \quad = \quad \bigcup_{k \geq 1} NTIME(n^k)$$

Corolário 2.1.1 $\mathbf{P} \subseteq \mathbf{NP}$

2.2 Problemas, Linguagens e Complexidade

Para resolver um problema por meios computacionais é necessário codificá-lo numa dada linguagem e escrever um algoritmo (um programa, nessa mesma linguagem), isto é, um método para resolver o problema num número finito de passos. Vamos considerar que cada problema pode ser descrito por um conjunto de parâmetros e de relações entre eles, com as quais se expressam as condições de solução do problema. Uma instância `I` dum problema Π , é obtida especificando valores para cada um dos parâmetros. Chamaremos instância genérica à descrição dum problema em termos dos seus parâmetros, ou apenas instância se na enunciação do mesmo. Por exemplo, considere-se um "caixeiro viajante" que tem de visitar todas as cidades de um dado país e voltar à cidade donde partiu, e pretende fazê-lo pelo caminho mais curto. Podemos enunciar este problema clássico da combinatória, em termos dum conjunto finito de elementos e de propriedades desses elementos:

(TSO) Dado um conjunto de cidades e as distâncias entre elas existe uma permutação das cidades que minimize a distância total?

Instância: $C = \{c_1, c_2, \ldots c_n\}$ e $d : C \times C \longrightarrow \mathbb{N}$ distância

Questão: Qual é a permutação $\pi : C \longrightarrow C$ que minimiza

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)})$$

Uma instância do problema é por exemplo $C = \{Porto, Aveiro, Viana, Espinho\}$ e $d(P,A) = 60$, $d(P,V) = 60$, $d(P,E) = 20$, $d(A,V) = 120$, $d(A,E) = 40$, $d(V,E) = 80$. A sequência $< V, P, E, A >$ minimiza a distância total, $d_T = 120$.

Pretende-se normalmente obter um algoritmo que (se existir!) seja o mais eficiente. A eficiência dum algoritmo pode-se medir em:

- tempo de execução em função dos dados
- espaço gasto para a obtenção da solução do problema.

Para cada problema a complexidade temporal vai depender da máquina usada e da codificação escolhida. Contudo, cada problema pode ser classificado do ponto de vista computacional, como:

- insolúvel
- tratável
- intratável

Esta classificação é válida independentemente da máquina ou modelo computacional usado (máquina de Turing, RAM's, etc) e da codificação escolhida, desde que seja "razoável", isto é, que satisfaça as seguintes condições:

1. A codificação duma instância `I` , deve ser concisa e não conter símbolos desnecessários.
2. Os números inteiros devem ser representados numa base maior ou igual a 2.

Podemos supor que os números inteiros se representam em binário; os conjuntos pela sequência dos seus elementos (codificados) e os grafos pela sequência de vértices seguida dos pares de vértices para cada ramo (ou pela matriz de adjacência). Neste caso, se um dado é um número n o seu comprimento será $\lceil \log_2 n \rceil$, se é um conjunto de n elementos o seu comprimento é n e se é um grafo o seu comprimento é o número de vértices mais o de ramos.

Considerem-se os seguintes problemas e procuremos resolvê-los !

1. **Problema da Paragem**

   Instância: Um algoritmo e um conjunto de dados para esse algoritmo.

   Questão: Esse algoritmo pára para esse conjunto de dados (ou entra num ciclo infinito)?

   Resposta: Não existe nenhum algoritmo que resolva este problema. É um problema insolúvel.

2. **Procura dum elemento numa sequência.**

   Instância: X, $V[i]$, $1 \leq i \leq n$

   Questão: Existe i tal que $X = V[i]$?

   Resposta: Existem algoritmos polinomiais que o resolvem, por exemplo:

   ```
   k <- 1;
   V[i+1] <-  X;
   enquanto V[k] <> X faça k <-  k+1;
   se k <> n então escreva(´sim´) senão escreva(´não´)
   fim.
   ```

   A complexidade deste algoritmo é $O(n)$, considerando o número de comparações que serão necessárias no "pior caso".

3. **(HC)** Existência de ciclo hamiltoniano

   Instância: Seja $G = (V, E)$ um grafo dirigido.

   Questão: Existe um ciclo $< (X_1, X_2)(X_2, X_3) \ldots (X_{n-1}, X_n)(X_n, X_1) >$, $X_i \in V$, $(X_i, X_{i+1}) \in E$ e e $n = |V|$, tal que todos os X_i são diferentes?

   Resposta: Não se conhece nenhum algoritmo polinomial que resolva este problema. Um algoritmo que corre em tempo exponencial no número de vértices é o seguinte:

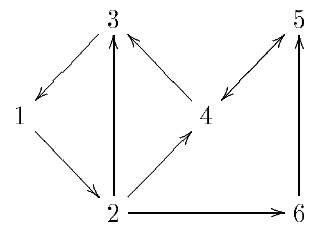
   ```
   para i <- 1 até n faça marca[i] <- falso;
   k <-  0;
   se procura(1) então escreva (´ sim´) senão escreva(´não´);
   fim.

   bool procura(i)
   {
         k <- k+1;
         marca[i]<- verdade;
         para todos (i,j) em E faça {
   ```

```
                se j=1 e k=n então retorna verdade
                se não marca[j] então se procura(j)
                                then retorna verdade
         }
         k <- k-1; marca[i]<- falso; retorna falso
     }
```

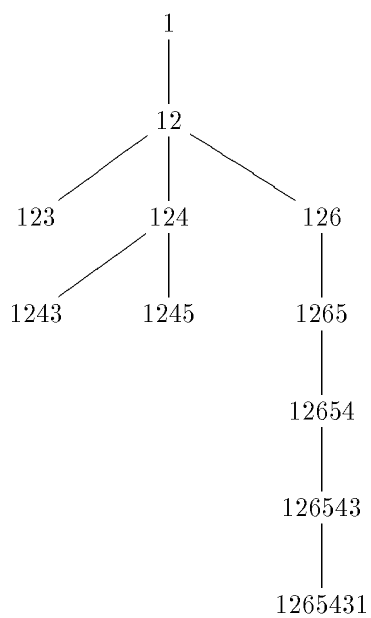
Exemplo 5 *Seja* $G = (V, E)$ *onde*

$$\begin{array}{rcl} V & = & \{1,2,3,4,5,6\} \\ E & = & \{(1,2),(2,3),(3,1),(2,4),(2,6),(4,3),(4,5),(5,4),(6,5)\} \end{array}$$

com o seguinte diagrama:



Neste caso, temos a seguinte árvore de procura, se $X_1 = 1$:



Se G é um grafo completo, o tempo de execução é $O(n)$ (caso melhor) mas no pior caso o algoritmo tem complexidade $O(n!)$.

Exercício 2.2.1 Escreva um programa para cada um dos problemas 2 e 3 e obtenha o tempo de execução para vários valores dos parâmetros. Comente os resultados.

Um problema é *tratável* se existe um algoritmo polinomial que o resolve. Se todos os algoritmos conhecidos para resolver um problema forem exponenciais o problema diz-se *intratável*. Isto porque o tempo de execução dum algoritmo exponencial torna-se rapidamente insuportável (quase "infinito") para comprimentos pequenos dos dados. O quadro seguinte ilustra a diferença entre as complexidades temporais citadas. Para valores crescentes do comprimento dos dados, n, indica o tempo de execução, tomando para base o microsegundo.

	Valores de n			
Complexidade temporal	10	20	40	60
n	10 μs	20 μs	40 μs	60 μs
n^2	0.1 ms	0.4 ms	1.6 ms	3.6 ms
n^5	0.1 s	3.2 s	1.7 minutos	13 minutos
2^n	10 ms	1 s	12.7 dias	366 séculos
10^n	2.7 horas	3.1×10^4 séculos	3.1×10^{24} séculos	

É importante notar que esta situação não se modifica significativamente se as máquinas usadas forem muito mais rápidas. Considerando aumentos de 100 e 1000 vezes na velocidade dum processador podemos ver pelo quadro seguinte, que o tamanho máximo dos problemas resolvidos num dado tempo por um algoritmo exponencial não é praticamente alterado (principalmente em comparação com o que sucede com os algoritmos polinomiais, mesmo de grau elevado).

	Máximo comprimento dos dados para 1 hora de CPU		
Complexidade temporal	Computador actual	Computador 100x rápido	Computador 1000x rápido
n	X_1	$100X_1$	$1000X_1$
n^2	X_2	$10X_2$ n	$31.6X_2$
n^5	X_3	$2.5X_3$	$3.98X_3$
2^n	X_4	$X_4 + 6.64$	$X_4 + 9.97$
10^n	X_5	$X_5 + 2$	$X_5 + 3$

Note-se contudo que um algoritmo exponencial não significa que para todas as instâncias demore um tempo que é exponencial nos dados, mas sim que no pior caso pode demorar esse tempo. Isto permite na prática usar algoritmos exponenciais para resolver alguns problemas (ex: método simplex para resolver problemas de programação linear).

Facto: Existe uma grande família de problemas para os quais só se conhecem algoritmos exponenciais - essencialmente problemas de optimização e decisão que exigem procuras exaustivas em determinados conjuntos.

A seguir apresentámos alguns problemas dos quais uns pertencem a essa classe e outros para os quais se conhecem algoritmos polinomiais para os resolver.

(EC) Dado um grafo $G = (V, E)$, G tem um ciclo euleriano, isto é, que passe por todos os ramos uma só vez?

(HC) Dado um grafo $G = (V, E)$, G tem um ciclo hamiltoniano, isto é, que passe por todos os vértices uma só vez ?

(PRIMO) Dado n inteiro, n é primo?

Dada uma família C de subconjuntos de um conjunto S, existe uma partição de S, $S = S_1 \cup S_2$, e tal que $\forall\ S' \in C,\ S' \cap S_1 \neq \emptyset$ e $S' \cap S_2 \neq \emptyset$

(DIOF) Dada uma equação de coeficientes inteiros, existem soluções inteiras?

(PAR) (Partição) Dado um conjunto A e uma função $s : A \longrightarrow \mathbb{N}$, existe $A' \subset A$ tal que

$$\sum_{a \in A'} s(a) = \sum_{a \in a \setminus A'} s(a)$$

(VC) (Cobertura por vértices) Dado um grafo $G = (V, E)$ e $K \leq\mid V \mid$ existe $V' \subseteq V$, $\mid V' \mid \leq K$ e $\forall (a, b) \in E$, $a \in V'$ ou $b \in V'$.

(LP) (Programação Linear) Dada uma matriz de inteiros A e dois vectores de inteiros b e c, existe um vector real x tal que $Ax \leq b$ e maximize cx ?

(3DM) Dado um conjunto $M \subseteq W \times Y \times Z$, onde $|W| = |Y| = |Z| = q$ e disjuntos, existe um subconjunto $M' \subseteq M$, tal que $|M'| = q$ e nenhum elemento de M' tem uma "coordenada" comum?

(SAT) Um literal é uma variável lógica ou a sua negação. Uma cláusula é uma disjunção de literais. Dado um conjunto de cláusulas (representando a sua conjunção) $C = \{c_1, c_2, ..., c_m\}$ com literais num conjunto finito U de variáveis lógicas, existe uma atribuição de valores de verdade às variáveis de tal modo que todas as cláusulas de C sejam simultaneamente satisfeitas?

(3SAT) Análogo ao anterior com todas as cláusulas com 3 literais ?

(2SAT) Análogo ao anterior com todas as cláusulas com 2 literais ?

(K-COLOR) Dado um grafo $G = (V, E)$ e um inteiro k, pode-se colorir G com k cores, isto é, existe uma função $\Phi : \mathbb{N} \longrightarrow \mathbb{Z}_k$, tal que, se $\{u, v\} \in E$, então $\Phi(u) \neq \Phi(v)$?

2.2.1 Problemas de decisão

Vamos começar por analisar um tipo particular de problemas: os problemas de decisão.

Um problema de decisão Π é um problema que apenas tem duas soluções possíveis: ou "sim" ou "não". O conjunto das suas instâncias $\mathtt{D}_\Pi$ é divido em dois subconjuntos $\mathtt{S}_\Pi$ e $\mathtt{N}_\Pi$, correspondentes respectivamente, às instâncias para as quais a resposta é "sim" e às instâncias para as quais a resposta é "não".

Estes problemas podem-se converter facilmente em problemas de reconhecimento de linguagens e, portanto, é possível estudar a sua complexidade computacional formalmente.

Dado um conjunto finito de símbolos Σ (alfabeto), uma linguagem L é um subconjunto de $\Sigma^\star$. Seja Σ um conjunto finito de símbolos e $c : D_\Pi \longrightarrow \Sigma^\star$, uma codificação de instâncias dum problema de decisão Π em $\Sigma^\star$. A *linguagem associada a* Π para a codificação c é:

$$\begin{aligned} L(\Pi, c) \quad = \quad & \{x \in \Sigma^\star \mid x = c(\mathtt{I}) \text{ onde } \mathtt{I} \text{ é uma instância para a qual a} \\ & \text{resposta de } \Pi \text{ é "sim"}, \mathtt{I} \in \mathtt{S}_\Pi\} \end{aligned}$$

Resolver o problema Π equivale a reconhecer $L(\Pi, c)$ usando um modelo computacional, por exemplo, uma máquina de Turing. Mais que isso, se as codificações usadas forem razoáveis podemos falar apenas na linguagem L associada a um problema Π. Isto é, se c e c' forem duas codificações razoáveis de um problema Π as linguagens $L(\Pi, c')$ e $L(\Pi, c)$ terão as mesmas propriedades (formais, computacionais).

Dado que a complexidade temporal dum algoritmo é uma função do comprimento dos dados, isto é, de cada instância do problema, é necessário obter esta medida independentemente das codificações. Assim, para cada problema de decisão Π associámos, independentemente da codificação, uma função comprimento $comp : \mathtt{D}_\Pi \longrightarrow \mathbb{N}$ nas instâncias $\mathtt{I}$ de Π que é *relacionada polinomialmente*[2] com o comprimento da sequência de símbolos de qualquer codificação razoável de $\mathtt{I}$. Isto é, para qualquer codificação c de Π existem das funções polinomiais p e p' tal que se $\mathtt{I} \in \mathtt{D}_\Pi$ e $x = c(\mathtt{I})$ então $comp(\mathtt{I}) \leq p(|x|)$ e $|x| \leq p'(comp(\mathtt{I}))$. Por exemplo, no problema de decisão do "caixeiro viajante", **(TSP)**, (ver página 55) o comprimento duma instância genérica $\mathtt{I}$ é:

$$comp(I) \quad = \quad m + \lceil \log_2 B \rceil + max\{\log_2 d(c_i, c_j) \mid c_i,\, c_j\, \in C\}$$

Exercício 2.2.2 Escolha uma codificação para o problema TSP e verifique que a função *comp* se "relaciona polinomialmente" com essa codificação.

Podemos então considerar que todos os problemas são codificados numa mesma codificação razoável. Isto, permite dum modo informal comparar as propriedades computacionais de dois problemas sem ter de os exprimir em termos de cada uma das suas codificações. Basta saber que existe uma. Exemplo duma codificação seria considerar as sequências finitas sobre o o conjunto $\Sigma = \{0, 1\}$. Então, qualquer problema de decisão pode ser identificado com uma linguagem $L \in \{0, 1\}$. Normalmente, para escrever algoritmos numa dada linguagem de programação, usam-se codificações menos minimalistas ... Ver exercício 2.6.8 da secção 2.6.2.

Informalmente podemos falar na *complexidade* de um problema e dizer que um problema *pertence* a uma dada classe de complexidade. Isto é, identificar um problema de decisão com a linguagem associada. Toda a teoria a seguir apresentada tem como base esta equivalência e o que será dito para problemas e algoritmos é exprimível formalmente em termos de linguagens e modelos computacionais.

A menos referência em contrário, os problemas considerados serão problemas de decisão. Note-se que dado um problema de optimização – em que não só se pretende saber se um problema tem solução, mas também qual é essa solução (óptima) – é sempre possível formular um problema correspondente de decisão. O problema de decisão é tratável se o de optimização o for, isto é, se o de decisão é intratável o de optimização também o é (o problema de decisão não é mais difícil que o correspondente de optimização). O inverso também é verdade para muitos casos. Consideremos o seguinte exemplo:

(Clique máximo) Um subgrafo completo maximal dum grafo G chama-se clique. O tamanho dum clique é o seu número de vértices.

Instância: Um grafo $G = (V, E)$ não dirigido.

Questão: Qual o tamanho do maior clique de G ?

O problema de decisão associado é:

(CLIQUE)

Instância: Um grafo $G = (V, E)$ não dirigido e um inteiro $k \leq |V|$

Questão: G contém algum clique com tamanho maior ou igual a k ?

[2] Lembre-se que, para um dado problema, apenas estamos interessados na existência de algoritmos polinomiais para o resolver.

Seja `CLID(G,k)` um algoritmo que resolve o problema de decisão **CLIQUE**. Se $|V| = n$, então o tamanho do maior clique pode ser determinado aplicando `CLID(G,k)` para $k = n, n-1, n-2, \ldots$ até que o resultado desse algoritmo seja "sim". Se `CLID` tem complexidade $f(n)$, então algoritmo para determinar o maior clique tem complexidade $n \times f(n)$. Inversamente, se o tamanho do maior clique pode ser determinado em tempo $g(n)$ então o problema de decisão pode ser resolvido em tempo $g(n)$.

Exercício 2.2.3 *Dado um conjunto U de variáveis lógicas uma atribuição de valores para U é uma função $t : U \longrightarrow \{0,1\}$. Se $t(u) = 1$, $u \in U$, dizemos que u é* verdade *em t, caso contrário dizemos que a variável u é* falsa *em t. Se $u \in U$ então u e $\overline{u}$ são literais em U. O literal u é* verdade *em t sse a variável u é* verdade *em t. O literal $\overline{u}$ é* verdade *em t sse a variável u é* falsa *em t. Uma cláusula em U é um conjunto (finito) de literais em U e é satisfeita por uma atribuição de valores, t, sse pelo menos um dos seus elementos é verdade para t. Considere, então o seguinte problema:*

(SATA)

Instância: *Seja $C = \{c_1, c_2, \ldots, c_m\}$ um conjunto de cláusulas com literais num conjunto finito $U = \{u_1, u_2, \ldots, u_n\}$*

Questão: *Determine uma atribuição de valores para U que satisfaça simultaneamente todas as cláusulas de C?*

Mostre que se existir um algoritmo polinomial `SAT(C)` *que determine se um conjunto de cláusulas é satisfazível então o problema* `SATA` *também é resolúvel em tempo polinomial.*

<u>Sugestão:</u>

Vamos supor que em C ocorrem todos os elementos de U. Basta então construir um algoritmo que utilize `SAT()` *n vezes, de cada vez para um* `C´` *que resulta de* `C` *substituindo sucessivamente cada uma das variáveis por o valor* `0` *ou* `1`*:*

```
C´<- C;
para  todos u[i] em U  faça {
   C´´ <- obtido de C´ substituindo 0 por todas as ocorrências de
                 u[i] e 1 por todas as ocorrências de ~u[i];
   se SAT(C´´) ="sim" então { t(u[i])<-0; C´<- C´´ }
                      senão { t(u[i])<- 1;
                             C´<- obtido  de  C´  substituindo  1
                                  por  todas  as ocorrências de u[i]
                                  e 0 por todas as ocorrências de ~u[i]}
 }
 fim.
```

2.2.2 Algoritmos determinísticos

A noção de algoritmo que usamos até agora e que é a habitual, pressupõe que cada instrução é univocamente determinada, isto é, para cada conjunto de dados existe uma só solução. Chamemos a estes algoritmos - que correspondem a máquinas de Turing determinísticas que param sempre ou a programas executáveis por um computador - algoritmos determinísticos. Podemos então identificar a classe **P** com:

P = {conjunto de todos os problemas de decisão que são resolúveis por um algoritmo determinístico em tempo polinomial}

Da lista de problemas dada (EC), (2SAT) e (2-COLOR) pertencem a **P**.

Exercício 2.2.4 *Encontre um algoritmo determinístico polinomial para cada um desses problemas. Sugestão:*

a) **EC** *Para um grafo (não dirigido) ter um ciclo euleriano tem de ser conexo e todos os vértices terem grau par. Estas duas condições podem ser implementadas em tempo polinomial.*

b) **2SAT** *Sendo* $U = \{u_1, \ldots, u_n\}$ *e* C *o conjunto de cláusulas, construa o grafo (dirigido)* $G = (V, E)$*, onde* $V = \{u_1, \ldots, u_n, \bar{u_1}, \ldots, \bar{u_n}\}$ *e sendo* l_i *igual a* u_i *ou* $\bar{u_i}$*,* $(l_i, l_j) \in E$ *se* $\{\bar{l_i}, l_j\} \in C$ *(isto é,* $u_i \Rightarrow l_j$*). Mostre que* C *é satisfazível se e só se as componentes fortemente conexas de* G *não contêm simultaneamente uma variável e a sua negação.*

c) **2-COLOR** *Para colorir um grafo com 2 cores vamos supor que o grafo é conexo (senão aplicámos o algoritmo a cada componente conexa). O seguinte algoritmo é* $O(n^3)$*, onde* n *é a ordem do grafo:*

```
colorir o vértice 1;
f<- verdade;
enquanto f faça { /* enquanto existirem vértices
                    por  colorir */
   f <- falso;
   para i<- 1 até n faça     /* pelo menos um vérice
                               é colorido */
     se não colorir(i) então f<-  verdade;
}
escreva(´sim´);
fim.

colorir(i){
  se i está colorido retorna verdade
  se i não tem vizinhos coloridos
            então retorna falso /* i tem no maximo n-1
                                  vizinhos */
            senão se os vizinhos tiverem cores contraditórias
                   então { escreva(´não´) ; fim }
                   senão {colorir i com a cor alternativa;
                          retorna verdade}
}
```

2.2.3 Algoritmos não determinísticos

Considere-se o problema do "caixeiro viajante" expresso como um problema de decisão:

(TSP) Dado um conjunto de cidades, as distâncias entre elas, e um limite B, existe um itinerário pelas cidades cuja distância total é menor ou igual a B ?

Instância: $C = \{c_1, c_2, \ldots c_n\}$, $d : C \times C \longrightarrow \mathbb{N}$ distância e $B \in \mathbb{N}$

Questão: Existe uma permutação $\pi : C \longrightarrow C$ tal que

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}) \leq B$$

Como se disse não se conhece nenhum algoritmo polinomial que resolva este problema. Contudo, suponhamos que temos um mecanismo que nos gera uma possível solução. Podemos verificar se ela realmente é uma solução, bastando para isso calcular o seu comprimento e compará-lo com B! E isto pode ser feito em tempo polinomial...

O mecanismo mencionado acima pode ser simulado por um algoritmo que permita a existência de instruções cujo resultado não é univocamente determinado mas sim limitado a um conjunto finito de possibilidades: algoritmo não-determinístico de verificação Um tal algoritmo termina com insucesso (resposta "não") se e só se nenhum conjunto de escolhas conduz a uma resposta "sim" (caso em que, o algoritmo termina com sucesso). Se existir algum conjunto de escolhas que conduza a uma solução de sucesso, esse conjunto é sempre gerado, isto é, em qualquer instrução que envolva uma escolha é escolhido um elemento correcto dum desses conjuntos. Para tal efeito ser o usadas a função e as instruções seguintes:

`escolha(S)` escolha arbitrária de um elemento de S

`sucesso` indica que o algoritmo termina com a resposta "sim"

`insucesso` indica que o algoritmo termina com a resposta "não"

Supõe-se que cada uma destas instruções é executada em tempo constante, isto é, $O(1)$. Por exemplo, o problema da procura dum elemento numa sequência pode ser "resolvido" não-deterministicamente pelo seguinte algoritmo:

```
i <-  escolha(1..n)
se V[i]= X então {escreva('sim'); sucesso}
           senão{ escreva('não'); insucesso}
```

Neste caso se existir algum $V[i]$ igual a X, um desses i é gerado na primeira instrução e o algoritmo termina em sucesso, caso contrário termina em insucesso. Mais precisamente, comparando com a noção clássica de algoritmo, o algoritmo *não resolve*[3] o problema, apenas *verifica* se uma possível solução é efectivamente solução.

Exercício 2.2.5 *Para cada um dos seguintes problemas escreva um algoritmo não-determinístico:*

a) Ordenar uma sequência de inteiros positivos.

b) **(HC)** *Existência de ciclo hamiltoniano num grafo (dirigido ou não).*

c) **(CLIQUE)**

Admitindo processamento paralelo, podemos interpretar um algoritmo não-determinístico do seguinte modo: Sempre que existe uma escolha o algoritmo produz várias cópias de si mesmo, uma para cada uma das possíveis alternativas. Estas cópias são executadas em simultâneo. Se uma chega a um resultado de `sucesso` todas as outras terminam. Se uma cópia chega a um resultado de `insucesso` só essa cópia termina.

[3] Isto porque não é indicado um método para efectuar cada uma das escolhas.

O tempo requerido por um algoritmo não-determinístico para um dado conjunto de dados é o menor número de passos necessários para chegar a um resultado de `sucesso`. Se tal não for possível o tempo requerido é $O(1)$. A sua complexidade é $O(f(n))$ se para todos os dados de comprimento n, $n \geq n_0$ que conduzem a um resultado de sucesso o, tempo requerido é no máximo $cf(n)$, para c e n_0 constantes.

O algoritmo não-determinístico dado para procurar um elemento numa sequência é $O(1)$.

Um algoritmo não-determinístico diz-se *polinomial* se a sua complexidade é $O(p(n))$, para algum polinómio, $p(n)$.

Formalmente podemos identificar estes algoritmos e com máquinas de Turing não-determinísticas polinomiais e os problemas correspondentes com linguagens da classe **NP**. Temos:

$$\begin{aligned}\mathbf{NP} = \{&\text{conjunto de todos os problemas de decisão que são resolúveis por um algoritmo}\\ &\text{não-determinístico polinomial}\}\end{aligned}$$

Exercício 2.2.6 *Determine a complexidade temporal de cada um dos algoritmos encontrados no exercício 2.2.5.*

Exercício 2.2.7 *Obtenha um algoritmo não-determinístico de $O(n)$ que determine se existe um subconjunto de n números a_i, $1 \leq i \leq n$ cuja soma seja M.*

A noção de *verificação polinomial* duma solução pode ainda formalizar-se em termos de *testemunhas concisas* duma linguagem.

Teorema 2.2.1 *Uma linguagem $L \subseteq \Sigma^\star$ pertence a* **NP** *se e só se existe uma linguagem $L_v \subseteq \Sigma^\star \times \Sigma^\star$ que pertence a* **P** *e uma função polinomial p tal que:*

$$L = \{x \mid \exists y (x, y) \in L_v \text{ e } |y| \leq p(|x|)\}$$

Para cada $x \in L$, y diz-se uma testemunha *concisa (ou polinomial) da $x \in L$.*

Dem. ($\Leftarrow$) Suponhamos que $L = \{x \mid \exists y (< x, y > \in L_v \text{ e } |y| \leq p(|x|)\}$ e $L_v \in \mathbf{P}$ e $p(n)$ é polinomial. Então L é decidida pela seguinte NDTM N: Com dados x, N gera uma sequência y de comprimento no máximo $p(|x|)$ e depois usa a TM que decide L_v para testar (verificar) em tempo polinomial se $(x, y) \in L_v$. Se $(x, y) \in L_v$ N aceita x caso contrário rejeita.[4]

($\Rightarrow$) Seja $L \in \mathbf{NP}$. Então existe uma NDTM N que decide L e cuja complexidade temporal é limitada por uma função polinomial $p(n)$. Para cada $x \in \Sigma^\star$, cada caminho de computação (sequência de escolhas de Δ) de N com dados x pode ser codificado numa sequência y de símbolos de Σ tal que $|y| \leq p(|x|)$. A linguagem L_v é definida do seguinte modo: um par $< x, y >$ pertence a L_v se y codifica um caminho duma computação de aceitação de N com dados x. É fácil ver que $L_v \in \mathbf{P}$ e que $x \in L$ se e só se $\exists y (< x, y > \in L_v$ e $|y| \leq p(|x|)$. $\square$

Alternativamente, podemos ainda considerar uma variante de máquinas de Turing determinísticas: máquinas de Turing com "escolhas" (de "adivinhação") polinomiais.

Exercício 2.2.8 Considere a seguinte definição duma máquina de Turing com "escolhas" (de "adivinhação")(GDTM) (ver [GJ79, Cap. 2.3]): uma GDTM, M, é uma máquina de Turing determinística com fita duplamente infinita à qual foi adicionado um módulo de *escolha* com apenas uma cabeça de escrita. A computação duma máquina GDTM para dados $x \in \Sigma^*$ é feita em dois passos:

[4] Note que o número de sequências y é finito.

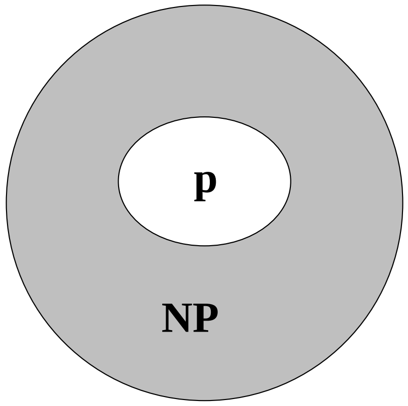


Figura 2.1: Relação entre **P** e **NP**

Escolha Inicialmente os dados x estão escritos na fita a partir da posição 1 até $\mid x \mid$, os restantes em branco, a cabeça de leitura-escrita na posição 1 e a cabeça de escrita a célula -1. O módulo de *escolha* dirige então a cabeça de escrita, ou escrevendo um símbolo de Γ e movendo-se para a esquerda, ou parando; neste último caso este módulo fica inactivo e entra-se no estado s_0. Note que este módulo escreveu um qualquer $s \in \Gamma^*$.

Verificação A computação procede exactamente como numa máquina de Turing determinística

A computação pára se é atingido um estado final e diz-se que é uma computação de aceitação se pára num estado de aceitação. Dizemos que M aceita x se existir pelo menos uma computação de aceitação para x.

Mostre que uma máquina GDTM é equivalente (polinomialmente) a uma NDTM. Sugestão: Considere a demonstração do teorema 2.2.1.

2.2.4 Relação entre P e NP

Já se viu que $\mathbf{P} \subseteq \mathbf{NP}$.

Pelo que foi dito no final da secção anterior, um algoritmo não-determinístico é essencialmente constituído por dois passos:

- escolher uma solução
- verificar se é solução

Se é polinomial isso equivale a dizer que a *verificação* da solução do problema pode ser feita em tempo polinomial. Temos assim que a distinção entre as classes **P** e **NP** corresponde diferença entre a resolução dum problema em tempo polinomial e a sua verificação em tempo polinomial.

Será que $\mathbf{NP} \subseteq \mathbf{P}$? Até agora ainda ninguém soube respondera esta pergunta !

Exercício 2.2.9 *Mostrar que se $\Pi \in \mathbf{NP}$ então existe um polinómio $p(n)$ tal que Π pode ser resolvido por um algoritmo determinístico com complexidade $O(2^{p(n)})$.*

2.2.5 Reduções entre problemas

Um problema `A` reduz a um problema `B` se dado um algoritmo para resolver `B` podemos construir um algoritmo que resolva `A`. Este conceito de redutibilidade é central na teoria da computabilidade e também vai ser fulcral na teoria dos problemas **NP**-completos.

Facto: Muitos dos problemas para os quais não se conhecem algoritmos polinomiais, reduzem-se polinomialmente uns aos outros de tal forma que se um deles tiver um algoritmo polinomial todos os outros têm !

Formalmente temos:

Definição 2.2.1 *Uma transformação polinomial duma linguagem* $L_1 \subseteq \Sigma_1^\star$ *numa linguagem* $L_2 \subseteq \Sigma_2^\star$ *é uma função* $f : L_1 \longrightarrow L_2$ *que satisfaz as seguintes condições:*

(i) Existe uma máquina de Turing polinomial que calcula f

(ii) Para todo o $x \in \Sigma_1^\star$, $x \in L_1$ *se e só se* $f(x) \in L_2$

Em termos de problemas, dizemos que um problema A *reduz-se polinomialmente* a um problema B, $A \propto B$, se e só se existe um algoritmo determinístico polinomial, F, que transforma cada instância $X \in A$ numa instância $F(X) \in B$, tal que X tem solução se e só se $F(X)$ tiver.

Proposição 2.2.1 *Se* $L_1 \propto L_2$ *e* $L_2 \in \mathbf{P}$ *então* $L_1 \in \mathbf{P}$.

Proposição 2.2.2 *A relação* $\propto$ *é transitiva.*

Exercício 2.2.10 *Mostrar as proposições 2.2.1 e 2.2.2.*

Duas linguagens L_1 e L_2 são *polinomialmente equivalentes* se $L_1 \propto L_2$ e $L_2 \propto L_1$. Análogamente, se definem problemas *polinomialmente equivalentes.* A classe $\mathbf{P}$ é uma classe de equivalência dessa relação, formada pelas linguagens mais "fáceis".

Vejamos alguns exemplos de reduções polinomiais !

Exemplo 6 HC $\propto$ TSP

(HC)

Instância: *Um grafo não dirigido* $G = (V, E)$ *e* $|V| = m$

Questão: G *tem um ciclo hamiltoniano?*

(TSP)

Instância: *Um conjunto finito* $C = \{c_1, c_2, \ldots c_n\}$ *de cidades,* $d : C \times C \longrightarrow \mathbb{N}$ *distância e* $B \in \mathbb{N}$.

Questão: *Existe uma permutação* $\pi : C \longrightarrow C$ *tal que*

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}) \leq B$$

A transformação polinomial, f, *é definida do seguinte modo: Seja* $G = (V, E)$ *e* $|V| = m$ *uma instância de HC. A instância correspondente de TSP é dada por:*

(i) $C = V$

(ii) Para cada par $v_i, v_j \in C$, $d(v_i, v_j) = 1$ *se* $(v_i, v_j) \in E$ *e* $d(v_i, v_j) = 2$, *caso contrário.*

(iii) $B = m$

Vamos ver duma forma informal que f é uma transformação polinomial de HC para TSP.

1. *Para construir as $m(m-1)/2$ distâncias $d(v_i, v_j)$, é necessário apenas verificar se $(v_i, v_j) \in E$. Isto pode ser feito em tempo polinomial.*

2. *Vejamos que G tem um ciclo hamiltoneano se e só se existe uma permutação π cuja soma das distâncias é menor que B. Suponhamos que $< v_1, \ldots, v_m >$ é um ciclo Hamiltoneano de G. Então, $< v_1, \ldots, v_m >$ é uma volta em $f(G)$ com distância total m=B. Inversamente, suponhamos que $< v_1, ..., v_m >$ é uma volta em $f(G)$ com distância total $\leq B$. Pela construção de d, por B=m, e pelo facto de exactamente m cidades serem visitadas, a distância entre duas cidades consecutivas tem de ser 1. Então, todos os pares (v_i, v_{i+1}), $1 \leq i \leq m$ e (v_1, v_m) pertencem a E e constituem um ciclo hamiltoniano para G.*

Concluímos então que se existir um algoritmo polinomial para TSP, então também podemos construir um algoritmo determinístico polinomial para HC e se HC for intratável, TSP também o é.

Genericamente se $\Pi_1 \propto \Pi_2$ dizemos que Π_2 é *pelo menos tão difícil como* Π_1.

Consideremos agora uma redução entre dois problemas de dois domínios diferentes: lógica e teoria de grafos.

Exemplo 7 3SAT $\propto$ VC

(3SAT)

Instância: *Seja $C = \{c_1, c_2, \ldots, c_m\}$ um conjunto de cláusulas com literais num conjunto finito U de variáveis lógicas, $| U |= n$, e tal que cada cláusula tem exactamente 3 literais.*

Questão: *Existe uma atribuição de valores de verdade para U que satisfaz simultaneamente todas as cláusulas de C?*

(VC)

Instância: *Um grafo não dirigido $G = (V, E)$ e $K \leq| V |$ inteiro positivo.*

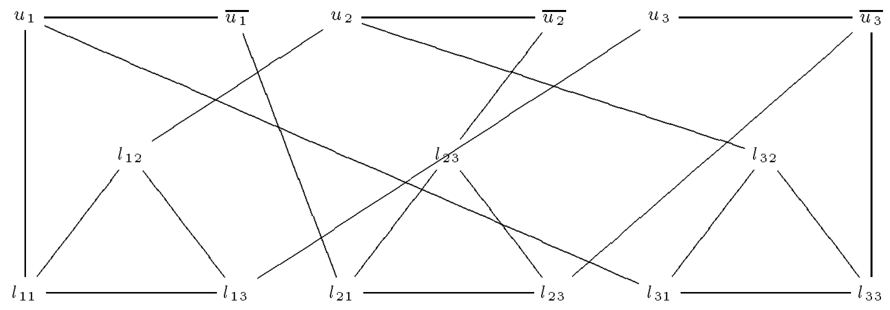
Questão: *Existe um subconjunto $V' \subseteq V$ tal que $| V' |\leq K$ e para cada ramo $\{u, v\} \in E$, $u \in V'$ ou $v \in V'$? (A V' chama-se* cobertura por vértices*de G.)*

Construção da transformação : *Suponhamos uma instância de* **3SAT**. *Vamos construir uma instância de* **VC** *indicando os conjuntos V e E. Para cada $u_i \in U$, $\{u_i, \overline{u_i}\} \in V$ e $\{u_i, \overline{u_i}\} \in E$. Note-se que uma cobertura por vértices deve conter pelo menos um dos vértices destes ramos. Para cada $c_j \in C$ adicionar a V três vértices correspondentes a cada um dos literais – l_{j1}, l_{j2}, l_{j3} – que ocorre em c_j e adicionar a E os ramos que unem estes vértices, de modo a formar um triângulo para cada cláusula (ver figura). Qualquer cobertura por vértices dever conter pelo menos 2 dos vértices de cada triângulo. Finalmente, para cada cláusula constroem-se ramos que unem cada literal l_{jk}, $k = 1, 2, 3$, ao literal correspondente, u_i ou $\overline{u_i}$.Seja $K = n + 2m$ e $G = (V, E)$, onde*

$$
\begin{aligned}
V &= U \cup \overline{U} \cup \bigcup_{i=1}^{m} \{l_{i1}, l_{i2}, l_{i3}\} \\
E &= \bigcup_{i=1}^{n} \{\{u_i, \overline{u_i}\}\} \cup \bigcup_{i=1}^{m} \{\{l_{i1}, l_{i2}\}, \{l_{i1}, l_{i3}\}, \{l_{i2}, l_{i3}\}\} \cup \bigcup_{i=1}^{m} \bigcup_{j=1}^{3} \{\{l_{ij}, \mu\}\},
\end{aligned}
$$

onde μ designa o literal correspondente u_k ou $\overline{u_k}$, para algum k, $1 \leq k \leq n$. Temos que $| V | = 2n + 3m$ e $| E | = n + 6m$. Toda esta construção pode ser feita em tempo polinomial.

Considerando a instância de **3SAT**, $U = \{u_1, u_2, u_3\}$ *e* $C = \{(u_1, \overline{u_2}, u_3), (\overline{u_1}, \overline{u_2}, \overline{u_3}), (u_1, u_2, \overline{u_3})\}$ *temos o seguinte grafo G: associado a* **VC**, *com* $K = 11$ *e* $G = (V, E)$:



Falta ver que C é satisfeita se e só se G tem uma cobertura por vértices de tamanho no máximo K.

Suponhamos que $V' \subset V$ é uma cobertura por vértices para G. Pelo que se disse acima V' contem exactamente um de cada $\{u_i, \overline{u_i}\}$ e 2 vértices l_{jk} correspondentes a cada cláusula. Então a seguinte atribuição de valores de verdade satisfaz C, $t : U \longrightarrow \{0, 1\}$:

$$t(u_i) = \begin{cases} 1 & se\ u_i \in V' \\ 0 & se\ \overline{u_i} \in V' \end{cases}$$

Exercício 2.2.11 Verificar que t satisfaz C.

Inversamente suponhamos que $t : U \longrightarrow \{0, 1\}$ é uma atribuição de valores de verdade que satisfaz C. A cobertura para V é a seguinte: $u_i \in V'$ se $t(u_i) = 1$ e $\overline{u_i} \in V'$ se $t(u_i) = 0$. Isto assegura que para cada cláusula uma dos ramos que unem l_{ji} aos literais u_i ou $\overline{u_i}$ fique coberta por V'. Basta então, adicionar a V' dois dos vértices da cada triângulo que não correspondem a esse literal.

Exercício 2.2.12 Verificar que V' é uma cobertura por vértices de G e $| V' | \leq K$.

2.3 Problemas NP-completos

Definição 2.3.1 *Uma linguagem L é* **NP**-*completa se $L \in$* **NP** *e para todo $L' \in$* **NP** *tem-se que $L' \propto L$.*

Em termos de problemas, um problema Π diz-se **NP**-completo se $\Pi \in$ **NP** e para todo $\Pi' \in$ **NP** tem-se que $\Pi' \propto \Pi$.

Pela proposição 2.2.1 os problemas **NP**-completos (representamos por **NPC** a classe destes problemas) são os mais difíceis em **NP**. Se um problema em **NP** é intratável então todos os problemas em **NPC** também o são. Se um problema **NP**-completo tiver um algoritmo determinístico polinomial então todos os problemas em **NP** tem também um, isto é, **P**=**NP**! Por outro lado, se **P** $\neq$ **NP** e $\Pi \in$ **NPC** então $\Pi \in$ **NP** $\setminus$ **P**. Note-se ainda que todos os problemas na classe **NPC** são polinomialmente equivalentes. Se **P** $\neq$ **NP** então a classe **NP** divide-se como indicado na figura 2.2.

No entanto, não parece claro como se pode mostrar que um problema é **NP**-completo: para cada problema em **NP** (que são em número infinito) é necessário arranjar uma redução polinomial para esse problema. A proposição seguinte mostra que se existir um problema **NP**-completo é mais fácil provar que outro problema é **NP**-completo.

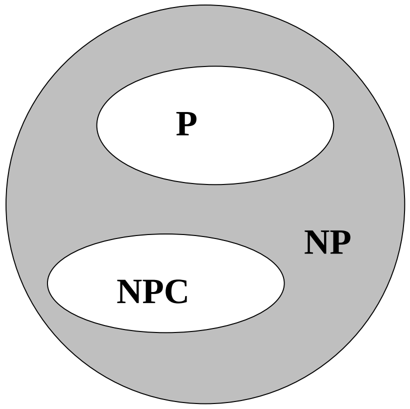


Figura 2.2: Classe **NP**

Proposição 2.3.1 *Se L_1 e L_2 pertencem a* **NP**, *L_1 é* **NP**-*completa e $L_1 \propto L_2$ então L_2 é* **NP**-*completa.*

Exercício 2.3.1 Mostrar a proposição 2.3.1.

Cook mostrou que SAT é NP-completo !
Dado que para cada problema Π em **NP** existe uma algoritmo polinomial não-determinístico `A` que o resolve. Cook, [Coo71], provou que é possível obter em tempo polinomial a partir desse algoritmo (genérico) e duma instância `I` do problema Π , um conjunto de cláusulas C, tal que C é satisfeito se e só se `A` termina com sucesso para essa instância `I` .

Teorema 2.3.1 **SAT** *é* **NP**-*completo.*

Dem. Temos que mostrar:

(i) $\mathbf{SAT} \in \mathbf{NP}$

(ii) $\forall \Pi \in \mathbf{NP} \implies \Pi \propto \mathbf{SAT}$

Seja $U = \{u_1, u_2, \ldots, u_n\}$ um conjunto de variáveis lógicas e $C = \{c_1, c_2, \ldots, c_m\}$ um conjunto de cláusulas com literais de U. Então, o seguinte algoritmo não-determinístico termina com sucesso se e só se C é satisfeito:

```
para i <- 1 até n faça
    u[i] <- escolha({0,1})
para i <- 1 até m faça
     se c[i] = 0  então insucesso
sucesso.
```

O tempo de execução deste algoritmo é $O(n)$ para escolher uma atribuição de valores de verdade para as variáveis, mais $O(m)$ para verificar se cada cláusula de C é satisfeita, portanto é proporcional ao comprimento dos dados. Logo, $\mathbf{SAT} \in \mathbf{NP}$.

Seja $L_{SAT} = L[SAT, c]$ a linguagem associada a **SAT** para uma dada codificação c. Temos que mostrar $\forall\ L \in \mathbf{NP},\ L \propto L_{SAT}$. Cada $L \in \mathbf{NP}$ é decidida por uma GDTM polinomial (ver exercício 2.2.8). Seja, então, M uma GDTM polinomial arbitrária, $M =$

$(S, \Sigma, \Gamma, b, s_0, \{s_y, s_n\}, \delta)$ tal que $L(M) = L$ e seja $T_M(n) = p(n)$, com $p(n) \geq n$, $\forall n \in \mathbb{N}$. A transformação genérica F_L será definida de $\Sigma^\star$ em instâncias de **SAT** (mais propriamente Σ_{SAT}), tal que para $x \in \Sigma^\star$, $x \in L$ se e só se $F_L(x)$ é satisfazível. Isto é, $x \in \Sigma$ é aceite por M se e só se $F_L(x)$ é um conjunto de cláusulas satisfazível. A ideia é "simular" a computação de M em termos de variáveis lógicas. Se $x \in \Sigma^\star$ é aceite por M então existe uma computação que dado x termina no estado s_y e que tanto o número de passos na fase de verificação como o número de símbolos na sequência "adivinhada"são limitados por $p(n)$, sendo $n = |x|$. Cada computação deste tipo envolve no máximo as posições de $-p(n)$ a $p(n) + 1$. Recorde-se que cada configuração, ou descrição instantânea de M, fica caracterizada pelo conteúdo das células da fita, pelo estado corrente e a célula lida pela cabeça. Como não há mais de $p(n)$ passos na fase de verificação, existem no máximo $p(n) + 1$ configurações a considerar. Podemos, então, associar a cada computação um número limitado de variáveis lógicas e uma atribuição de valores de verdade para elas. Numerem-se os elementos de S s_0, $s_1 = s_y$, $s_2 = s_n$, $s_3, \ldots, s_r$, onde $r = |S| - 1$, e os elementos de Γ, $a_0 = b$, $a_1, \ldots, a_v$, onde $v = |\Gamma| - 1$. Consideram-se 3 tipos de variáveis:

Variável	*Variação*	*Significado*
$S[i,k]$	$0 \leq i \leq p(n)$ $0 \leq k \leq r$	no passo i M está no estado s_k
$H[i,j]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n) + 1$	no passo i a cabeça está na célula j
$A[i,j,k]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n) + 1$ $0 \leq k \leq v$	no passo i o conteúdo da célula j é o símbolo a_k

Uma computação de M induz uma atribuição de valores de verdade às variáveis acima definidas duma maneira óbvia, com a convenção de que se o programa terminar em $t \leq p(n)$ passos, a configuração mantém-se a mesma de $t+1$ a $p(n)$. No passo 0 o conteúdo da fita tem os dados x, escritos nas células de 1 a n e a "advinha" w escrita nas células de -1 a $-|w|$, com as restantes células em branco[5]. A transformação F_L constrói uma conjunto de cláusulas envolvendo estas variáveis de tal modo que a atribuição de valores de verdade satisfaz esse conjunto sse é a atribuição de valores de verdade induzida por uma computação que aceita x cuja fase de verificação demora $p(n)$ ou menos passos e cuja sequência "adivinhada"tem comprimento no máximo $p(n)$. Tem-se

$x \in L$	$\Leftrightarrow$	existe uma computação de M que aceita x
	$\Leftrightarrow$	existe uma computação de M que aceita x com $p(n)$ ou menos passos na fase de verificação e com uma sequência adivinhada w de comprimento exactamente $p(n)$
	$\Leftrightarrow$	existe uma atribuição de valores de verdade que satisfaz o conjunto de cláusulas em $F_L(x)$

As cláusulas em $F_L(x)$ podem ser divididas em 6 grupos:

[5]Note que uma qualquer atribuição de valores de verdade para estas variáveis não corresponde provavelmente a nenhuma computação de M.

Grupo	Restrições impostas	Cláusulas nesse Grupo
G_1	No passo i, M está exactamente num só estado.	$\{S[i,0], S[i,1], \ldots, S[i,r]\}$ $\quad 0 \le i \le p(n)$ $\{\overline{S[i,j]}, \overline{S[i,j']}\}$ $\quad 0 \le i \le p(n)$, $0 \le j < j' \le r$
G_2	No passo i a cabeça está exactamente numa só célula da fita.	$\{H[i,-p(n)], H[i,-p(n)+1], \ldots, H[i,p(n)+1]\}$ $0 \le i \le p(n)$ $\{\overline{H[i,j]}, \overline{H[i,j']}\}$ $0 \le i \le p(n)$ $-p(n) \le j < j' \le p(n)+1$
G_3	No passo i, cada célula contém exactamente um símbolo de Γ.	$\{A[i,j,0], A[i,j,1], \ldots, A[i,j,v]\}$ $0 \le i \le p(n)$ $-p(n) \le j \le p(n)+1$ $\{\overline{A[i,j,k]}, \overline{A[i,j,k']}\}$ $0 \le i \le p(n)$, $-p(n) \le j \le p(n)+1$ $0 \le k < k' \le v$
G_4	No passo 0, a computação está na configuração inicial para a fase de verificação para dados $x = a_{k_1} \ldots a_{k_n}$.	$\{S[0,0]\}, \{H[0,1]\}, \{A[0,0,0]\}$, $\{A[0,1,k_1]\}, \ldots, \{A[0,n,k_n]\}$, $\{A[0,n+1,0]\}, \ldots, \{A[0,p(n)+1,0]\}$
G_5	No passo $p(n)$, M entrou no estado s_y.	$\{S[p(n),1]\}$
G_6	Para cada i, $0 \le i < p(n)$, a configuração de M em $i+1$ resulta duma única aplicação de δ dada a configuração em i.	(Ver texto)

É fácil ver que as cláusulas dos grupos G_1 a G_5 cumprem a função especificada para cada grupo e que uma atribuição de valores de verdade para elas corresponde a uma computação que aceita x. Apenas falta especificar as cláusulas do grupo G_6. Consideramos dois subgrupos.

1. Este subgrupo garante que se a cabeça não está no passo i na célula j então o seu símbolo não é mudado de i para $i+1$. As cláusulas são da forma[6]:

$$\{H[i,j], \overline{A[i,j,l]}, A[i+1,j,l]\} \quad \begin{array}{l} 0 \le i < p(n) \\ -p(n) \le j \le p(n)+1 \\ 0 \le l \le v \end{array}$$

2. Este subgrupo garante que as modificações duma configuração para a seguinte estão de acordo com a função de transição δ. Para cada (i,j,k,l), $0 \le i < p(n)$, $-p(n) \le j \le$

[6] Isto é, $\neg H[i,j] \implies (A[i,j,l] \implies A[i+1,j,l])$.

$p(n)+1$, $0 \le k \le r$ e $0 \le l \le v$ tem-se as seguintes 3 cláusulas[7] :

$$\{\overline{H[i,j]}, \overline{S[i,k]}, \overline{A[i,j,l]}, H[i+1,j+D]\}$$
$$\{\overline{H[i,j]}, \overline{S[i,k]}, \overline{A[i,j,l]}, S[i+1,k']\}$$
$$\{\overline{H[i,j]}, \overline{S[i,k]}, \overline{A[i,j,l]}, S[i,j,l']\}$$

onde se $s_k \in S \setminus \{s_y, s_n\}$ então os valores de D,k' e l' são tais que $\delta(s_k, a_l) = (s_{k'}, a_{l'}, D)$ e se $s_k \in \{s_y, s_n\}$ então $D = 0$,$k' = k$ e $l' = l$.

O número de cláusulas por grupo é:

Grupo	Número de cláusulas	
G_1	$(p(n)+1)(1+r(r+1)/2)$	
G_2	$(p(n)+1)(1+u(u+1)/2)$	onde $u = 2(p(n)+1)$
G_3	$2(p(n)+1)^2(1+v(v+1)/2)$	
G_4	$p(n)+4$	
G_5	1	
G_6	$2(p(n)+1)^2(v+1)+6(p(n))(p(n)+1)(r+1)(v+1)$	

Se $x \in L$ então existe uma computação de M com dados x de tamanho $p(n)$ ou menos, e esta computação, dada a interpretação das variáveis, impõe uma atribuição de valores de verdade que satisfaz todas as cláusulas de $C = G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \cup G_6$.

Inversamente, a construção de C é tal que qualquer atribuição de valores de verdade que satisfaça C corresponde a uma computação de M que aceita x. Segue que $F_L(x)$ é satisfazível se e só se $x \in L$.

Apenas resta ver que, para uma linguagem L, a transformação $F_L(x)$ pode ser construída a partir de x em tempo limitado por uma função polinomial em $n = |x|$. Dado L e M uma NTDM que decide L em tempo $p(n)$ tem de se construir o conjunto de variáveis U e o conjunto de cláusulas C. Para ver que a construção da transformação é limitada polinomialmente, basta ver que $Comp[F_L(x)]$ é limitado superiormente por um polinomial em n. Para este problema (**SAT**), dada uma codificação razoável, $Comp[F_L(x)] = |U|.|C|$. Note-se que cada cláusula não pode conter mais de $2|U|$ literais e o número de símbolos necessários para descrever um literal é um factor da ordem $log|U|$. Dado que r e v são fixos e não dependem de x, tem-se $|U| = O(p(n)^2)$ e $|C| = O(p(n)^2)$. Então, $Comp[F_L(x)] = O(p(n)^4)$. □

Assim, usando a proposição 2.3.1, para mostrar que um problema Π é **NP**-completo basta:

a) Mostrar que $\Pi \in$ **NP**

b) Seleccionar um problema $\Pi' \in$ **NPC**

c) Construir uma transformação F de Π' em Π

d) Mostrar que F é uma redução polinomial

Proposição 2.3.2 **3SAT** *é* **NP**-*completo.*

Dem. a) **3SAT** $\in$ **NP** Seja $U = \{u_1, u_2, \ldots, u_n\}$ um conjunto de variáveis lógicas e $C = \{c_1, c_2, \ldots, c_m\}$ um conjunto de cláusulas com 3 literais de U. Então, o seguinte algoritmo não-determinístico termina com sucesso se e só se C é satisfeito:

[7] Isto é $S[i,k] \land H[i,j] \land A[i,j,k] \implies (H[i+1,j+D] \land S[i+1,k'] \land A[i+1,j,l'])$

```
para i <- 1 até n faça
    u[i] <- escolha({0,1})
para i <- 1 até m faça
     se c[i] = 0  então insucesso
sucesso.
```

O tempo de execução deste algoritmo é $O(n)$ para escolher uma atribuição de valores de verdade para as variáveis, mais $O(m)$ para verificar se cada cláusula de C é satisfeita, portanto é proporcional ao comprimento dos dados. Logo, $\mathbf{3SAT} \in \mathbf{NP}$.

b) Seleccionámos **SAT** (é o único problema **NP**-completo que conhecemos...)

c) Vamos transformar **SAT** em **3SAT**. Seja $U = \{u_1, u_2, \ldots, u_n\}$ um conjunto de variáveis lógicas e $C = \{c_1, c_2, \ldots, c_m\}$ um conjunto de cláusulas duma instância de **SAT**. Vamos construir um conjunto de cláusulas C' com 3 literais num conjunto U' de variáveis tal que C' é satisfeito se e só se C o é. Cada cláusula $c_i \in C$ é transformada num conjunto C'_i de cláusulas com 3 literais do conjunto U mais variáveis adicionais (usadas só para as cláusulas de C'_i) dum conjunto U'_i. A estrutura de C'_i e U'_i dependem do número de literais k, em c_i, de acordo com a seguinte tabela:

k	c_i	U'_i	C'_i
1	(l_1)	$\{u_i^1, u_i^2\}$	$\{(l_1, \overline{u_i^1}, \overline{u_i^2}), (l_1, u_i^1, \overline{u_i^2}), (l_1, \overline{u_i^1}, u_i^2), (l_1, u_1^1, u_i^2)\}$
2	(l_1, l_2)	$\{u_i^1\}$	$\{(l_1, l_2, u_i^1), (l_1, l_2, \overline{u_i^1})\}$
3	(l_1, l_2, l_3)	$\emptyset$	$\{(l_1, l_2, l_3)\}$
≥ 4	$(l_1, l_2, \ldots, l_k)$	$\{u_i^1, \ldots, u_i^{k-3}\}$	$\{(l_1, l_2, \overline{u_i^1}), (u_i^1, l_3, \overline{u_i^2}), (u_i^2, l_4, \overline{u_i^3}),$ $\ldots, (u_i^{k-4}, l_{k-2}, \overline{u_i^{k-3}}), (u_i^{k-3}, l_{k-1}, l_k)\}$

e

$$
\begin{aligned}
U' &= U \cup \bigcup_{i=1}^{m} U'_i \\
C' &= \bigcup_{i=1}^{m} C'_i
\end{aligned}
$$

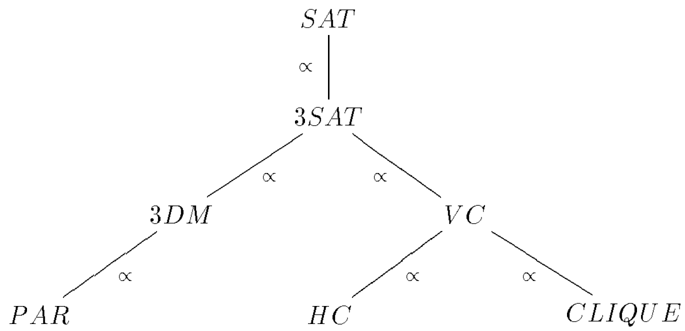
d) Vejamos que C' é satisfeita se e só se C o é. Suponhamos $t : U \longrightarrow \{0,1\}$ uma atribuição de valores de verdade que satisfaz C. Vejamos como t pode ser estendida a uma atribuição de valores de verdade para C', $t' : U' \longrightarrow \{0,1\}$. Para isso basta ver quais os valores de t' para cada elemento de U'_i (e para cada caso verificar que as cláusulas em C' são todas satisfeitas). Dizemos que o conjunto U' é do tipo k se corresponder a uma cláusula em C com k literais, de acordo com a tabela anterior. Se U'_i for do tipo 1 ou 2 qualquer valor pode ser dado aos seus elementos, por exemplo $t'(u_i^j) = 1, u_i^j \in U_i$. Se U' for do tipo 3 é vazio e portanto não é preciso fazer nada. Se U'_i for do tipo k, $k \geq 4$, então pelo menos um dos literais de c_i tem $t(l_m) = 1$, para algum $1 \leq m \leq k$. Se for l_1 ou l_2 então fazemos $t'(u_i^j) = 1$ para $1 \leq j \leq k-3$. Se for l_k ou l_{k-1} fazemos $t'(u_i^j) = 0$ para $1 \leq j \leq k-3$. Caso contrário $t'(u_i^j) = 1$ para $1 \leq j \leq m-2$ e $t'(u_i^j) = 0$ para $m-1 \leq j \leq k-3$.

Inversamente, se t' é uma atribuição de valores de verdade para C' é fácil ver que a restrição de t' a U deve satisfazer C.

e) Note-se que o número máximo de cláusulas em C' é limitado por um polinómio em mn e portanto o tamanho duma instância de **3SAT** é limitado por um polinómio no tamanho da instância de **SAT**. Isto é, a construção da transformação pode ser feita em tempo polinomial. Donde **SAT** $\propto$ **3SAT**, e portanto **3SAT** é **NP**-completo.

□

Em 1972 Karp [Kar72] publica uma lista com 21 problemas que mostra serem **NP**-completos, indicando as reduções entre eles. Dessa lista salientámos 6, que já foram referidos anteriormente, e o diagrama seguinte mostra uma sequência de reduções que pode ser usada para a sua demonstração:



Pelo diagrama e pela redução atrás apresentada podemos concluir que o problema de decisão do "caixeiro viajante" é **NP**-completo: **HC** é **NP**-completo e **HC** $\propto$ **TSP**!

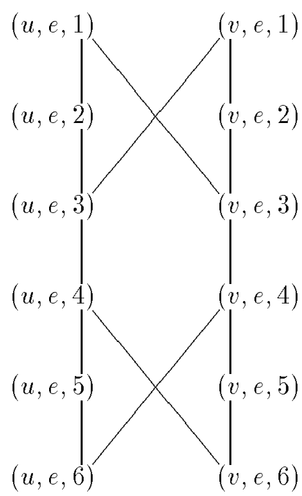
Proposição 2.3.3 **HC** *é* **NP**-*completo.*

Dem. a) **HC** $\in$ **NP** porque um algoritmo não determinístico apenas tem de escolher uma ordenação dos vértices e verificar em tempo polinomial que o conjunto de ramos correspondentes está contido em E.

b) Seleccionámos **VC**

c) Vamos transformar **VC** em **HC**. Seja $G = (V, E)$ e $K \leq \mid V \mid$ uma instância de **VC**. Temos de construir um grafo $G' = (V', E')$ tal que G' tem um ciclo Hamiltoneano se e só se G tem uma cobertura por vértices de tamanho no máximo K. Primeiro, o grafo G' tem K vértices selectores $a_1, a_2, \ldots, a_K$, que serão usados para seleccionar K vértices do conjunto V. Depois, para cada ramo em E, G' contêm uma componente de "teste de cobertura"que será usada para assegurar que pelo menos uma das extremidades desse ramo é um dos K vértices seleccionado. A componente para um ramo $e = \{u, v\} \in E$

é:

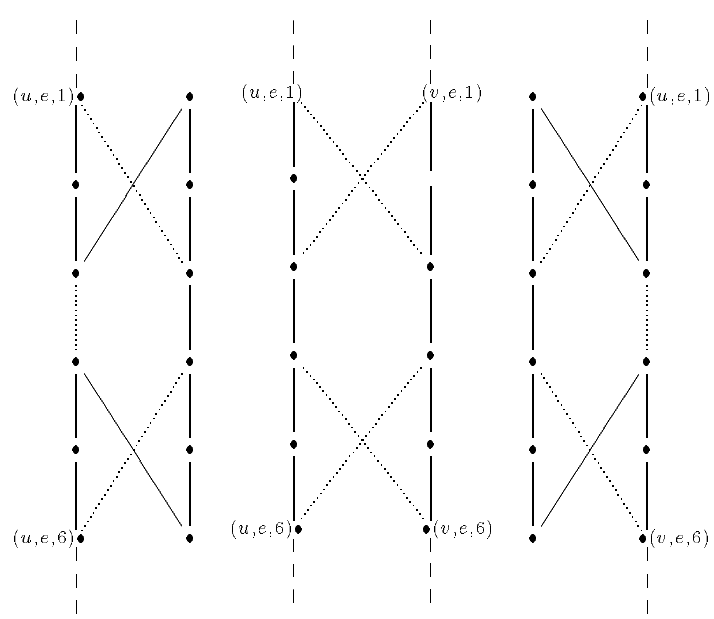


com

$$
\begin{aligned}
V'_e &= \{(u,e,i),(v,e,i) : 1 \le i \le 6\} \\
E'_e &= \{\{(u,e,3),(v,e,1)\},\{(v,e,3),(u,e,1)\}\} \\
&\quad \cup\{\{(u,e,6),(v,e,4)\},\{(v,e,6),(u,e,4)\}\}
\end{aligned}
$$

Cada V'_e tem 12 vértices e cada E'_e tem 14 ramos. Na construção final, os únicos vértices desta componente que entraram noutros ramos são os vértices $(u,e,1)$,$(v,e,1)$, $(u,e,6)$ e $(v,e,6)$. Isto implica que qualquer ciclo Hamiltoneano tem de passar pelos ramos de E'_e de uma das seguintes maneiras:

(a) (b) (c)

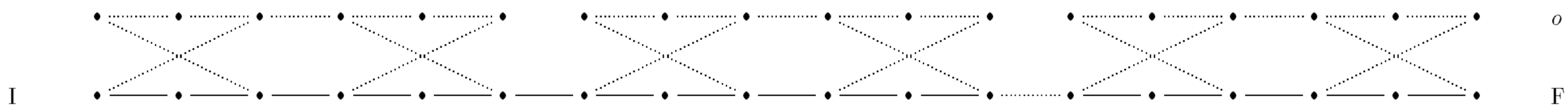


Ramos adicionais serão usados na construção para juntar pares de componentes de "teste de cobertura" ou para juntar componentes a um vértice seleccionador. Para cada

$v \in V$, ordenem-se os ramos incidentes $e_{v[1]}$, $e_{v[2]}$, ..., $e_{v[gr(v)]}$, onde $gr(e)$ é o grau de v, isto é, o número de ramos que incidem em v. Todas as componentes de "teste de cobertura" correspondentes a estes ramos (tendo v como extremidade) são unidas pelos seguintes ramos:

$$E'_v \quad = \quad \{\{(v, e_{v[i]}, 6), (v, e_{v[i+1]}, 1)\} \mid \ 1 \leq i \leq gr(v)\}$$

Cria-se um único caminho em G' que incluí exactamente os vértices da forma (v, y, z):



onde $I = (v, e_{v[1]}, 1)$, $F = (v, e_{v[gr(v)]}, 6)$ e cada vértice dessa linha é da forma $(v, e_{v[i]}, j)$ para $1 \leq j \leq 6$ e $1 \leq i \leq gr(v)$. Finalmente, adicionam-se ramos para unir o primeiro e o último vértice de cada um destes caminhos a todos os seleccionadores a_1, a_2, ..., a_K. Estes ramos são:

$$E'' \quad = \quad \{\{a_i, (v, e_v[1], 1)\}, \{a_i, (v, e_v[gr(v)], 6)\} \mid 1 \leq i \leq K, \ v \in V\}$$

O grafo $G' = (V', E')$ é então definido por:

$$\begin{aligned} V' &= \{a_i \mid 1 \leq i \leq K\} \cup (\bigcup_{e \in E} V'_e)\}) \\ E' &= (\bigcup_{e \in E} E'_e) \cup (\bigcup_{v \in V} E'_v) \cup E'' \end{aligned}$$

d) É fácil verificar que G' pode ser construído em tempo polinomial. Vejamos que G' tem um ciclo Hamiltoneano se e só se G tem uma cobertura por vértices de tamanho no máximo K. Suponhamos que $< v_1, v_2, \ldots, v_n >$ onde $n = |V'|$ é um ciclo Hamiltoneano. Considere qualquer porção do ciclo que comece num vértice de $\{a_1, a_2, \ldots, a_K\}$, termine num vértice desse conjunto e que não passe por nenhum outro elemento desse conjunto. Dadas as restrições da forma como um ciclo Hamiltoneano pode passar por uma componente de "teste de cobertura", esta porção do ciclo deve passar por um conjunto dessas componentes correspondentes exactamente aos ramos de E que incidem num vértice particular $v \in V$. Cada componente é atravessada por um dos modos referidos e nenhum vértice de outra componente é encontrado. Então os K vértices de $\{a_1, a_2, \ldots, a_K\}$ dividem o ciclo Hamiltoneano em caminhos, cada um correspondente a um vértice distinto $v \in V$. Dado que o ciclo Hamiltoneano deve incluir todos os vértices de todas as componentes de "teste de cobertura", e como os vértices duma componente para $e \in E$ só podem ser atravessados por um caminho correspondente a uma extremidade de e, todos os ramos de E têm de ter uma extremidade nesse conjunto de K vértices seleccionados. Então, esse conjunto forma uma cobertura por vértices de tamanho K para G.

Inversamente, suponhamos $V^\star \subseteq V$ uma cobertura por vértices de G e $|V^\star| \leq K$. Podemos supor que $|V^\star| = K$, pois podem sempre adicionar-se vértices. Sejam $v_1, \ldots, v_K$ os elementos de $V^\star$. Escolhemos os ramos no ciclo Hamiltoneano de G' da seguinte forma. Da componente de "teste de cobertura" representante de cada $e = \{u, v\} \in E$, escolher os ramos correspondentes a uma das três maneiras, (a), (b) ou (c), como podem ser percorridas, consoante $\{u,v\} \cap V^\star$ é igual a $\{u\}$, $\{u,v\}$ ou $\{v\}$. Note-se que uma dessas condições tem de se verificar porque $V^\star$ é uma cobertura de G. Em seguida escolhe-se todos os ramos em E'_{v_i} para $1 \leq i \leq K$. Finalmente, os ramos:

$$\begin{array}{rl} \{a_i, (v_i, e_{v_i[1]}, 1)\} & 1 \leq i \leq K \\ \{a_{i+1}, (v_i, e_{v_i[gr(v_i)]}, 6)\} & 1 \leq i < K \\ \{a_1, (v_K, e_{v_K[gr(v_i)1]}, 6)\} & \end{array}$$

Deixa-se ao leitor a verificação de que este conjunto de ramos corresponde a um ciclo Hamiltoneano.

□

Exercício 2.3.2 [⋆] Mostre que **3SAT** $\propto$ **3DM** e **3DM** $\propto$ **PAR**. Sugestão: Consulte [GJ79, Cap 3].

Exercício 2.3.3 Mostrar que os seguintes problemas são **NP**-completos:

a) **CLIQUE**

Sugestão: **VC** $\propto$ **CLIQUE**, sabendo que se $G = (V, E)$ é um grafo e $V' \subseteq V$ então V' é uma cobertura por vértices de G sse $V \setminus V'$ é um clique no grafo complementar de G, G^c onde $G^c = (V, E^c)$ e $E^c = \{(u,v) : u, v \in V \text{ e } (u,v) \notin E\}$.

b) **4SAT**

Sugestão: **3SAT** $\propto$ **4SAT**

c) **3COLOR**

Sugestão: **3SAT** $\propto$ **3COLOR**

Uma maneira simples de mostrar que um problema Π é **NP**-completo é obter uma restrição de Π que corresponda a um problema **NP**-completo, isto é, mostrando que Π contém como caso especial um problema **NP**-completo.

Exemplo 8 SUBGRAFO ISOMORFO

Instância: *Dois grafos não dirigidos* $G = (V, E)$ *e* $G' = (V', E')$

Questão: G *contem um subgrafo isomorfo a* G'*, isto é,*

$$\exists V_1 \subseteq V, \exists E_1 \subseteq E \exists f : V' \longrightarrow V_1 \textit{ bijecção, tal que } \forall (a,b) \in E', (f(a), f(b)) \in E_1$$

Se G' *for um grafo completo, este problema tem como restrição o problema do* **CLIQUE**. *Logo é* **NP**-*completo.*

Exercício 2.3.4 Mostrar por restrição que os seguintes problemas são **NP**-completos:

a) **Knapsack** (Problema do "saco de viagem")

Instância: Um conjunto finito U, $s : U \longrightarrow \mathbb{N}$ (tamanho) e $v : U \longrightarrow \mathbb{N}$ (valor), $B \in \mathbb{N}$ (tamanho máximo) e $K \in \mathbb{N}$ (valor objectivo).

Questão: Existe um subconjunto $U' \subset U$ tal que $\sum_{u \in U'} s(u) \leq B$ e $\sum_{u \in U'} v(u) \geq K$?

Sugestão: Restrição a **PAR** com $\forall u \in U, s(u) = v(u)$ e $B = K = 1/2 \sum_{u \in U} s(u)$.

b) **Caminho mais longo**

Instância: Um grafo $G = (V, E)$ e um inteiro positivo $K \leq |V|$

Questão: G tem um caminho de tamanho maior ou igual a K que não repete nenhum vértice?

Sugestão: Restringir a **HC** com $K = |V|$.

c) **(X3C)** Cobertura exacta

Instância: Um conjunto X com $|X| = 3q$ e uma colecção C de subconjuntos de X com 3 elementos.

Questão: É verdade que C contém uma cobertura exacta de X, isto é, uma sub- colecção $C' \subseteq C$ tal que cada elemento de X ocorre num e num só elemento de C'?

Sugestão: Restringir a **3DM** considerando M um conjunto desordenado $W \cup X \cup Z$.

d) **Conjuntos Disjuntos**

Instância: Dada a colecção C de conjuntos finitos e um inteiro positivo, $K \leq |C|$.

Questão: C contém K conjuntos disjuntos?

A classe dos problemas **NP**-completos inclui muitos problemas para os quais se fizeram muitos esforços para encontrar algoritmos polinomiais. O facto de que se um deles tiver um algoritmo polinomial todos os outros têm, torna-os os "mais difíceis" da classe **NP** e reforça a conjectura de que eles não pertencem classe **P** e portanto $\mathbf{P} \neq \mathbf{NP}$. Assim se se mostra que um novo problema é **NP**-completo não vale muito a pena procurar um algoritmo polinomial para ele mas sim tentar resolvê-lo por outros processos!

Ao tentar provar que um dado problema é **NP**-completo duas coisas podem acontecer:

a) não se conseguir provar que pertence a **NP** (um caso trivial é se o problema não é de decisão).

b) não se conseguir provar que é completo

No primeiro caso podemos ainda conseguir provar que é completo, mostrando que esse problema se pode transformar polinomialmente num problema **NP**-completo - e sendo assim só ser resolvido em tempo polinomial se $\mathbf{P} = \mathbf{NP}$. Neste caso o problema diz-se **NP-hard** (é pelo menos tão difícil como os **NP**-completos).

No segundo caso se se mostrar que é **NP** (mas não **P**) então é um candidato da classe $\mathbf{NP} - \mathbf{P} - \mathbf{NPC}$ que chamaremos **NPI** – problemas de dificuldade "intermédia". Teoricamente se $\mathbf{P} \neq \mathbf{NP}$ mostra-se que esta classe é não vazia e que existe uma infinidade de classes de equivalência entre **P** e **NP**.

2.4 Redução de Turing e Problemas NP-hard

Vamos definir uma forma mais geral de redutibilidade polinomial, que pode ser aplicada a uma classe mais geral de problemas: **problemas de procura** em que para cada instância `I` , a resposta (em vez de ser apenas "sim" ou "não") pode ser um elemento dum conjunto finito de soluções ou "não" (se o dito conjunto for vazio). Repare que esta classe contém os problemas de decisão e os problemas de optimização.

Um problema de procura Π consiste num conjunto D_Π de instâncias, e para cada $\mathtt{I} \in D_\Pi$ num conjunto finito $S_\Pi[\mathtt{I}]$ de soluções de `I` . O problema Π tem solução para a instância `I` se $S_\Pi[\mathtt{I}] \neq \emptyset$. Um algoritmo resolve o problema Π , se dado uma instância `I` tem como resposta "não" se $S_\Pi[\mathtt{I}] \neq \emptyset$ ou se tem como saída uma solução s pertencente a $S_\Pi[\mathtt{I}]$.

Por exemplo, o problema do caixeiro viajante tem como soluções todas as voltas de comprimento mínimo, e um algoritmo resolve o problema se para cada instância indicar uma das possíveis soluções.

Formalmente, dado um alfabeto um problema de procura vai corresponder a uma relação binária em Σ^+. A uma relação binária R em Σ^+ pode-se associar uma família de funções $f : \Sigma^+ \longrightarrow \Sigma^+$ tal que para todo $x \in \Sigma^+$, $f(x) = \epsilon$ se não existe nenhum $y \in \Sigma^+$ tal que $(x, y) \in R$ ou $f(x) = y$ se tal y existe. Diz-se que f realiza R. Uma TM M resolve R se a função f_M calculada por M realiza R.

Dado um problema de procura Π e uma codificação c sobre $\Sigma^\star$ associámos a relação $R[\Pi, c]$ definida por:

$$R[\Pi, c] \quad = \quad \{(x, y) \mid x = c(\mathtt{I}),\ \mathtt{I} \in D_\Pi \text{ e } y = c(s),\ s \in S_\Pi[\mathtt{I}]\}$$

O problema Π com codificação c é solúvel em tempo polinomial se existe uma TM polinomial que resolve $R[\Pi, c]$.

Exercício 2.4.1 Identifique uma linguagem $L \subseteq \Sigma^+$ com uma relação binária em Σ^+. Conclua que um problema de decisão é um problema de procura.

Em termos de problemas, uma redução polinomial de Turing dum problema Π num problema Π' é um algoritmo determinístico `A` que resolve Π usando uma subrotina `S` que resolve Π', e tal que se `S` é um algoritmo polinomial para Π' então `A` é um algoritmo polinomial para Π . Diz-se que Π *é Turing redutível a* Π', e representa-se por $\Pi \ \propto_t \ \Pi'$.

Em termos de linguagens corresponde a existir uma máquina de Turing determinística com oráculo Π' que reconhece Π em tempo polinomial.

Definição 2.4.1 *Uma máquina de Turing com oráculo (OTM) corresponde a uma máquina de Turing básica* $O = (S, \Gamma, \Sigma, b, s_0, \delta, F)$ *com as seguintes características adicionais:*

- *possuí uma fita adicional – a fita de oráculo – e uma cabeça se escrita/leitura que opera nessa fita.*
- *o conjunto de estados* S *inclui dois estados especiais* s_c*, estado de consulta do oráculo e* s_r*, estado de continuação da computação.*
- *A função de transição é* $\delta : S \setminus F \cup \{s_c\} \times \Gamma \times \Sigma \longrightarrow S \times \Gamma \times \Sigma \times \{l, r\} \times \{l, r\}$

Um passo de computação duma OTM é análogo ao de uma TM básica (considerando as duas fitas) excepto quando o controlo finito se encontra no estado s_c. Neste estado, a computação depende duma função de oráculo $g : \Sigma^\star \longrightarrow \Sigma^\star$. Seja $y \in \Sigma^\star$ a sequência de símbolos nas células de 1 a $|y|$ da fita de oráculo e seja $g(y) = z$. então, *num* passo de computação a fita de oráculo é modificada para conter a sequência $z \in \Sigma^\star$ nas células de 1 a $|z|$ e brancos nas restantes células. A cabeça de oráculo fica a ler a célula 1 e os estado passa a ser s_r. A fita normal e sua cabeça não são alteradas neste passo.

Exercício 2.4.2 Descreva formalmente um passo de computação duma OTM.

Uma OTM O com função de oráculo g associada designa-se por O_g. As noções de computação, função calculada f_O^g e complexidade $T_{O_g}(n)$ são definidas de modo idêntico ao das TM básicas.

Definição 2.4.2 *Sejam R e R' duas relações binárias em $\Sigma^\star$. Uma* redução polinomial de Turing *de R em R', $R \propto_t R'$, é uma OTM O tal que para toda a função $g : \Sigma^\star \longrightarrow \Sigma^\star$ que realiza R', O_g é uma OTM polinomial e a função f_O^g calculada por O_g realiza R.*

Proposição 2.4.1 *A relação $\propto_t$ é transitiva.*

Exercício 2.4.3 Mostre a proposição 2.4.1.

Proposição 2.4.2 *Se Π e Π' são problemas de decisão então:*

$$\Pi \propto \Pi' \Longrightarrow \Pi \propto_t \Pi'$$

Dem. A transformação F da redução polinomial $\propto$ define um algoritmo para resolver Π usando uma subrotina para resolver Π': dada uma instância de Π , constrói uma instância $F(X)$ de Π', aplica a subrotina a $F(X)$ e tem como resultado a resposta dada pela subrotina (pois X tem solução sse $F(X)$ tem solução). □

Exercício 2.4.4 Enuncie e demonstre a proposição 2.4.2 em termos de linguagens, relações e máquinas de Turing com oráculo.

Definição 2.4.3 *Uma relação R é* **NP-hard** *se existe uma linguagem $L \in$* **NPC** *tal que $L \propto_t R$. Um problema de procura Π diz-se* **NP-hard** *se existe um problema* **NP***-completo Π' tal que $\Pi' \propto_t \Pi$.*

Em particular, um problema de decisão Π é **NP-hard** se $\forall \Pi' \in$ **NP**, $\Pi' \propto_t \Pi$ e é **NP**-completo se além disso $\Pi \in$ **NP**.

Dado um problema de decisão Π o seu complementar designa-se por Π^c e é tal que $S_{\Pi^c} = D_\Pi \setminus S_\Pi$.

Exercício 2.4.5 Mostrar que dado um problema $\Pi \in$ **NP** e o seu problema complementar Π^c, se tem $\Pi \propto_t \Pi^c$ e vice-versa. Explique porque não parece para muitos problemas que se possa ter $\Pi^c \propto \Pi$.

Pelo exercício anterior concluímos que se Π é **NP**-completo ou **NP-hard** então Π^c é **NP-hard**.

Vimos que um problema de decisão D não é "mais difícil" que o correspondente de optimização O, vejamos que muitas vezes também não é "mais fácil"! Isto é não só $D \propto_t O$ mas também $O \propto_t D$. Se o problema D é **NP**-completo então o problema de optimização O pode ser resolvido em tempo polinomial se e só se $\mathbf{P} = \mathbf{NP}$.

Exemplo 9 *Consideremos o problema do "caixeiro viajante" mais uma vez... Comecemos por definir um problema intermédio:*

(TSE)

Instância: *Um conjunto* $C = \{c_1, c_2, \ldots, c_m\}$ *de cidades,* $d : C \times C \longrightarrow \mathbb{N}$ *distância,* $B \in \mathbb{N}$ *e uma volta "parcial"* $\theta =< c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(k)} >$ *de* k *cidades distintas,* $1 \leq k \leq m$.

Questão: *Pode-se estender* Θ *a uma volta completa* $< c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(k)}, c_{\pi(k+1)}, \ldots c_{\pi(m)} >$ *com comprimento total menor ou igual a* B*?*

Este problema é **NP** *(mostre!) e como* **TSP** *é completo, tem-se que* **TSE** $\propto_t$ **TSP**. *Seja* **(TSO)** *o problema de optimização, resta ver que* **TSO** $\propto_t$ **TSE** *e por transitividade vem* **TSO** $\propto_t$ **TSP**.

Considerem-se instâncias de **TSE** *e* **TSO** *para* C *e* d *comuns. Suponhamos que* $\mathtt{S(C, d, \Theta, B)}$ *é uma subrotina que resolve* **TSE**. *Seja* $B^\star$ *a volta óptima para essa instância de* **TSO** *e suponhamos que essa volta óptima começa em* c_1. *É óbvio que* $B^\star$ *tem como limite inferior* $B_{MIN} = m$ *e como limite superior* $B_{MAX} = mx$ *onde* $x = max\{d(c_i, c_j) \mid (c_i, c_j) \in C \times C\}$. *Podemos então usar uma pesquisa binária para determinar* $B^\star$ *que chama* $\mathtt{S(C, d, < c_1 >, B)}$ *para vários valores de* B*, no máximo* $\lceil \log_2 B \rceil$ *vezes:*

```
B_MIN <-  m; B_MAX  <- mx
enquanto  B_MAX  -  B_MIN <> 1 faça{
        B<- [(B_MAX    + B_MIN   )/2 ];
        se S(C,d,<c >,B) = "sim" então B_MAX<- B senão B_MIN <- B
}
        B*  <- B_MIN
```

Conhecido $B^\star$ *determina-se uma volta óptima usando a subrotina* $\mathtt{S}$. *Para isso constroem-se sequências* Θ *que possam ser estendidas a voltas óptimas. Dado* $< c_1 >$ *e como é estendível, existe* $c_j \in C \backslash \{c_1\}$ *tal que* $< c_1, c_j >$ *é uma volta parcial estendível. Podemos encontrar* c_j *no máximo em* $m-2$ *chamadas a* $\mathtt{S[C, d, < c_1, c_i >, B^\star]}$. *Então uma volta óptima pode ser construída usando* $\mathtt{S}$ *no máximo* $(m-1)(m-2)/2$ *vezes. Considerando o comprimento dos dados,* $n = m + \lceil \log_2 B_{MAX} \rceil$*, o algoritmo anterior fornece uma redução polinomial de Turing entre* **TSO** *e* **TSE**, *como pretendido.*

Note-se que basta provar que um dado problema é Turing–redutível a um problema em **NP** para saber que ele não é "mais difícil" que um problema **NP**-completo.

Definição 2.4.4 *Um problema* Π *é* **NP-easy** *se existe um problema* $\Pi' \in$ **NP** *tal que* $\Pi \propto_t \Pi'$.

Exercício 2.4.6 Mostrar que os problemas de optimização (ou de procura de solução) associados com os seguintes problemas **NP**-completos são **NP-easy**:

a) **SAT**

b) **VC**

c) **HC**

Concluímos assim que a restrição feita aos problemas de decisão na teoria apresentada não provocou perda de generalidade, pois a maior parte dos problemas cujos correspondentes de decisão são NP-completos, são **NP-easy** e portanto só serão resolvidos em tempo polinomial se **P** = **NP** !

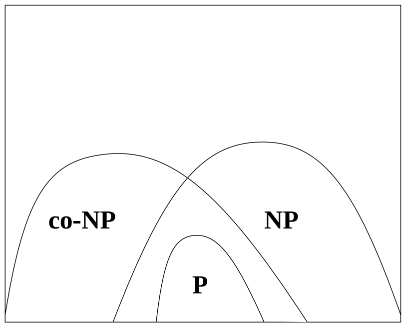


Figura 2.3: **NP**e **coNP**

2.5 Estrutura das Classes de Problemas

2.5.1 Classe coNP e Estrutura de NP

A classe dos problemas **P** é fechada em relação complementação. Dado um problema de decisão Π , o conjunto de soluções do problema complementar, Π^c, corresponde às soluções cuja resposta para o problema Π é "não". Se Π tem um algoritmo determinístico polinomial, um algoritmo para Π^c obtém-se trocando as respostas "sim" por "não" (continuando a ser determinístico e polinomial). Como já se viu, o mesmo não acontece com a classe dos problemas **NP**. Nem sempre se pode provar que se $\Pi \in \mathbf{NP}$, então $\Pi^c \in \mathbf{NP}$. Por exemplo, o complementar do problema de decisão do "caixeiro viajante" **(TSP)** pode-se formular nos seguintes termos: Dado um conjunto de cidades, a distância inter-cidades e um limite B, é verdade que não existe *nenhum* circuito por todas as cidades com comprimento menor ou igual a B? A resposta a esta questão necessita da verificação de todos (ou quase todos) os circuitos correctos, o que não parece ser possível usando um algoritmo não-determinístico em tempo polinomial!

Em termos formais, não parece que este problema tenha para cada instância, uma *testemunha concisa*. Seja a classe

$$\mathbf{coNP} \quad = \quad \{\Pi^c \mid \Pi \in \mathbf{NP}\}$$

Ou em termos de linguagens, dado um alfabeto Σ:

$$\mathbf{coNP} \quad = \quad \{\Sigma^\star \setminus L \mid L \subseteq \Sigma \text{ e } L \in \mathbf{NP}\}$$

Dado que existem muitos problemas em **coNP**que parecem não estar em **NP**, podemos conjecturar que $\mathbf{NP} \neq \mathbf{coNP}$. Note-se que se esta conjectura for verdadeira implica que $\mathbf{P} \neq \mathbf{NP}$. Porquê? Como se viu $\mathbf{P} \in \mathbf{coNP} \cap \mathbf{NP}$. Na figura 2.3 tem-se a relação entre estas classes.

O resultado seguinte relaciona os problemas **NP**-completos com esta conjectura.

Proposição 2.5.1 *Se existe um problema* **NP**-*completo* Π , *tal que* $\Pi^c \in \mathbf{NP}$, *então* $\mathbf{NP} = \mathbf{coNP}$.

Exercício 2.5.1 Mostre a proposição 2.5.1.

Se $\mathbf{P} \neq \mathbf{NP}$ e $\mathbf{NP} \neq \mathbf{coNP}$ a estrutura para **NP** é a apresentada na figura 2.4.

Exercício 2.5.2 Considere o problema complementar de **3SAT**:

$\mathbf{3SAT}^c$

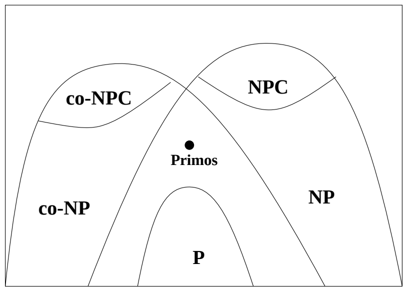


Figura 2.4: Estrutura de **NP**

Instância: Um conjunto U de variáveis lógicas e um conjunto C de cláusulas, cada uma com 3 literais.

Questão: Não existe nenhuma atribuição de valores de verdade às variáveis que satisfaça C?

Mostre que se tivéssemos um algoritmo polinomial para decidir este problema, todos os problemas de **NP** poderiam ser decididos polinomialmente. Comente em relação completitude desse problema.

O resultado da proposição 2.5.1 permite ainda encontrar um forte candidato classe **NPI** (ou então a **P**): o problema de determinar se um dado inteiro é composto.

(COMPOSTO)

Instância: Um número inteiro $K > 1$

Questão: Existem inteiros m, $n > 1$ tais que $K = mn$?

O seu problema complementar é

(PRIMOS)

Instância: Um número inteiro K

Questão: K é primo?

Tanto o problema **(COMPOSTO)** como o complementar, o problema (**PRIMOS**) pertencem a **NP**.

Nenhum destes problemas pode ser **NP**-completo a menos que **NP** = **coNP**. Assim, se não existir nenhum algoritmo polinomial que os resolva, pertencem a **NPI**(=**NP** − **P** − **NPC**).

Mais ainda, estes problemas podem ser resolvidos por algoritmos polinomiais, mas que podem falhar com probabilidade arbitrariamente pequena... Nestas circunstâncias podemos conjecturar que ¶ = $NP \cap$ **coNP**! Um problema que "contribuiu" para esta conjectura foi o da "Programação Linear"(LP) que durante muitos anos esteve nas condições do problema dos números compostos mas para o qual já se provou a existência dum algoritmo polinomial...

Exercício 2.5.3 Mostre que (**COMPOSTO**) $\in$ **NP**.

Para uma demonstração que (**PRIMOS**) pertence a **NP**, ver [Pap94, Cap. 10] ou [Pra75].

2.5.2 Hierarquia polinomial

Nesta secção vamos estudar problemas de decisão que são **NP-hard** mas que parecem não ser **NP-easy**.

Podemos generalizar a noção de redução de Turing permitindo que o algoritmo usado seja não-determinístico, o que equivale, em termos de linguagens, a considerar máquinas de Turing não-determinísticas com oráculo. Temos por uma lado uma escolha e por outro uma consulta ao oráculo. Dados dois problemas Π e Π', se existir um algoritmo não-determinístico polinomial que resolva Π usando uma subrotina que resolva Π', indica-se $\Pi \propto_t^N \Pi'$.

Vamos ver um problema que é **NP-hard** mas que não parece ser **NP-easy** e para o qual se pode encontrar uma redução $\propto_t^N$.

Exemplo 10 (EME)

Instância: *Uma expressão booleana E com literais num conjunto de variáveis U, constantes V e F e conectivas lógicas $\vee$, $\wedge$, $\neg$ e $\Longrightarrow$. E ainda, $K \in \mathbb{N}$.*

Questão: *Existe uma expressão booleana E' com K ou menos literais e tal que E' é equivalente a E ?*

Este problema é **NP-hard** *porque* **SAT** $\propto_t$ **EME** *(Verifique!). No entanto, não parece fácil mostrar que ele é Turing redutível a um problema de* **NP**. *Contudo, usando um algoritmo não-determinístico com um "oráculo" podemos reduzir este problema ao problema da "satisfação de expressões booleanas":*

(SBE)

Instância: *Uma expressão booleana E com literais num conjunto de variáveis U, constantes V e F e conectivas lógicas $\vee$, $\wedge$, $\neg$ e $\Longrightarrow$.*

Questão: *Existe uma atribuição de valores de verdade a U que satisfaz E?*

Este problema contém em particular o problema **SAT**, *portanto é* **NP***-completo.*

Seja uma instância de **EME** *com dados U, E e K, suponha-se B_K o conjunto das expressões booleanas com K ou menos literais e seja, ainda,* `S[E,U]` *uma subrotina que resolve* **SBE**. *O seguinte algoritmo estabelece a redução,* **EME** $\propto_t^N$ **SBE**:

```
E´<- escolha(B_K)
se S[~((E=>E´)/\(E´=>E)),U] = "não" então sucesso
        senão insucesso
```

Concluímos então que **EME** *pertence a uma classe mais ampla de problemas aparentemente "mais difíceis" do que os da classe* **NP**.

As classes $\mathbf{P^Y}$ e $\mathbf{NP^Y}$ são definidas do modo seguinte:

$$\begin{aligned} \mathbf{P^Y} &= \{L \mid \exists L' \in \mathbf{Y} \text{ e } L \propto_t L'\} \\ \mathbf{NP^Y} &= \{L \mid \exists L' \in \mathbf{Y} \text{ e } L \propto_t^N L'\} \end{aligned}$$

$\mathbf{P^{NP}}$ é a classe dos problemas a que chamamos anteriormente **NP-easy**. O problema **EME** pertence a $\mathbf{NP^{NP}}$.

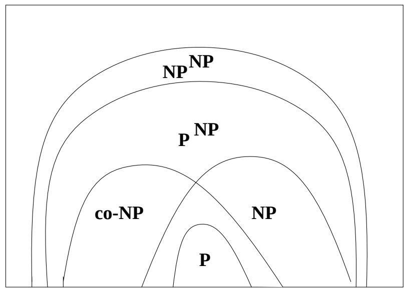


Figura 2.5: Classes $\mathbf{P}$, $\mathbf{NP}$, $\mathbf{coNP}$, $\mathbf{P}^{\mathbf{NP}}$e $\mathbf{NP}^{\mathbf{NP}}$

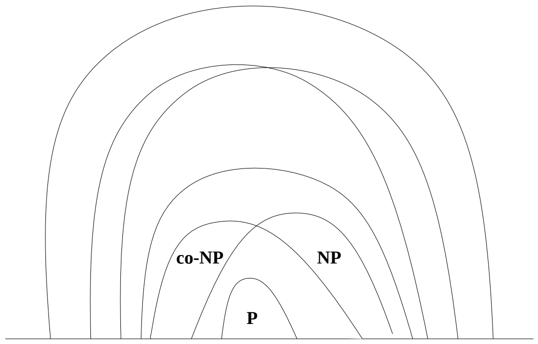


Figura 2.6: Hierarquia Polinomial

Exercício 2.5.4 (i) Mostre que o seguinte problema é $\mathbf{coNP}$.

EE

Instância: Duas expressões booleanas E e E' com literais num conjunto de variáveis U, constantes V e F e conectivas lógicas $\vee$, $\wedge$, $\neg$ e $\Longrightarrow$.

Questão: E e E' são equivalentes?

(ii) Conclua, novamente, que $\mathbf{EME} \in \mathbf{NP}^{\mathbf{NP}}$ usando a alínea anterior.

Se $\mathbf{P} \neq \mathbf{NP}$ a figura 2.5 mostra a relação entre as classes $\mathbf{P}$, $\mathbf{NP}$, $\mathbf{coNP}$, $\mathbf{P}^{\mathbf{NP}}$ e $\mathbf{NP}^{\mathbf{NP}}$. Este processo de definir novas classes pode ser estendido indefinidamente, produzindo classes de dificuldade aparentemente crescente, [MS72]. Esta hierarquia de classes chama-se *hierarquia polinomial* e é definida do seguinte modo:

$$\begin{array}{lcllll}
\Sigma^{\mathbf{P}}_{\mathbf{0}} & = & \Pi^{\mathbf{P}}_{\mathbf{0}} & = & \Delta^{\mathbf{P}}_{\mathbf{0}} & = \mathbf{P} \\
\Delta^{\mathbf{P}}_{\mathbf{k+1}} & = & \mathbf{P}^{\Sigma^{\mathbf{P}}_{\mathbf{k}}} & & & \\
\Sigma^{\mathbf{P}}_{\mathbf{k+1}} & = & \mathbf{NP}^{\Sigma^{\mathbf{P}}_{\mathbf{k}}} & & & \\
\Pi^{\mathbf{P}}_{\mathbf{k+1}} & = & \mathbf{co}\Sigma^{\mathbf{P}}_{\mathbf{k+1}} & & &
\end{array}$$

Em particular, $\Sigma^{\mathbf{P}}_{\mathbf{1}} = \mathbf{NP}^{\mathbf{P}} = \mathbf{NP}$, $\Pi^{\mathbf{P}}_{\mathbf{1}} = \mathbf{coNP}$ e $\Delta^{\mathbf{P}}_{\mathbf{1}} = \mathbf{P}$. Como se viu $\mathbf{EME}$ pertence a $\Sigma^{\mathbf{P}}_{\mathbf{2}}$, embora não se saiba se $\Sigma^{\mathbf{P}}_{\mathbf{1}} = \Sigma^{\mathbf{P}}_{\mathbf{2}}$, ou se existe $k \geq 0$ tal que $\Sigma^{\mathbf{P}}_{\mathbf{k}} \neq \Sigma^{\mathbf{P}}_{\mathbf{k+1}}$. A relação de inclusão entre as várias classes é dada pelo diagrama da figura 2.6.

Para um dado problema de decisão Π , como situá-lo na hierarquia? Vamos apresentar um resultado que permite determinar um majorante do menor k tal que $\Pi \in \Sigma^{\mathbf{P}}_{\mathbf{k}}$.

Recordemos que um problema está em **NP**se tiver uma "testemunha concisa". Esta noção pode ser estendida para qualquer problema de decisão Π , isto é, qualquer linguagem L.

Dado um alfabeto Σ uma relação k-ária em $\Sigma^\star$ é reconhecida em tempo polinomial se existir uma TM que reconhece precisamente os tuplos $(x_1,\ldots,x_k) \in R$, isto é, tal que $R(x_1,\ldots,x_k)$ se verifica.

Podemos então dizer que:

- $L \in \mathbf{P}$ se e só se existe $R_1 \in \Sigma^\star$ reconhecível polinomialmente tal que $L = \{x \mid R_1(x)\}$
- $L \in \mathbf{NP}$ se e só se existe $R_2 \in \Sigma^\star \times \Sigma^\star$ reconhecível polinomialmente e uma função polinomial p tal que $L = \{x \mid \exists y, |y| \leq p(|x|) \text{ e } R_2(x,y)\}$
- $L \in \mathbf{coNP}$ se e só se existe $R_2^c \in \Sigma^\star \times \Sigma^\star$ reconhecível polinomialmente e uma função polinomial p tal que $L = \{x \mid \forall y, |y| \leq p(|x|) \text{ e } R_2^c(x,y)\}$

O teorema seguinte é de [Wra76]:

Teorema 2.5.1 *Seja $L \subseteq \Gamma^\star$ uma linguagem sobre um alfabeto Γ e($|\Gamma| \geq 2$). Para todo $k \geq 1$ $L \in \Sigma_\mathbf{k}^\mathbf{P}$ se e só se existem funções polinomiais $p_1, \ldots, p_k$ e uma relação R reconhecida em tempo polinomial tal que:*

$$\begin{array}{lll} x \in L & \textit{sse} & \exists y_1 \in \Gamma^\star |y_1| \leq p_1(|x|) \\ & & \forall y_2 \in \Gamma^\star |y_2| \leq p_2(|x|) \\ & & \ldots \\ & & Q y_k \in \Gamma^\star |y_k| \leq p_k(|x|) \\ & & R(x, y_1, \ldots, y_k) \end{array}$$

onde Q em y_k é $\exists$ ou $\forall$ consoante k é ímpar ou par.

Usando o teorema 2.5.1 para provar que $\mathbf{EME} \in \Sigma_\mathbf{2}^\mathbf{P}$, considere-se a relação R, com $R(x, y_1, y_2)$ se e só se x é $< E, K >$ (instância) onde E é uma expressão booleana e $K \in \mathbb{N}$, y_1 é uma expressão booleana E' com no máximo K literais, e y_2 é uma atribuição de valores às variáveis $t : U \longrightarrow \{0,1\}$ que satisfaz $E \Leftrightarrow E'$.

As classes $\Pi_\mathbf{k}^\mathbf{P}$ também podem ser caracterizadas de modo análogo, bastando no teorema 2.5.1 tocar todos os quantificadores $\exists$ por $\forall$ e vice-versa.

Corolário 2.5.1 *Seja $L \subseteq \Gamma^\star$ uma linguagem sobre um alfabeto Γ e ($|\Gamma| \geq 2$) e $i \geq 1$. $L \in \Sigma_\mathbf{k}^\mathbf{P}$ se e só se existe uma relação reconhecida em tempo polinomial tal que a linguagem $\{< x, y >\mid R(x,y)\} \in \Pi_\mathbf{k-1}^\mathbf{P}$ e existe p tal que $L = \{x \mid \exists y,\ |y| \leq p(|x|) \text{ e } R(x,y)\}$.*

Corolário 2.5.2 *Seja $L \subseteq \Gamma^\star$ uma linguagem sobre um alfabeto Γ e ($|\Gamma| \geq 2$) e $i \geq 1$. $L \in \Pi_\mathbf{k}^\mathbf{P}$ se e só se existe uma relação reconhecida em tempo polinomial tal que a linguagem $\{< x, y >\mid R(x,y)\} \in \Sigma_\mathbf{k-1}^\mathbf{P}$ e existe p tal que $L = \{x \mid \forall y,\ |y| \leq p(|x|) \text{ e } R(x,y)\}$.*

A noção de completitude pode ser definida para cada k e para cada um dos tipos de classes da hierarquia, caracterizando os problemas "mais difíceis" de cada classe. Para cada classe é possível construir um problema $\Sigma_\mathbf{k}^\mathbf{P}$-completo e portanto torna-se mais fácil provar completitude de outros problemas.

Na prática só se conhecem alguns problemas (além dos construídos teoricamente) candidatos a $\Sigma_\mathbf{2}^\mathbf{P} \setminus (\Sigma_\mathbf{1}^\mathbf{P} \cup \Sigma_\mathbf{1}^\mathbf{P})$ ou mesmo $\Sigma_\mathbf{2}^\mathbf{P}$-completos.

Um resultado interessante em relação a esta hierarquia é o seguinte:

Proposição 2.5.2 *Se para algum k, $\Sigma_{\mathbf{k}}^{\mathbf{P}} = \Pi_{\mathbf{k}}^{\mathbf{P}}$ então $\Sigma_{\mathbf{j}}^{\mathbf{P}} = \Pi_{\mathbf{j}}^{\mathbf{P}} = \Sigma_{\mathbf{k}}^{\mathbf{P}}$ para todo o $j \geq k$.*

Exercício 2.5.5 Mostrar a proposição 2.5.2. Sugestão: Mostre que $\Sigma_{\mathbf{k}}^{\mathbf{P}} = \Pi_{\mathbf{k}}^{\mathbf{P}}$ implica $\Sigma_{\mathbf{k+1}}^{\mathbf{P}} = \Pi_{\mathbf{k}}^{\mathbf{P}}$.

Isto é, a hierarquia pode colapsar a partir de certa ordem e portanto não ser infinita. Em particular, se $\mathbf{P} = \mathbf{NP}$, então $\Sigma_{\mathbf{j}}^{\mathbf{P}} = \mathbf{P}$, para todo $j \geq 0$. Se $\Sigma_{\mathbf{0}}^{\mathbf{P}} \neq \Sigma_{\mathbf{k}}^{\mathbf{P}}$ para algum $k > 0$, então $\mathbf{P} \neq \mathbf{NP}$. Temos então que qualquer problema da hierarquia só pode ser resolvido em tempo polinomial se e só se $\mathbf{P} = \mathbf{NP}$ (e portanto embora possa não ser **NP-easy** na prática tem as mesmas consequências!). Contudo, o estudo destas classes pode ajudar a estabelecer a conjectura de que $\mathbf{P} \neq \mathbf{NP}$.

2.6 Exercícios de Revisão sobre Classes de Complexidade

2.6.1 Ordens de Grandeza

Nesta secção são revistas algumas noções sobre ordens de grandeza de funções (reais).

Definição 2.6.1 *Sendo* $f : \mathbb{N} \to \mathbb{R}$ *e* $g : \mathbb{N} \to \mathbb{R}$

1. $f(n)$ *é* $O(g(n))$ *se*
$$\exists c \in \mathbb{R}^{+} \exists m \forall n \geq m \mid f(n) \mid \leq c \mid g(n) \mid$$
(diz-se que f *é da ordem de* g*, isto é,* f *cresce como* g *ou mais lentamente)*

2. $f(n)$ *é* $\Omega(g(n))$ *se* $g(n)$ *é* $O(f(n))$

3. $f(n)$ *é* $\Theta(g(n))$ *se* $f(n)$ *é* $O(g(n))$ *e* $g(n)$ *é* $O(f(n))$

4. $f(n)$ *é* $o(g(n))$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$*, isto é,* $g(n)$ *cresce mais rapidamente do que* $f(n)$.

Se $f(n)$ é $O(p(n))$ e $p(n)$ é uma função polinomial então diz-se que $f(n)$ é de ordem polinomial.

Teorema 2.6.1 *Se* $f(n)$ *é* $o(g(n))$ *então* $f(n)$ *é* $O(g(n))$.

Exercício 2.6.1 Mostre o teorema anterior.

Exercício 2.6.2 Mostre as seguintes igualdades:

(i) $f(n) = O(f(n))$

(ii) $c.O(f(n)) = O(f(n))$ para qualquer c constante

(iii) $O(f(n) + g(n)) = O(f(n)) + O(g(n))$

(iv) $O(O(f(n))) = O(f(n))$

(v) $O(f(n))O(g(n)) = O(f(n)g(n))$

(vi) $O(f(n))g(n) = f(n)O(g(n))$

Exercício 2.6.3 Mostre que:

(i) $a_m n^m + \ldots + a_0$ é $\Theta(n^m)$, para $a_m > 0$. Sugestão: Para mostrar que é $\Omega(n^m)$ considere $c = \frac{2}{a_m}$ e $n_0 = mh$ com $h = (2/a_m)max(|a_{m-1}|, \ldots, |a_0|)$

(ii) n^α é $o(n^\beta)$, se $0 \leq \alpha < \beta$.

(iii) n^α é $o(\beta^n)$, se $\beta > 1$. Conclua que para qualquer polinómio $p(n)$ e $\beta > 1$, $p(n)$ é $O(\beta^n)$.Sugestão: Considere $\beta = (1 + \epsilon)$ e a expansão do binómio de Newton.

(iv) para qualquer polinómlio $p(n)$ e constante c existe um inteiro n_0 tal que, $\forall\, n > n_0$, $2^{cn} > p(n)$. Calcule esse n_0 para n^2 e $c = 1$ e para $100n^{100}$ e $c = \frac{1}{100}$. Conclua que $p(n)$ não é $\Omega(c^n)$, $c > 1$

(v) $\log n$ é $O(n)$ e $o(n^\alpha)$, $\alpha > 0$

(vi) $\forall a, b > 1$, $\log_a n$ é $\Theta(\log_b)$. Sugestão: Recorde que para $n > 0$, $\log_a n$ é y tal que $a^y = n$.

(vii) $\forall \alpha > 1$, α^n é $o(n!)$

Exercício 2.6.4 Mostre que Θ é uma relação de equivalência.

Exercício 2.6.5 Sendo $f(n)$ e $g(n)$ qualquer par de funções a seguir apresentadas, determine se $f(n)$ é $O(g(n))$, $f(n)$ é $\Omega(g(n))$ ou $f(n)$ é $\Theta(g(n))$

$$n^2 \quad n^3 \quad n^{\log n} \quad 2^n \quad n^n \quad 2^{2^n}$$

2.6.2 Codificações Razoáveis

Definição 2.6.2 *Duas codificações c_1 e c_2, dizem-se* polinomialmente relacionadas*, se existem polinómios p e p' tal que, sendo x_1 e x_2 as representações dum mesmo objecto x, respectivamente em c_1 e c_2, se tem $\mid x_1 \mid \leq p(\mid x_2 \mid)$ e $\mid x_2 \mid \leq p'(\mid x_1 \mid)$.*

Exercício 2.6.6 Considere o problema de codificar um inteiro não-negativo m numa base $b \geq 2$. Mostre que

(i) os comprimentos das codificações em bases diferentes estão polinomialmente (até linearmente, $O(n)$) relacionadas entre si.

(ii) a codificação em unário não está polinomialmente relacionada com a codificação numa base $b \geq 2$

Exercício 2.6.7 Considere o problema de codificar um grafo não dirigido $G = (V, E)$. Dois processos são:

- matriz de adjacências representada por $\mid V \mid^2$ bits
- lista de ramos: a cad vertice associar a lista dos pontos de chegada dos ramos que dele partem

(i) Defina mais rigorosamente as codificações mencionadas

(ii) Dê exemplo de um grafo codificado das duas formas

(iii) Mostre que os comprimentos das duas codificações estão polinomialmente (quadraticamente) relacionados entre si

Exercício 2.6.8 Supondo o alfabeto $\Sigma = \{0, 1, -, [,], (,), , \}$ invente uma codificação genérica para representar

- um inteiro em binário
- a referência ao n-ésimo elemento dum conjunto, sequência, etc...
- uma sequência de elementos

Usando as coficações anteriores diga como representar: um conjunto (finito) de objectos, um grafo, uma função finita e um número racional.

2.6.3 Classes de Complexidade e NP-completitude

Exercício 2.6.9 Caminhos Hamiltonianos
Sejam respectivamente **HPN** e **HPE** os problemas de decisão "existência de um caminho hamiltoniano qualquer (entre vértices distintos)" e "existência de um caminho hamiltoniano entre dois vértices (distintos) especificados". Note que, no primeiro caso, uma instância é um grafo não dirigido $G = (V, E)$ enquanto no segundo caso uma instância é da forma $G = (V, E)/x/y$ onde G é um grafo não dirigido e $x, y \in V$ são os vértices especificados.

1. Mostra que HPE $\propto$ HPN
2. Mostra que HPN $\propto$ HPE

Exercício 2.6.10 Colorir grafos
Considera os seguintes problemas de decisão a que chamamos $C(n)$ (onde, para cada problema n é fixo):

(n-COLOR)

Instância: Um grafo dirigido $G = (V, E)$.

Questão: É possível colorir G com n cores, isto é, existe uma função

$$f : V \longrightarrow \{1, \cdots, n\}$$

tal que sempre que $(a, b) \in E$ é $f(a) \neq f(b)$?

(i) Mostra que $C(1)$ e $C(2)$ pertencem à classe P.

(ii) Supondo que se sabe que $C(3)$ é **NP**-completo mostra que $C(4)$ também é **NP**-completo. Um método consiste em mostrar que $C(4)$ é **NP**e que $C(3) \propto C(4)$.

(iii) Supondo que se sabe que $C(3)$ é **NP**-completo mostra que $C(n)$ é **NP**-completo para todo o $n \geq 3$.

Exercício 2.6.11 Acabas de demonstrar que um determinado problema Π é **NP**-completo. Qual o significado *prático* dessa descoberta? Podemos concluir que todos os algoritmos para resolver Π são quase inúteis uma vez que vão demorar um tempo exponencial?

Exercício 2.6.12 *Caminhos entre 2 vértices de um grafo*
Considera um grafo não dirigido $G(V, E)$ e dois vértices fixos a e b de V.

1. Mostra que o número de caminhos simples e distintos entre a e b pode ser exponencial no tamanho do grafo — por exemplo em $max(|V|, |E|)$. Nota: um caminho diz-se *simples* quando não passa mais que uma vez por cada nó ("não tem ciclos"). Dois caminhos são distintos quando têm pelo menos um vértice diferente.

   Sugestão 1: Considera um grafo completo de n nós. *Sugestão 2:* Considera um grafo "quadriculado" de n por n, dois vértices opostos (na diagonal) e os caminhos entre eles que progridem sempre no sentido da diagonal.

2. Considera o problema do caminho mais longo:

   (CML)

Instância: $G(V,E)$, $a \in V$, $b \in V$, com $a \neq b$ e inteiro k.

Questão: Existe um caminho simples entre a e b de comprimento maior ou igual a k?

Mostra que CML é **NP**-completo. Nota: Existem algoritmos polinomiais como o de Dijkstra e o de Floyd para o problema do caminho mais curto.

3. Classifica os problemas de decisão correspondentes às questões indicadas; a instância é sempre um grafo *dirigido* $G(V,E)$, $a \in V$, $b \in V$ com $a \neq b$ e k (todavia, para os 2 últimos problemas a instância não contêm k). Podes usar os resultados mencionados (ou cuja demonstração é proposta) nas alíneas anteriores; estes resultados são também válidos em grafos dirigidos.

   (a) Todos os caminhos entre a e b têm comprimento maior ou igual a k.

   (b) Todos os caminhos entre a e b têm comprimento menor ou igual a k.

   (c) Todos os caminhos simples entre a e b têm comprimento menor ou igual a k.

   (d) Existe um caminho de comprimento menor que k.

   (e) Existe um caminho de comprimento maior que k.

   (f) Não existe nenhum caminho entre a e b.

   (g) Existem caminhos de comprimento arbitrariamente grande entre a e b?

Exercício 2.6.13 Considera o seguinte problema de decisão:

(IC)*(Implicação de Cláusulas)*

Instância: Conjuntos de cláusulas C_1 e C_2 com variáveis lógicas de um conjunto U.

Questão: C_1 implica C_2? Por outras palavras: C_2 tem o valor lógico *verdade* sempre que C_1 tiver o valor lógico *verdade*?

(i) Descreve de forma resumida e utilizando uma linguagem de alto nível um algoritmo determinístico que resolve **(IC)**. O algoritmo é polinomial ou exponencial?

(ii) A que classe de complexidade pertence **IC**? Porquê?

(iii) **(PCE)***(Poucas Cláusulas Equivalentes)*

Instância: Conjunto de cláusulas C, conjunto U de variáveis lógicas e inteiro positivo k.

Questão: Existe um conjunto C' de k ou menos cláusulas utilizando variáveis de U que seja equivalente a C (isto é, tal que C e C' tenham o mesmo valor lógico qualquer que seja a atribuição de valores lógicos às variáveis de U)?

Escreve o que descobrires sobre a classe de complexidade a que pertence PEC.

Exercício 2.6.14 Considera uma máquina de Turing M, uma palavra x e um inteiro positivo t. Diz se é possível exprimir em cláusulas (eventualmente dependentes de M, x e t) os seguintes factos por forma que as cláusulas sejam satisfazíveis se e só se os factos forem verdadeiros.

1. A máquina M com os dados x pára em não mais de t passos.

2. A máquina M com os dados x pára (num número qualquer de passos).

3. A máquina M com os dados x não pára.

Exercício 2.6.15 Considera o problema **EL** da equivalência de 2 expressões lógicas que podem conter variáveis proposicionais, variáveis proposicionais negadas e as conectivas $\vee$ e $\wedge$:

(EL)

Instância: Um conjunto U de variáveis lógicas e 2 expressões lógicas E_1 e E_2 com variáveis em U.

Questão: "E_1 e E_2 são equivalentes?", isto é, têm o mesmo valor lógico para todas as atribuições de valores lógicos às variáveis de U?

Exemplo:

$$U = \{u_1, u_2\},\ E_1 = (u_1 \vee \overline{u_2}) \wedge (\overline{u_1} \vee u_2),\ E_2 = u_1 \wedge \overline{u_1},$$

(i) Considera as seguintes afirmações com as quais se pretende mostrar a dificuldade de resolver **EL**. Comenta-as devidamente.

  (a) **EL** é **NP**-completo porque nunca ninguém conseguiu um algoritmo polinomial para o resolver.

  (b) **EL** é **NP**porque não existe nenhum algoritmo polinomial para o resolver.

  (c) **EL** é **NP**-completo porque se conseguissemos resolver **EL** em tempo polinomial também era possível resolver **SAT** em tempo polinomial pois uma instância (U, C) de **SAT** é satisfazível se e só se a instância de **EL**

$$U',\ E_1 = E_c,\ E_2 = (u_n \wedge \overline{u_n})$$

  não tiver solução, onde $U' = U \cup \{u_n\}$, u_n é uma nova variável e E_c é a expressão lógica correspondente ao conjunto C de cláusulas (conjunção de disjunções).

(ii) Determina a classe de complexidade a que *de facto* pertence **EL**.

Exercício 2.6.16 Classifica o mais exactamente possível os seguintes problemas. Justifica. Em todos eles é dada uma colecção C de k objectos com tamanhos inteiros t_1, t_2, ..., t_k e um tamanho (inteiro) da "mala" t, sendo $t_i \leq t$ para $1 \leq 1 \leq k$. Repara que um mesmo problema Π pode pertencer simultâneamente a várias classes, por exemplo ser decidivel, **NP**e P; a última caracterização é a mais exacta.

1. É possível arrumar todos os objectos numa mala, isto é, $\sum_{i=1}^{k} t_i \leq t$?

2. Dado (também) m pergunta-se é possível arrumar todos os objectos em não mais de m malas?

3. Dado (também) m pergunta-se: Todas as arrumações exigem pelo menos m malas?

4. Qual o menor número de malas em que é possível arrumar todos os objectos?

5. Dados (também) m e c pergunta-se: todos os subconjuntos de C com c objectos podem ser arrumados em não mais de m malas?

6. Dados (também) m, t_m e c pergunta-se: todos os subconjuntos de C com c objectos cuja soma dos tamanhos é pelo menos t_m podem ser arrumados em não mais de m malas?

Exercício 2.6.17 Considera o problema de decisão do **(CLIQUE)** que se sabe ser **NP**–completo.

1. O correspondente problema de optimização é o seguinte: "dado um grafo não dirigido, qual o tamanho do maior *clique* que ele possui?" Mostra que se existisse um algoritmo polinomial para o problema de decisão então também existia um algoritmo polinomial para o de optimização (o recíproco é evidente).

2. Classifica os seguintes problemas de decisão utilizando apenas a informação de que o problema do **(CLIQUE)** é **NP**–completo.

   (a) Dado G e k pergunta-se: G tem dois (ou mais) "cliques" disjuntos de tamanho não inferior a k?

   (b) Dado G pergunta-se: G tem um "clique" de ordem 22?

   (c) Dado G pergunta-se: não terá G um "clique" de ordem 22?

   (d) Dados G_1 e G_2 pergunta-se: existe em G_1 um subgrafo isomorfo a G_2?

   (e) Dados G_1 e G_2 pergunta-se: não terá G_1 nenhum subgrafo isomorfo a G_2?

   (f) Dado G e k pergunta-se: existe um subconjunto com um máximo de k vértices tal que todo o ramo tem (pelo menos) uma extremidade que é um vértice desse subconjunto?

Exercício 2.6.18 Considere um problema de decisão cuja instância é (A, k) sendo k um inteiro não negativo e cuja pergunta é "$P(A, k)$?" (P é uma proposição). No correspondente problema de optimização pergunta-se "qual o máximo k tal que $P(A, k)$?". Supomos que se verifica a seguinte condição: sempre que existe solução para um determinado valor de k, existe solução para todos os k' tal que $0 \leq k' \leq k$.

1. Mostra que se existir um algoritmo polinomial para o problema de optimização então também existe um algoritmo polinomial para o problema da decisão.

2. Mostra que em condições bastante gerais o inverso (da alínea anterior) também é verdadeiro. Sugestão: considera a possível existência de um majorante de k.

3. Exemplifica os resultados anteriores para o problema do "clique".

Exercício 2.6.19 A Hierarquia cresce
Mostra que, para todo o $k \geq 0$ é

$$\Delta_k^p \subseteq \Sigma_{k+1}^p \subseteq \Delta_{k+1}^p$$

Capítulo 3

Entropia, Algoritmos e Complexidade Mínima

Este capítulo é uma introdução às aplicações da teoria da Informação aos problemas de complexidade mínima, isto é, à procura de *minorantes* da complexidade da classe dos algoritmos que resolvem um determinado problema.

Seja P um problema dado e seja $f(n)$ a complexidade (por exemplo, em termos de ordem de grandeza) do algoritmo mais eficiente A que resolve esse problema (eventualmente pode haver mais do que um); chamemos a $f(n)$ a complexidade óptima do problema. Normalmente o algoritmo óptimo A é desconhecido mas, tanto do ponto de vista teórico como do prático, é importante conhecer o mais exactamente possível — isto é, limitar a um intervalo tão pequeno quanto possível — a complexidade óptima $f(n)$.

A complexidade de qualquer algoritmo que solucione P é obviamente um majorante da complexidade óptima.

Determinar minorantes de $f(n)$ é muito mais difícil uma vez que, para se mostrar que $g(n)$ é um minorante de $f(n)$ é necessário considerar *todos* os algoritmos para P e mostrar que nenhum deles tem uma eficiência melhor do que $f(n)$; como o número desses algoritmos é infinito, um ataque directo à determinação de minorantes está fora de questão.

A Teoria da Informação pode ser utilizada para a determinação de minorantes de complexidade. A ideia básica, em linhas gerais é a seguinte. Para que um algoritmo consiga solucionar todas as "instâncias" de um determinado comprimento é necessário que obtenha, da "instância" em questão, informação suficiente para poder determinar a solução correcta. Por outras palavras, o facto de se "seleccionar" deterministicamente, de entre todas as soluções possíveis, a solução correcta, obriga a uma obtenção de informação da "instância" que representa, por si só, uma tarefa que qualquer algoritmo que resolva o problema dado tem que executar.

Neste trabalho discutiremos primeiro problemas em que a informação é obtida por pesagens em balanças de pratos. Seguidamente consideraremos a comparação entre valores numéricos como fonte de informação em problemas como a determinação de máximo, o "merge" de 2 sequências e a ordenação.

3.1 Um problema

Existem 3 bolas, sendo 2 de peso igual e uma mais pesada; podemos fazer apenas uma "pesagem" numa balança de pratos que nos dá 3 resultados possíveis: ou cai para a esquerda ou para a direita ou fica equilibrada. Qual a bola mais pesada? A resposta é simples: comparemos com a balança o peso da bola 1 com a bola 2:

- Se forem iguais, a bola 3 é a mais pesada de todas.
- Se a bola 1 pesa mais que a 2, então a bola 1 é a mais pesada de todas.
- Se a bola 2 pesa mais que a 1, então a bola 2 é a mais pesada de todas.

Neste caso, o número de soluções que o problema pode ter é 3 (qualquer uma das bolas pode ser a mais pesada) e o número de resultados possíveis das pesagens é também 3. Em geral podemos enunciar o seguinte Teorema.

Teorema 1 *Para que seja sempre possível chegar à solução é condição necessária que* $a \leq b$ *onde* a *representa o número de soluções do problema e* b *representa o número de casos que é possível distinguir com a informação obtida pelo algoritmo.*

Se são permitidas k pesagens, o número de resultados possíveis das pesagens é 3^k.

> **Problema**
> Existem 14 bolas, todas iguais, com excepção de uma que pode ser mais ou menos pesada que as outras; com 3 pesagens determinar qual a diferente e se é mais leve ou pesada que as outras.

Neste caso $a = 28$ pois qualquer uma das 14 bolas pode ser a mais leve ou a mais pesada e $b = 3^3 = 27$; como $a > b$, podemos afirmar que não existe solução!

> **Problema**
> Existem 5 bolas todas de pesos diferentes; determinar, com um máximo de 4 pesagens, a sequência das bolas ordenadas pelos seus pesos.

Temos $a = 5! = 120$, (todas as permutações das 5 bolas) e $b = 3^4 = 81$; mais uma vez o problema não tem solução. Qual o número mínimo de pesagens para que o problema *possa* ter solução? Admitamos que os pesos são tais que a balança nunca fica equilibrada (o que é consistente com o facto de os pesos serem todos diferentes). Cada pesagem só dá origem a 2 decisões possíveis. Assim temos que p, o número de pesagens, deve satisfazer $2^p \geq 120$, ou seja

$$p \geq \lceil \log 120 \rceil = 7$$

3.2 As 12 bolas

O Teorema 1 anterior não só é útil para determinar casos de impossibilidade mas pode ser também usado para seleccionar as pesagens que podem constituir uma solução. Vejamos um problema com solução e procuremo-la.

> **Problema**
> Existem 12 bolas, todas iguais, excepto uma (que pode ser mais ou menos pesada que as outras); com 3 pesagens determinar a diferente e se é mais leve ou pesada.

Neste caso $a = 24$ e $b = 3^3 = 27$ pelo que é $a \leq b$ e o problema poderá ter ou não solução; o Teorema 1 só exprime uma condição necessária para haver solução.

Procuremos uma possível solução. Utilizaremos apenas pesagens com igual número de bolas em cada prato. Se, por exemplo, colocassemos 4 bolas no prato da esquerda e 3 no da direita, e a balança caísse para a esquerda, não tiravamos qualquer conclusão: qualquer uma destas 7 bolas podia ser a mais pesada, a mais leve, ou podiam ser todas iguais!

Primeira pesagem

Temos $a = 24$ e $b = 27$; depois da primeira pesagem ficará $b' = 9$ onde b' representa o valor de b para o problema subsequente. Se colocassemos 1 bola em cada prato e a balança ficasse equilibrada, restavam 20 hipóteses (10 bolas) para serem decididas em 2 pesagens; como $20 > 9$, isso seria impossível, pelo Teorema 1. A mesma conclusão se tira se usarmos 2 (pois $16 > 9$) ou 3 (pois $12 > 9$) bolas em cada prato. Se usarmos 5 bolas em cada prato e a balança cair para a esquerda, teríamos que decidir uma de 10 hipóteses (uma das 5 bolas da esquerda pode ser mais a pesada, ou uma das 5 bolas da direita pode ser a mais leve) com 2 pesagens; ora $10 > 9$ e isso é impossível; com 6 bolas em cada prato ainda é pior. Conclusão: teremos que colocar 4 bolas em cada prato, seja $(1, 2, 3, 4) - (5, 6, 7, 8)$. Temos a considerar os casos

1. Fica equilibrado
   Uma das restantes 4 bolas, 9, 10, 11 ou 12 é a diferente; temos 8 hipóteses e $8 < 9$, tudo bem, só pode haver um resultado de pesagem desperdiçado. Façamos seguidamente a pesagem $(9, 10, 11) - (1, 2, 3)$, notando que 1, 2 e 3 são bolas iguais.
   (a) Se cai para a esquerda, ou 9 ou 10 ou 11 é a diferente e é mais pesada $(3 \leq 3)$; com uma nova pesagem entre 9 e 10 a situação fica resolvida.
   (b) Se cai para a direita é análogo: ou 9 ou 10 ou 11 é a diferente e é mais leve $(3 \leq 3)$; com uma nova pesagem entre 9 e 10 a situação fica resolvida.
   (c) Se fica equilibrada, a bola 12 é diferente; comparada com, por exemplo, a bola 1, determinamos se é mais leve ou mais pesada (aqui a balança não pode ficar equilibrada: é o resultado desperdiçado).
2. Cai para a esquerda
   Há 8 hipóteses: ou 1 ou 2 ou 3 ou 4 é a mais pesada; ou 5 ou 6 ou 7 ou 8 é a mais leve $(8 \leq 9)$ Façamos a pesagem $(1, 2, 5) - (3, 4, 6)$.
   (a) Cai para a esquerda: 1 ou 2 é a mais pesada ou 6 é a mais leve; uma nova pesagem decide a situação $(1) - (2)$.
   (b) Cai para a direita; análogo: 3 ou 4 é a mais pesada ou 5 é a mais leve; fazemos a pesagem $(3) - (4)$.
   (c) Fica equilibrada: a mais leve é 7 ou 8; pesemos $(7) - (8)$ (não pode dar igual).
3. Cai para a direita
   Raciocínio análogo ao caso 2.

   Exercício 1
   O problema análogo com 13 bolas é insolúvel (por isso é que se diz que 13 dá azar). Provar esse facto.

3.3 Entropia e informação

Se existem n hipóteses para uma situação e p_i é a probabilidade de o facto i ser verdadeiro a entropia deste estado de conhecimento é o seguinte valor que nunca é negativo

$$S = -\sum_{i=1}^{n} p_i \log p_i$$

Esta entropia mede a falta de informação ou incerteza deste estado de conhecimento; admitimos que os logaritmos são na base 2 embora isso não seja importante aqui. Uma entropia grande significa que é necessária uma grande quantidade de informação para descrever a situação. Uma entropia nula quer basicamente dizer que o conhecimento da situação é completo.

Vejamos em 2 situações o valor tomado pela entropia.

– Se temos a certeza que a hipótese 1 é válida, $p_1 = 1$ e $p_i = 0$ para $i \geq 2$. Resulta que o valor limite de S quando todos os p_i, com $i \geq 2$, tendem para 0 (e, consequentemente, p_1 tende para 1)
$$S = 1 \times \log 1 = 0$$
levando em consideração que $\lim_{p \to 0} p \log p = 0$.

– Se qualquer das n hipóteses é igualmente provável, $p_i = 1/n$ para todos os $1 \leq i \leq n$. Temos pois
$$S = -n\frac{1}{n} \log \frac{1}{n} = -\log \frac{1}{n} = \log n$$
Por exemplo, se $n = 16$, vem $S = 4$; a entropia (usando o logaritmo na base 2) mede, o número de bits de informação necessários para especificar completamente uma das hipóteses.

A fórmula da entropia não "caiu do céu". Shannon em *the Mathematical Theory of Information* demonstra que a *única* função $S(p_1, \cdots, p_n)$ que satisfaz os seguintes pressupostos razoáveis

– S é contínua em todos os p_i.

– Se todos os $p_i = 1/n$, a função $S(n)$ deverá crescer monotonicamente com n. Mais hipóteses (=acontecimentos) representam mais incerteza.

– Se uma escolha se desdobrar em duas escolhas sucessivas, a função original S deve ser a soma pesada dos S de cada escolha (ver o trabalho referido de Shannon para mais pormenores).

é do tipo
$$S = -k \sum_{i=1}^{n} p_i \log p_i$$
onde $k > 0$.

3.4 Informação e pesagens

Uma pesagem do tipo considerado pode ter 3 resultados; seja:

n o número de hipóteses possíveis antes da pesagem.

n_1, n_2 e n_3 os números de hipóteses que restam após cada pesagem.

Supondo probabilidades iguais para as situações possíveis, a entropia é diminuida de, respectivamente, $\log(n/n_1)$, $\log(n/n_2)$ e $\log(n/n_3)$. Ora, como pretendemos limitar o número máximo de pesagens em todas as situações, a situação mais favorável seria quando a divisão fosse disjunta (em hipóteses) e igual, isto é $n_1 = n_2 = n_3 = n/3$. Nesta hipótese a pesagem daria um "trit" de informação:
$$\text{Um trit} = \log_2 3 \text{ bits} \approx 1.585 \text{ bits}$$

Em termos de informação expressa em trits, a possível solução do problema das 12 bolas (ver o teorema1) resulta da desigualdade:

$$3 \text{ trits} \geq \log_3 24\text{trits} \approx 2.893 \text{ trits}$$

Nota: Nota: a desigualdade $a \leq b$ deve ser válida em todos os nós da árvore.

Exercício 2
Construir a árvore da solução dada para as 12 bolas, representando em cada nó:

- A pesagem efectuada.
- A entropia (antes da pesagem) expressa em trits.
- A entropia máxima, dado o número que falta de pesagens (0, 1, 2 ou 3).

3.5 Informação e Comparações

3.5.1 Ordenação

Vejamos agora um problema que tem a ver com os importantes algoritmos de ordenação.

Problema (Ordenação)
Sejam n bolas de pesos diferentes; pretende-se colocá-las por ordem crescente de pesos usando apenas operações de comparação entre 2 bolas, isto é, pesagens com uma bola num prato e outra noutro. Note-se que a balança nunca fica equilibrada uma vez que não existem 2 bolas de igual peso.

Aplicando directamente o Teorema 1 temos.

O número de hipóteses possíveis é $a = n!$.

O número de situações que se podem discriminar com c comparações é $b = 2^c$.

Resulta que qualquer algoritmo de ordenação que obtenha a informação através de comparações deve satisfazer $a \leq b$, ou seja,

$$2^c \geq n!$$

ou ainda

$$c \geq \lceil \log(n!) \rceil$$

Exercício 3
Determine minorantes de c para os primeiros valores de n (por exemplo entre 1 e 6) e procure algoritmos de ordenação que se aproximem tanto quanto possível desses valores (no caso mais desfavorável).

Usando a fórmula de Stirling como aproximação para o factorial

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

temos (podiamos ser mais rigorosos; o sinal $\approx$ pode ser substituido por $>$ e este facto pode ser utilizado no que se segue).

$$2^c \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

ou seja (os logaritmos são, como é usual, na base 2)

$$c \geq \frac{1}{2}\log(2\pi n) + n\log n - n\log e$$

Vemos pois que qualquer algoritmo de ordenação baseado em comparações deve fazer (no pior caso) pelo menos cerca de $n\log n$ comparações, isto é deve ser pelo menos de ordem $O(n\log n)$.

Concluimos pois que, em termos de ordens de grandeza (a afirmação não é válida se considerarmos os valores exactos do número de comparações), são conhecidos algoritmos de ordenação óptimos. Um exemplo é o "heapsort"; o "quicksort" não é óptimo, uma vez que as nossas conclusões são em termos do comportamento no pior caso (para o qual o "quicksort" é $O(n^2)$).

O teorema 1 dá para cada problema um limite mínimo da complexidade dos algoritmos respectivos; pode ser expresso, por outras palavras, da forma seguinte: o algoritmo tem que obter pelo menos tanta informação quanta é necessária para distinguir o caso particular do problema em questão. Para certos problemas — como o da ordenação — existem algoritmos próximos deste limite teórico; para outros, os melhores (teoricamente possiveis) algoritmos estão muito longe dele e há que recorrer a outros métodos, em geral mais complicados, para obter melhores minorantes.

3.5.2 Máximo

Um outro problema, mais simples que o da ordenação é o seguinte.

Problema (Máximo)
Determinar o maior de n valores distintos.

Neste caso a aplicação do Teorema fornece um resultado bem fraco: são precisas pelo menos $\log n$ comparações (o número de soluções possíveis é n); ora não é difícil mostrar que são necessárias $n-1$ comparações!

Exercício 4
Mostre que qualquer algoritmo baseado em comparações para a determinação do máximo entre n elementos tem de efectuar, pelo menos, $n-1$ comparações.

O problema de não se conseguir fazer melhor pode, de certo modo, ser explicado pelo facto de não ser possível eliminar cerca de $n/2$ elementos em cada comparação; de facto, no algoritmo usual — o maior dos elementos encontrados até à altura é comparado com o elemento corrente — fazem-se $n-1$ comparações, eliminando apenas um candidato em cada comparação.

A situação é semelhante para a procura do elemento de ordem $\lfloor n/2 \rfloor$ (a meio da tabela). Aqui existem algoritmos que fazem cerca de $3n$ comparações.

Vejamos contudo duas aplicações mais "fortes" do teorema:

Exercício 5
Temos n bolas; uma pesa 1, outra 2,…e a outra 2^{n-1} Mostre que é possível determinar a mais pesada em não mais de cerca de $\log n$ pesagens (aproximando-se do limite dado pelo teorema).

Exercício 6
Analise a complexidade mínima do seguinte problema. Existem n elementos distintos v_1, v_2,…,v_n ordenados por ordem crescente; é dado um valor x e pergunta-se: qual o menor i tal que $v_i < x$ (sendo $i = 0$ se x é menor que todos os elementos)? Por outras palavras, onde deveria x ser colocado na lista dos elementos?

Notas sobre a solução
Aqui existem $n+1$ respostas possíveis; o teorema diz-nos que o número de comparações deve satisfazer: $2^c \geq n+1$, ou seja $c \geq \lceil \log(n+1) \rceil$.

Ora existe um algoritmo — o da pesquisa binária — que faz exactamente este número de comparações; não pode haver melhor!

Exercício 7
Em cada um dos casos seguintes pergunta-se quantas comparações são necessárias pelo teorema? E na realidade? Qual a entropia da situação inicial?

1. Eu "penso" num inteiro entre 1 e 1000; tu tentas adivinhar qual é, tentando de cada vez um número; eu respondo: é maior, menor ou é esse.
2. Existem n inteiros entre 1 e n; determinar com comparações o seu valor.

3.5.3 "Merge"

Consideremos agora o algoritmo do "merge".

Problema ("Merge")
São dadas 2 sequências de ordenadas de inteiros, uma com n e a outra com m elementos. Pretende-se obter a sequência ordenada constituída pelos $m+n$ elementos.

Um algoritmo de resolver este problema é baseado na aplicação iterativa do seguinte passo. Os dois menores elementos das sequências dadas são comparados; o menor deles é retirado e colocado na sequência resultado. Quando uma das sequências dadas "chegar ao fim", o resto da outra é "passado" para a sequência resultado. Supondo todos os elementos distintos, é fácil ver que são feitas no máximo $m+n-1$ comparações.

Vejamos se este algoritmo se aproxima do limite teórico; o número de situações possíveis diferentes entre as duas sequências é:

$$\binom{m+n}{n}$$

Este número, igual a $(m+n)!/(n!m!)$, pode ser aproximado usando a fórmula de Stirling; vamos supor apenas o caso $m=n$.

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2} \approx (\pi n)^{-1/2} 2^{2n}$$

Sendo c o número de comparações efectuadas, deverá ser pois:

$$2^c \geq \left\lceil (\pi n)^{-1/2} 2^{2n} \right\rceil$$

ou seja:

$$c \geq \left\lceil 2n - \frac{1}{2}\log(\pi n) \right\rceil$$

Ora, como dissemos, o algoritmo do "merge" faz neste caso $2n-1$ comparações na pior hipótese. Vemos que, para n grande, o algoritmo é quase óptimo. Por exemplo, para $n=100$, o algoritmo faz 199 comparações, enquanto o minorante (confiando na aritmética do yap) é $\lceil 195.852 \rceil = 196$.

Capítulo 4

Complexidade de Kolmogorov e Informação Algorítmica

Bibliografia [LV90], [Cha87], [Cha94], [LV94]

Quanto é possível comprimir a informação dum objecto? Consideremos a unidade de informação o *bit*. A cada bit está associada uma escolha entre duas possibilidades e (a informação de) cada objecto pode ser codificado por uma sequência de bits. Assim, podemos restringir-nos ao estudo de sequências finitas de bits, isto é, a elementos de $\{0,1\}^\star$. Certamente, numa dada sequência pode haver bits em redundantes. Isto é, a mesma sequência pode ser descrita por outra de menor comprimento. Iremos estudar a possibilidade de *comprimir* a informação associada a um objecto.

4.1 Informação e Objectos

A Teoria de Informação segundo Shannon ([Sha48]) quantifica a informação duma situação em termos da probabilidade de cada uma das suas n instâncias.

Seja $C_n \subset \{0,1\}^\star$ o conjunto das sequências de comprimento n e suponhamos que cada uma é igualmente provável. Como $|C_n| = 2^n$ e a probabilidade de cada uma das suas sequências é $\frac{1}{2^n}$, a entropia associada é n. Isto é, o número de bits necessários para descrever uma qualquer sequência de C_n é precisamente igual ao seu comprimento. Contudo, existem sequências nesse conjunto que podem ser descritas com menos bits. Por exemplo, a sequência 1^n pode ser descrita em $O(\log n)$ bits, considerando a representação em binário de n e um algoritmo apropriado. Exemplo dum tal algoritmo é:

```
para i<-1 até n faça {
   escreva(1)
}
```

Esta sequência é *compressível* porque pode ser descrita com menos bits que o seu tamanho; diz-se que a informação associada é "pequena".

Por outro lado, uma sequência específica correspondente a n lançamentos duma moeda, associando "coroa" a 0 e "cara" a 1 provavelmente não se pode descrever com menos de n bits.

Consoante ao seu grau de regularidade ou, inversamente, de *aleatoriedade* uma sequência pode ser mais ou menos compressível.

Informalmente, a quantidade de informação (complexidade) duma sequência x é definida como o tamanho (número de bits) do menor "algoritmo" que tenha como resultado x.

A Complexidade de Kolmogorov (ou Teoria Algorítmica da Informação), tem como objectivo estudar a quantidade de informação dum objecto (sequência). A noção básica é a de aleatoriedade. Uma sequência é aleatória se não for compressível. Iremos ver que, existem algumas sequências que são muito compressíveis, muitas outras que são incompressíveis e algumas que para serem comprimidas necessitam de algoritmos intratáveis.

Consideremos mais alguns exemplos.

(i) A sequência

$$110011001100\ldots1100 \quad = \quad (1100)^{1000}$$

é compressível, isto é, existe um programa p cujo resultado é essa sequência e cujo tamanho é menor que 4000 bits.

Exercício 4.1.1 *Escreva em linguagem C um programa que tenha como resultado a sequência* $(1100)^{1000}$*. Indique o tamanho do programa em bits .*

(ii) A sequência $\langle 3141596\cdots\rangle$ constituída pelos primeiros 10^6 dígitos de π, é compressível. Existem programas "pequenos" que produzem sucessivamente os dígitos de π. Esta sequência pode ser descrita em $O(1)$ bits.

(iii) Um milhão de bits obtidos por um gerador de pseudo-aleatórios. (função `rand`). Existem programas "pequenos" que geram esta sequência.

Critério Uma sequência $x \in \{0,1\}^\star$ é compressível se, numa linguagem fixada, existe um programa $p \in \{0,1\}^\star$ tal que p produz x e $|p| < |x|$.

Programas e máquinas de Turing

A linguagem ou modelo de computação que iremos considerar é a máquina de Turing. Em particular, o "programa" é o conteúdo inicial da fita de entrada de uma máquina de Turing universal específica.

Todas as máquinas de Turing que consideraremos têm 3 fitas e as correspondentes cabeças:

- Uma fita de entrada que contém o programa e que apenas pode ser lida, avançando a cabeça para a direita (de uma unidade).
- Uma fita de saída, inicialmente vazia, que apenas pode ser escrita, avançando a correspondente cabeça para a direita (de uma unidade).
- Uma fita de trabalho, inicialmente vazia, que pode ser lida e escrita, podendo a correspondente cabeça mover-se em cada passo de uma unidade para a direita ou para a esquerda.

As máquinas que consideramos são determinísticas, podendo cada passo ser uma operação de leitura, de escrita ou uma operação na fita de trabalho. Estas convenções não são essenciais; poderia existir apenas uma fita de trabalho cujo conteúdo inicial seria o programa. Contudo, são muito utilizadas e úteis em certas circunstâncias.

Se n é o número da máquina M, então, a máquina de Turing universal que tomamos como referência, com o programa 0^n1p, tem um comportamento idêntico ao da máquina M com o programa p. A sequência $0^n 1p$ codifica o par de palavras $\langle n, p\rangle$; por exemplo, se $n = 1000$ e $p = 0011101'$, o par$\langle n, p\rangle$ fica codificado na palavra (dado que n é 8)

$$0^8 1p = 0000000010011101$$

Veremos outros processos de codificar um par de palavras x e y numa só palavra w (por forma que x e y possam ser calculados a partir de w). É óbvio que a simples concatenação xy não serve para esse efeito!

4.2 Complexidade Algorítmica

Seja $\Sigma = \{0, 1\}$. Iremos identificar $\Sigma^\star$ com $\mathbb{N}$ considerando a bijecção que a cada $n \in \mathbb{N}$ faz corresponder a n-ésima sequência binária na ordem lexicográfica (crescente): $(0, \epsilon)$, $(1, 0)$, $(2, 1)$, $(3, 00)$, $(4, 01)$, $(5, 10)$, $(6, 11)$, $(7, 000)$, ... Assim x pode denotar um inteiro ou uma sequência de bits. O comprimento ou tamanho (número de bits) de x é dado, como habitualmente, por $|x|$.

Exercício 4.2.1 *Determine formas fechadas para a bijecção indicada $f : \mathbb{N} \to \Sigma^\star$ e para a sua inversa f^{-1}.*

Seja M uma máquina de Turing calculadora de funções e $x,\ p \in \Sigma^\star$. Denotamos por $M(p) = x$ o facto da máquina de Turing M com dados p parar com a sequência x na fita (M calcula x com dados p). A complexidade $K_M(x)$ de x em relação a M é definida por:

$$K_M(x) \quad = \quad \begin{cases} min\{|p| \ :\ M(p) = x\} & \\ \infty & \text{se não existe } p \text{ tal que } M(p) = x \end{cases}$$

Exemplo 11 *Consideremos algumas máquinas de Turing e analisemos a complexidade de algumas sequência em relação a essas máquinas.*

(i) Seja M_1 uma máquina de Turing que se limita a copiar o programa x para a fita de saída. Então, se $x \in \Sigma^\star$ tem-se $K_{M_1}(x) = |x|$.

(ii) Seja M_2 uma máquina de Turing que para quaisquer dados apaga a fita e pára. Então

$$K_{M_2}(x) \quad = \quad \begin{cases} 0 & \textit{se } x = \epsilon \\ \infty & \textit{se } x \neq \epsilon \end{cases}$$

(iii) Seja M_3 uma máquina de Turing que calcule $2x$.

$$K_{M_3}(x) \quad = \quad \begin{cases} |x/2| & \textit{se } x \textit{ é par} \\ \infty & \textit{caso contrário} \end{cases}$$

(iv) Máquina Universal U. Se existe uma máquina de Turing M que, com um programa p calcula x então x também pode ser calculado pela máquina universal. Mais concretamente, a máquina u com o programa $0^{e(M)}1p$ produz x. Usando a tese de Church-Turing podemos afirmar que, se x pode ser calculado, então pode ser calculado por U. "Universal" é um nome bem posto!

Pode-se definir a complexidade de x em relação a outra sequência y. A ideia é que a informação de y pode ser usada para definir um programa p que calcule x. A complexidade $K_M(x|y)$ de x em relação a M condicional a y é definida por

$$K_M(x|y) \quad = \quad \begin{cases} min\{|p| \ : \ M(\langle p,y\rangle) = x\} & \\ \infty & \text{se não existe } p \text{ tal que } M(\langle p,y\rangle) = x \end{cases}$$

onde $\langle\cdot,\cdot\rangle$ é a bijecção usual de $\mathbb{N}^2$ em $\mathbb{N}$.

Exemplo 12 *Para qualquer máquina de Turing M e para toda a sequência $x \in \Sigma^\star$, tem-se que $K_M(x|\epsilon)$ é igual a $K_M(x)$. E como será $K_M(x|x)$?*

Pela definição, a complexidade K_M de x depende da máquina M, portanto a noção de compressibilidade é também dependente da máquina usada. Em particular, uma sequência x pode ser incompressível (aleatória) para uma máquina M e compressível para outra máquina M'. Vamos ver contudo que, a menos duma constante, a complexidade de x não depende da máquina M escolhida. O Teorema da Invariância permite concluir que dadas duas máquinas de Turing M_1 e M_2 para todo $x \in \Sigma$

$$|K_{M_1}(x) - K_{M_2}(x)| \leq c_{M_1M_2}$$

isto é, a complexidade de x em relação a duas máquinas difere duma constante que não depende de x mas apenas das máquinas referidas.

Teorema 4.2.1 (Teorema da Invariância) *Existe uma máquina de Turing U, tal que para toda a máquina de Turing M, existe uma constante c_M tal que para toda a sequência $x \in \Sigma^\star$*

$$K_U(x) \leq K_M(x) + c_M$$

Dem. Seja U uma máquina universal de Turing e seja $n = e(M)$ o número da máquina M para uma dada enumeração (codificação) e. Suponhamos que $M(p) = x$. Considerando os inteiros codificados em unário temos que $U(O^n1p) = x$. Escolhendo $c_M = n + 1$ vem

$$\forall x \in \{0,1\}^\star \;\; K_U(x) \leq n + 1 + K_M(x)$$

□

O Teorema da Invariância também é válido se considerarmos a complexidade $K_M(x|y)$ de x condicional a y.

Exercício 4.2.2 Enuncie e prove o Teorema da Invariância para complexidades condicionais.

O Teorema da Invariância pode ainda ser expresso em termos de funções recursivas.

Exercício 4.2.3 Defina complexidade de $x \in \Sigma^\star$ em relação a uma função recursiva f.

Teorema 4.2.2 (Teorema da Invariância para funções recursivas) *Existe uma função recursiva f_0 tal que para toda a função recursiva f existe uma constante c_f tal que para toda a sequência $x \in \Sigma^\star$*

$$K_{f_0}(x) \leq K_f(x) + c_f$$

Em todos os casos, considerando sequências x suficientemente grandes a constante pode ser desprezada. Podemos, ainda, fixar uma máquina universal U de referência e estudar a complexidade duma sequência x em relação apenas a U. O índice U pode ser omitido na definição de $K(x)$ ou de $K(x|y)$

Exemplo 13 *Mostrar que para todo o* $x \in \Sigma$, $K(xx) \leq K(x) + O(1)$.

Seja U *a máquina de referência, de número* $n = e(U)$ *e seja* p *o programa mínimo tal que* U *com dados* p *calcula* x. *Tem-se que* $K(x) = |p|$. *Seja* V *uma máquina de Turing, de número* $n_V = e(V)$, *que com dados* 0^n1p *simula* U *com dados* p, *mas que duplica o resultado antes de parar. Assim* V *com dados* 0^n1p *calcula* xx *e portanto* U *com dados* $0^{n_V}10^n1p$ *também calcula* xx. *Então, para todo o* $x \in \Sigma^\star$

$$K(xx) \leq K(x) + n_V + n + 2$$

Exemplo 14 *Seja* $K(x, y) = K(\langle x, y\rangle >)$, *onde* $\langle ., . >$ *é a bijecção usual de* $\mathbb{N}^2$ *em* $\mathbb{N}$. *Poderíamos pensar que, para quaisquer* x *e* y,

$$K(x, y) \leq K(x) + K(y) + O(1)$$

Vamos ver que isso não é verdade.

Seja p *o menor programa que produz* x *e* q *o menor programa que produz* y. *Suponhamos que* M *é uma máquina, de número* $n = e(M)$, *que dado* pq *usa* p *para produzir* x *e em seguida usa* q *para produzir* y. *Se* M *existir então* U *com dados* 0^n1pq *calcula* $\langle x, y\rangle$ *e ter-se-ia*

$$K(x, y) \leq K(x) + K(y) + n + 1$$

Contudo, uma tal máquina não existe porque é necessário saber dividir os dados pq *de modo a identificar* p *e* q. *Isto é, não se pode usar como codificação de* p *e* q *a sequência* pq *e portanto* M *não pode ser de complexidade constante.*

Podemos separar p *de* q *se for conhecido o comprimento de* p, *por exemplo, se os dados forem* $\overline{|p|}pq$ *e soubermos separar* p *de* $\overline{|p|}$.

Podemos codificar $|p|$ *em* $O(\log |p|)$ *bits considerando o seguinte método: dada uma sequência* $x \in \Sigma^\star$, *a sequência* $\overline{x}$ *é obtida inserindo um* 0 *entre dois símbolos adjacentes de* x, *e adicionando um* 1 *no final. Por exemplo,* $\overline{01011} = 0010001011$. *Com esta representação é imediato detectar quando uma sequência termina. A sequência* $x' = \overline{|x|}x$ *é denominada uma versão* auto-delimitada *de* x *e tem comprimento* $|x| + 2\log|x|$ *(bits). Se* x *for a representação em binário de um inteiro* n, *então* x' *requer* $\log n + 2 \log \log n$ *bits.*

Voltando ao nosso exemplo, suponhamos $p = 110101101$ *e* $q = 1110110111$. *Então,* $|p| = 9$ *e* $\overline{|p|} = \overline{1001} = 10000011$. *A codificação de* p *e* q *é então*

$$p'q = 100000111101011011110110111$$

No caso geral, tem-se que

$$K(\langle x, y\rangle) \leq K(x) + K(y) + O(\log min(K(x), K(y)))$$

Exemplo 15 *Linguagens* `C` *e* `Prolog`. *Será que um programa de manipulação simbólica em Prolog é menor que um programa equivalente em* C? *E para um programa de calculo numérico ter-se-á o inverso? O Teorema da Invariância mostra que para exprimir sucintamente um algoritmo num programa não interessa qual a linguagem de programação usada, a menos duma constante aditiva que apenas depende das duas linguagens e não do algoritmo. Seja* $\pi_1, \pi_2, \ldots$ *uma enumeração lexicográfica de todos os programas sintácticamente correctos em* `Prolog` *e* $\gamma_1, \gamma_2, \ldots$ *uma enumeração análoga para os programas em* `C`. *Cada programa de ambas as linguagens pode ser visto como uma função recursiva parcial dos seus dados nos seus resultados. Escolhendo programas de referência adequados podemos definir* $K_{\texttt{Prolog}}(x)$ *e* $K_{\texttt{C}}(x)$ *em analogia com* $K(x)$. *Todas esta medidas coincidem a menos de uma constante aditiva. Provemos este facto. Cada enumeração* *contém um programa universal; a enumeração de* $\mathtt{C}$ *contém um interpretador de* C *e também um interpretador de Prolog, seja* γ_p. *Então,* $K_{\mathtt{C}}(x) \leq K_{\mathtt{Prolog}}(x) + c_p$, *onde* c_p *é uma constante associada a* p. *Por exemplo,* c_p *pode ser* $2|p|+1$ *se codificarmos* p *de forma auto-delimitada por* $0^{|p|}1p$. *Analogamente se prova que existe* π_c *tal que* $K_{\mathtt{Prolog}}(x) \leq K_{\mathtt{C}}(x) + c_c$. *Logo, para todo o* x, $|K_{\mathtt{C}}(x) - K_{\mathtt{Prolog}}(x)| \leq c_p + c_c$

4.3 Compressibilidade e Aleatoriedade

O Teorema da Invariância permite obter um majorante da complexidade $K(x)$ de $x \in \Sigma^\star$. Intuitivamente, a complexidade de Kolmogorov de x não deve exceder, a menos de uma constante, o seu comprimento $|x|$, dado que x é obviamente uma descrição de x.

Teorema 4.3.1 *Existe uma constante* c, *tal que para todo o* x *e* y *tem-se que*

$$\begin{aligned} K(x) &\leq |x| + c \\ K(x|y) &\leq K(x) + c \end{aligned}$$

Dem. Seja $n = |x|$ e M uma máquina de Turing que apenas copia os seus dados (programa) para a fita de saída. Para todo $x \in \Sigma^\star$, $K_M(x) = n$. Pelo Teorema da Invariância, $p = 0^{e(M)}1x$ é um programa para a máquina universal de referência U que produz x. Então vem,

$$K(x) \leq n + e(M) + 1$$

o que prova a primeira inequação.

Para provar a segunda inequação, constrói-se uma máquina de Turing M que para todos os $y,\ z \in \Sigma^\star$ calcula x com dados $\langle y, z\rangle$ se e só se a máquina universal de referência U calcula x dado z. Então, $K_M(x|y) = K(x)$. Pelo Teorema da Invariância para condicionais, existe c tal que

$$K(x|y) \leq K_M(x|y) + c = K(x) + c$$

□

Dado que temos um majorante da complexidade de x, será que podemos obter minorantes da complexidade de x? Quanto pode ser comprimida uma dada sequência x? Por outro lado, será que todas as sequências são compressíveis? Será que existem sequências incompressíveis? E quantas?

Já vimos que há sequências que podiam se descritas por programas muito mais curtos que elas próprias. Consideremos outro exemplo. Seja $f(1) = 2$ e $f(i) = 2^{f(i-1)}$ para todo $i > 1$. Esta função cresce muito rapidamente com i. Contudo para cada i o inteiro $y = 2^{f(i)}$ tem no máximo complexidade $K(i) + c$, para uma constante c independente de i.

E quanto à incompressibilidade? Dizemos que uma sequência x de tamanho $|x| = n$ é *incompressível* se $K(x) \geq n$.

Proposição 4.3.1 *Para todo o* $n > 1$ *existe uma sequência* x *de tamanho* $|x| = n$ *que é incompressível.*

Dem. Um simples argumento de contagem demonstra o resultado. Para qualquer $n \geq 0$ existem 2^n sequências binárias de tamanho n. Contudo o número de descrições (programas) de tamanho menor que n é apenas $\sum_{i=0}^{n-1} 2^i = 2^n - 1$.

□

Exercício 4.3.1 Mostre para qualquer n e para qualquer sequência $y \in \Sigma^\star$, existe uma sequência $x \in \Sigma^\star$, com $|x| = n$ tal que $K(x|y) \geq n$.

Exemplo 16 *Será que uma sequência incompressível pode ter uma subsequência compressível? Seja $x = uvw$ tal que x pode ser descrita por um programa pequeno p para v e pela sequência uw. Se para cada uma das subsequências p e u se considerar um prefixo com uma descrição auto-delimitada do seu tamanho, então $q = \overline{|p|}p\overline{|u|}uw$ é um programa para x. E, $|q| \leq K(v) + |uw| + O(\log |x|)$ bits. Então,*

$$K(x) \leq K(v) + |uw| + O(\log |x|)$$

Se x for incompressível é $K(x) \geq |x| + c$, logo temos (dado que $|x| = |v| + |uw|$):

$$K(v) \geq |v| - O(\log |x|)$$

Intuitivamente vemos que, se x e v forem de tamanhos comparáveis, v pode, em princípio, ser compressível mas muito pouco...

Quantas são as sequências que são incompressíveis? Existe pelo menos uma sequência de tamanho n que não pode ter uma complexidade menor que n. Pelo menos metade das sequências de tamanho n não são compressíveis para complexidade menor que $n-1$, uma vez que só existem $2^{n-1}-1$ programas de tamanho menor que $n-1$. Pelo mesmo raciocínio, pelo menos três quartos de todas as sequências de tamanho n não podem ser compressíveis para complexidade menor que $n-2$; e assim por diante. Para $n-c$, só 2^{-c} sequências de tamanho n podem ser descritas com menos de $n-c$ bits.

Em geral, para cada constante c dizemos que uma sequência x é *c-incompressível* se $K(x) \geq |x| - c$.

Teorema 4.3.2 (Teorema da Incompressibilidade) *Seja c um inteiro positivo. Para cada y fixo, todo o conjunto finito $A \subseteq \Sigma^\star$ com cardinal $\#A = m$ tem pelo menos $m(1-2^{-c})+1$ elementos x com $C(x|y) \geq \log m - c$.*

Exercício 4.3.2 Mostre o teorema 4.3.2.

Sendo $A = \{x \; : \; |x| = n\}$ tem-se que $\#A = 2^n$. Pelos teoremas 4.3.2 e 4.3.1 tem-se que para quase todo o $x \in A$ x, $n - c \leq K(x) \leq n + c$ sendo c a constante do teorema 4.3.1. Isto é, há poucas sequências de complexidade baixa...

Exemplo 17 *Seja $p(x)$ o menor programa que produz $x \in \Sigma^\star$ na máquina de referência U, isto é, $K(x) = |p(x)|$. Vamos ver que $p(x)$ é incompressível, no sentido em que existe uma constante $c > 0$ tal que para toda a sequência x, $K(p(x)) \geq |p(x)| - c$.*

Por redução ao absurdo, suponhamos que para qualquer constante c existe um x e uma programa $q = p(p(x))$ que produz p e tal que $|q| < |p| - c$. Seja V uma máquina semelhante à máquina universal de referência U excepto que V simula U e depois usa o seu resultado como dados e simula U mais uma vez. Seja $n = e(V)$. Tem-se que U com dados 0^n1q calcula x. Então, $K(x) < |p| - c + n + 1$ e se $c \geq n + 1$ obtemos uma contradição. Podemos concluir então que para $p(x)$ suficientemente grandes tem-se

$$\lim_{|p(x)| \to \infty} \frac{K(p(x))}{|p(x)|} = 1$$

Exemplo 18 *Quantas sequências $x \in \Sigma^\star$ têm $K(x) < |x| - 20$?*

Considere-se a razão

$$\frac{2^{|x|-20}-1}{2^{|x|}} \approx 10^{-6}$$

Exemplo 19 *Quantas sequências $x \in \Sigma^\star$ têm $K(x) \leq \alpha|x|$ com $0 < \alpha < 1$?*

Considere-se a razão

$$f_\alpha(n) = \frac{\#\{x\ :\ |x| = n,\ K(x) \leq \alpha n\}}{\#\{x\ :\ |x| = n\}} \leq \frac{2^{\lfloor \alpha n \rfloor + 1} - 1}{2^n} < 2^{-(1-\alpha)n+1}$$

Então

$$\lim_{n\to\infty} f_\alpha(n) = 0$$

Do exemplo anterior, podemos concluir que quase todas as sequências são incompressíveis!!

Dada uma função $g(n)$ dizemos que uma sequência x de tamanho n é *g-incompressível* se

$$K(x) \geq n - g(n)$$

Como acima, há 2^n sequências de tamanho n e apenas $2^{n-g(n)} - 1$ descrições de tamanho menor que $n - g(n)$. Então,

$$f(n) = \frac{\#\{x\ :\ |x| = n,\ K(x) < n - g(n)\}}{\#\{x\ :\ |x| = n\}} = \frac{2^{n-g(n)}-1}{2^n} < 2^{-g(n)}$$

e $\lim_{g(n)\to\infty} f(n) = 0$

As sequências que são incompressíveis (ou c-incompressíveis para c pequeno) não apresentam regularidades, senão essa regularidade poderia ser usada para a comprimir. Intuitivamente, dizemos que essas sequências são *aleatórias*. Formalmente pode-se provar que as sequências incompressíveis verificam os testes clássicos de aleatoriedade.

Uma sequência finita x de tamanho n diz-se *aleatória (segundo Kolmogorov)* se é $O(\log n)$-incompressível.

4.4 Propriedades de K como função inteira

A função $K(x)$ pode ser vista como uma função de $\mathbb{N}$ em $\mathbb{N}$.

Lema 4.4.1 *Para qualquer $x \in \Sigma^\star$ tem-se que:*

a) $K(x) \leq |x|$ *a menos uma constante que não depende de* x

b) Para todo a constante c*,*

$$\frac{\#\{x\ :\ |x| = n,\ K(x) < n - c\}}{2^n} < 2^{-c}$$

c) $\lim_{x\to\infty} K(x) = \infty$

d) Seja $m(x) = min\{K(y)\ :\ y \geq x\}$*, isto é, a maior função monótona crescente que limita inferiormente* $K(x)$*. Então,* $\lim_{x\to\infty} m(x) = \infty$*.*

e) Para qualquer função parcial recursiva $\phi(x)$ que tenda monotonamente para ∞ tem-se que $m(x) < \phi(x)$, para x suficientemente grande.

f) $K(x)$ é contínua no sentido em que para qualquer h, $|K(x+h)-K(x)| \leq 2|h|$ a menos duma constante que não depende de h nem de x.

Dem. d) Para cada n existe um menor x_n tal que para todo o $x > x_n$ o menor programa p que produz x tem tamanho maior ou igual a n. Isto porque, para cada n existe apenas um número finito de programas de tamanho $< n$. Claramente, $x_{n+1} \geq x_n$, para todo o n. Por outro lado, $m(x) = n+1$ para todo o $x_n < x < x_{n+1}$. O que prova o pretendido.

e) Por redução ao absurdo. Suponhamos que existia $\phi(x)$ tal que $\phi(x) \leq m(x)$ para uma infinidade de x. O conjunto $A = \{x : \phi(x) < \infty\}$ é infinito e recursivamente enumerável. Então, tem um subconjunto $B \subset A$ que é infinito e recursivo. Seja

$$\psi(x) = \begin{cases} \phi(x) & \text{se } x \in B \\ \phi(y) & \text{com } y = max\{z : z \in B,\ z < x\}, \text{ caso contrário} \end{cases}$$

A função ψ é total recursiva e $\psi(x) \leq m(x)$ para uma infinidade de x.

Defina-se $M(a) = max\{x : K(x) \leq a\}$. Então, $M(a)+1 = min\{x : m(x) > a\}$. Tem-se que

$$F(a) = max\{x : \psi(x) \leq a+1\} \geq min\{x : m(x) > a\} > M(a)$$

para uma infinidade de a's e $F(a)$ é recursiva total. Isto é, $K(F(a)) > a$ para uma infinidade de a's. Mas pelo Teorema da Invariância

$$K(F(a)) \leq K_F(F(a)) + O(1) \leq |a| + O(1)$$

Isto implica que existe uma constante c tal que $|a| + c \geq a$, para um número infinito de a's, o que é impossível.

f) Seja p o programa tal que $K(x) = |p|$. Podemos descrever $x+h$ por $\overline{h}p$. Como $|\overline{h}| \leq 2|h|$, tem-se o pretendido (a menos uma constante). □

Um efeito da quantidade de informação associada ao tamanho de uma sequência x é que $K(x)$ é não monótona *em prefixos*. Mais concretamente:

Exemplo 20 *É certo que, para $m \leq n$ podemos ter $K(m) > K(n)$ (Porquê?). Mas então, para $x = 1^n$ e $y = 1^m$ vem $K(x) < K(y)$, apesar de y ser um prefixo próprio de x. Por exemplo, se $n = 2^k$ então $K(1^n) \leq \log\log n + O(1)$, mas pelo Teorema da Incompressibilidade existe $m < n$ tal que $K(1^m) \geq logn - O(1)$. Em conclusão, a complexidade duma parte é maior que a complexidade do todo.*

Podemos tentar resolver a incongruência do exemplo anterior considerando o tamanho de x como dado, isto é, considerando a complexidade condicional de x dado $|x| = n$, $K(x|n)$. Informalmente isto permite um ganho da ordem de $\log|x|$ bits no tamanho do menor programa para x.

Para qualquer sequência x, com $|x| = n$ tem-se $K(x|n) \leq K(x) + O(1)$. No caso geral o tamanho de x não parece poder fornecer muita informação acerca do padrão de 0's e 1's em x. No entanto, às vezes a informação de $|x|$ permite determinar esse padrão:

Exemplo 21 *Seja $x = 1^n$. Então, $K(x) = K(n) + O(1)$ e $K(x|n) = O(1)$.*

Contudo, como se disse, no caso geral $K(x|n)$ não vai resolver o problema da não monotonia de prefixos:

Exemplo 22 *Seja $x = n0\ldots0$ com $|x| = n$, isto é, $x = n0^{n-|n|}$. Para cada n, x é denominada a n-sequência. Existe uma constante c tal que para todo o n, $K(x|n) \leq c$. Por exemplo, dado n podemos encontrar a n-ésima sequência binária (considerando a bijecção referida entre $\mathbb{N}$ e $\Sigma^\star$) e preencher com zeros até obter uma sequência de tamanho n. Suponhamos que n é aleatório, isto é, que $K(n) \geq |n|$. Então, para $x = n0^{n-|n|}$ tem-se ainda $K(x|n) \leq c$. Mas $K(n||n|) \geq K(n) - K(|n|) \geq \log n - 2\log\log n$.*

Teorema 4.4.1 *A função $K(x)$ não é uma função recursiva. Mais ainda, nenhuma função parcial recursiva $\phi(x)$, definida num conjunto infinito, pode coincidir com $K(x)$ em todo o seu domínio de definição.*

Dem. Qualquer conjunto infinito r.e. contém um subconjunto infinito recursivo. Seja A um conjunto recursivo infinito no domínio de definição de ϕ. A função $\psi(m) = min\{x : K(x) \geq m,\ x \in A\}$ é total recursiva (dado que $K(x) = \phi(x)$ em A) e toma valores arbitrariamente grandes. Também, por definição de ψ, $K(\psi(m)) \geq m$. Por outro lado, $K(\psi(m)) \leq K_\psi(\psi(m)) + c_\psi$ e $K_\psi(\psi(m)) \leq |m|$. Então, $m \leq \log m$ a menos uma constante independente de m, o que é falso. □

Logo, pela Tese de Church-Turing $K(x)$ não é computável.

Contudo,

Teorema 4.4.2 *Existe uma função recursiva total $\phi(t, x)$, monótona decrescente em t, tal que $\lim_{t\to\infty} \phi(t, x) = K(x)$.*

Dem. Seja c a constante tal que $K(x) \leq |x| + c$ para todo o x. Define-se, $\phi(t, x)$ como o tamanho do menor programa p, com $|p| \leq |x| + c$, e tal que a máquina de referência U com dados p pára com resultado x em t passos. Basta notar que o conjunto dos p tal que $|p| \leq |x| + c$ é finito e portanto podemos executar U para cada um desses p por t passos. Suponhamos primeiro que a máquina pára com resultado x para um ou mais desses p; neste caso define-se $\phi(t, x)$ como o menor comprimento desses programas. Se não existir p nessas condições, define-se $\phi(t, x)$ é $|x| + c$. Claramente, $\phi(t, x)$ é recursiva total e monotonamente decrescente com t. O limite existe pois para cada x existe um t tal que U pára com resultado x depois de t passos começando com dados p tal que $K(x) = |p|$. □

Embora se tenha provado que quase todas as sequências são aleatórias, num dado sistema formal apenas para um número finito dessas sequências é possível demonstrar a sua aleatoriedade. De facto, para qualquer função recursiva total tal que $\lim_{x\to\infty} f(x) = \infty$ o conjunto dos x tal que se pode provar que $K(x) > f(x)$ é finito. Assim se $f(x) \ll |x|$, embora se saiba que quase todos os x verificam $K(x) \gg f(x)$ isso só se pode demonstrar para um número finito de x.

Teorema 4.4.3

(i) O conjunto $A = \{(x, a) : K(x) \leq a\}$ é r.e. mas não é recursivo.

(ii) Qualquer função parcial recursiva $\phi(x)$ que é um minorante de $K(x)$ é limitada (isto é, existe c tal que, para todo o x, é $\phi(x) < c$).

(iii) Seja $f(x)$ uma função total recursiva tal que $f(x) \leq \log x$ e $\lim_{x\to\infty} f(x) = \infty$. Então, o conjunto $B = \{x : K(x) \leq f(x)\}$ é simples*, isto é, é r.e. e o complementar de B é infinito mas não contém um subconjunto infinito r.e.*

Dem.

(i) O facto de A ser r.e. resulta do Teorema 4.4.2. Contudo, A não é recursivo. Sabemos que existe uma constante c tal que para todo o x, tem-se que $K(x) \leq |x| + c$. Se A fosse recursivo então usando este majorante podíamos calcular $K(x)$ (Verifique!), o que contradiz o Teorema 4.4.1.

(ii) Seja ϕ uma função parcial recursiva e seja $D = \{x : \phi(x) \leq K(x)\}$. Se D é finito nada há a provar. Suponhamos que D é infinito e ϕ é ilimitada. Considere-se uma enumeração do domínio de ϕ e defina-se a função total recursiva g

$$g(n) = min\{x : \phi(x) \geq n\}$$

onde o mínimo é considerado nessa enumeração. Supondo ϕ ilimitada, para cada n existe um tal x. Seja $k = e(\phi)$ na enumeração de todas as funções recursivas associada à máquina universal de referência U. Então, podemos usar ϕ e n para calcular x, isto é, $K(x) \leq |n| + |\overline{k}|$ a menos duma constante. Por outro lado, $K(x) \geq \phi(n) \geq n$. Para n suficientemente grande, obtemos uma contradição.

(iii) B é r.e. pela alínea (i). O complementar de B, $\overline{B} = \{x : K(x) > f(x)\}$ é infinito pelo Teorema da Incompressibilidade. Vamos demonstrar que B é simples. Suponhamos que existia D, conjunto infinito r.e. contido em $\overline{B}$. A restrição f_D de f a D é uma função parcial recursiva que é um limite inferior de $K(x)$. Pela alínea (ii) f_D é limitada. Como $f(x)$ é ilimitada isto implica que D é finito. Absurdo!

□

Teorias formais e complexidade de Kolmogorov

Recordemos sucintamente a noção de sistema (teoria) formal. Uma teoria formal T é um conjunto de fórmulas. Numa teoria T considera-se o conjunto das fórmulas *verdadeiras* (axiomas) e o conjunto de fórmulas *demonstráveis* em T, de acordo com uma noção (sintáctica) de demonstração. Uma teoria é *axiomatizável* se for r.e. Por exemplo, se os seus axiomas podem ser efectivamente enumerados e se existe um algoritmo que enumera todas as demonstrações de fórmulas de T a partir dos axiomas. Uma teoria é *decidível* se o seu conjunto é recursivo. Uma teoria é *consistente* se para toda a fórmula x, não é verdade que x e $\neg x$ estejam ambas em T. Uma teoria é *integra* se todas as fórmulas de T são verdadeiras. Por exemplo, os axiomas da Aritmética de Peano (considerados como uma teoria da lógica de 1ª ordem) são uma axiomatização da teoria elementar dos números.

O corolário seguinte é uma versão do Teorema da Incompletitude de Gödel, isto é, que um sistema formal que contenha a aritmética de Peano ou é inconsistente ou contém teoremas (fórmulas verdadeiras) que não podem ser demonstrados no sistema.

Corolário 4.4.1 *Existe um conjunto B r.e. com um complementar infinito, tal que para toda a teoria T consistente e axiomatizável existe apenas um número finito de n's tal que a fórmula $n \notin B$ é verdadeira e demonstrável em T. (Mas, com um número finito de excepções um número infinito dessas fórmulas são verdadeiras).*

Dem. Seja B o conjunto do Teorema 4.4.3, alínea (iii), e seja $\overline{B}$ o seu complementar. O conjunto $D \subseteq \overline{B}$ dos elementos n que possuem uma demonstração em T que pertencem a $\overline{B}$ é r.e. Como B é *simples*, $\overline{B}$ não contém um conjunto infinito r.e. Então, D é finito. □

Exemplo 23 (Chaitin) *Seja T uma teoria integra axiomatizável cujos axiomas e regras de inferência podem ser descritas com cerca de k bits. Então, T não pode ser usado para demonstrar a aleatoriedade de nenhum número com mais de k bits. Se o sistema pudesse demonstrar a aleatoriedade para um número com muito mais de k bits então a primeira dessas demonstrações (na enumeração de todas as demonstrações obtidas por aplicação repetida dos axiomas e regras de inferência) podia ser usada para obter uma contradição: um programa com aproximadamente k-bits que produzia o número aleatório, referido na demonstração, um número para o qual, por hipótese, o menor programa que o produzia tinha muito mais que k bits. Formalmente,*

- *Seja T uma teoria axiomatizável (isto é r.e.) que pode ser descrita com k bits: $K(T) \leq k$*
- *Suponhamos que todas as fórmulas de T são verdadeiras (no modelo dos número naturais)*
- *Seja $S_c(x)$ a fórmula com o significado*

  x é lexicograficamente a menor sequência de tamanho c com $K(x) \geq c$

  Tem-se que $K(S_c) \leq \log c$ a menos uma constante fixa independente de T e de c.

Para cada c, existe um x tal que $S_c(x) = \textbf{true}$ é uma afirmação verdadeira. E S_c garante que esse x é único. Combinando as descrições de T e de S_c obtemos uma descrição para x. Nomeadamente, para cada candidato y de tamanho c, podemos decidir se $S_c(y) = \textbf{true}$ ou se $\neg S_c(y) = \textbf{true}$ por enumeração de todas as demonstrações em T. Para distinguir as descrições temos de codificar T numa versão auto-delimitada, isto é, em no máximo $2k$ bits. Então para uma constante independente de T e de c temos $K(x) \leq 2k + \log c + c'$, o que contradiz $K(x) \geq c$ para todos os $c > c_T$, com $c_T = 3k + c''$.

Enumerabilidade de conjuntos e complexidade de Kolmogorov

Usando a complexidade de Kolmogorov podemos quantificar a distinção entre conjuntos r.e. e recursivos.

Definição 4.4.1 *A* sequência característica *dum conjunto $A \subseteq \mathbb{N}$ é uma sequência binária infinita $\chi = \chi_1\chi_2\ldots$ definida por*

$$\chi_i = \begin{cases} 1 & se\ i \in A \\ 0 & caso\ contrário \end{cases}$$

Se A e $\overline{A}$ são r.e então a função $f(i) = \chi_i$ é recursiva e a complexidade condicional $K(\chi_{1:n}|n)$ é limitada por uma constante fixa para todo o n, onde $\chi_{1:n}$ representa a subsequência entre χ_1 e χ_n. Contudo, no caso geral, $K(\chi_{1:n}|n)$ pode crescer ilimitadamente com n, embora logaritmicamente. Isto é, estas sequências são "pouco" aleatórias.

Teorema 4.4.4 *(Lema de Barzdin)*

(i) Qualquer sequência característica χ de um conjunto r.e. A satisfaz $K(\chi_{1:n}|n) \leq \log n + c$ para todo o n, onde c é uma constante que depende de A (mas não de n).

(ii) Mais ainda, existe um conjunto r.e. tal que a sua sequência característica χ satisfaz $K(\chi_{1:n}) \geq \log n$, para todo o n.

Dem. (i) Como A é r.e existe uma função parcial recursiva ϕ tal que A é o domínio de definição de ϕ, isto é, $A = \{x : \phi(x) \downarrow\}$. Usando o método de "dovetail" podemos calcular $\phi(1)$, $\phi(2), \ldots$. Desde modo temos uma enumeração de A pela ordem em que as computações do $\phi(i)$-ésimo termina. O prefixo $\chi_{1:n}$ pode ser reconstruído a partir do número m de 1's que contém. Conhecendo m, podemos usar ϕ para enumerar os elementos de A até ter encontrado m elementos menores ou iguais a n. Seja $B = \{a_1, a_2, \ldots, a_m\}$ o conjunto desses elementos. Então, de B podemos reconstruir todos os 1's de $\chi_{1:n}$ e os restantes elementos de $\chi_{1:n}$ são 0's. Isto é, podemos descrever $\chi_{1:n}$ a partir duma descrição de ϕ e de m. Sendo $m \leq n$, vem $K(m) \leq \log n + c$ e como $K(\phi) < \infty$ temos o resultado.

(ii) Seja U a máquina de referência universal. Defina-se $\chi = \chi_1\chi_2\ldots$ por

$$\chi_i = \begin{cases} 1 & \text{se } U(\langle i, i\rangle) = 0 \\ 0 & \text{se } U(\langle i, i\rangle) \neq 0 \text{ ou } U(\langle i, i\rangle) \uparrow \end{cases}$$

interpretando $U(\langle i, i\rangle)$ como o cálculo de $\{i\}(i)$. Então, χ é a sequência característica dum subconjunto de $\mathbb{N}$ r.e. (Qual?). Vamos provar que χ verifica a propriedade enunciada. Suponhamos que $K(\chi_{1:n}) < \log n$ para algum n. Seja p o menor programa que calcula $\chi_{1:n}$ com tamanho menor que $\log n$. Logo, $p < n$. Então $U(\langle p, p\rangle) = \chi_p$ o que contradiz a definição de χ_p.

□

4.5 Informação Algorítmica

Se a complexidade condicional $K(x|y)$ é muito menor que $K(x)$, isso significa que y contém informação sobre x

Definição 4.5.1 *A* informação algorítmica *de x contida em y é:*

$$I(y : x) = K(x) - K(x|y)$$

Considerando a máquina de referência U tal que $U(\langle \epsilon, x\rangle) = x$ então

$$K(x|x) = 0$$

e

$$I(x : x) = K(x)$$

Isto é, dado que esta igualdades são válidas a menos uma constante aditiva, independente de x, $K(x)$ pode ser vista como informação (algorítmica) contida no próprio x. Como já se disse, esta definição da quantidade de informação tem a vantagem de se referir a objectos individuais, e não a objectos considerados como elementos dum conjunto de objectos com uma distribuição de probabilidade (Teoria da Informação de Shannon).

Contudo, ao contrário do que sucede na Teoria da Informação clássica, as igualdades $I(x : y) = I(y : x)$ e $K(x, y) = K(x) + K(y|x)$ só se verificam a menos um factor logarítmico.

Teorema 4.5.1 *(Komolgorov) Para todo os x, $y \in \mathbb{N}$,*

$$K(x, y) = K(x) + K(y) + O(min\{\log K(x), \log K(y|x)\})$$

Como $K(x, y) = K(y, x)$ a menos uma constante aditiva tem-se

Corolário 4.5.1 *A menos um termo aditivo* $O(min\{\log K(x), \log K(y)\})$,

$$K(x) - K(x|y) = K(y) - K(y|x)$$

e portanto,

$$|I(x:y) - I(y:x)| = O(min\{\log K(x), \log K(y)\})$$

Exemplo 24 *Para cada* n *existe uma sequência* x *de tamanho* n *tal que* $K(x|y) \leq n$. *Analogamente existe uma infinidade de valores* n *tal que* $K(n) \geq |n|$. *Escolhendo* x *tal que o seu tamanho* n *é aleatório (naquele sentido), temos a menos constantes aditivas:*

$$\begin{aligned} I(x:n) &= K(n) - K(n|x) \geq |n| \\ I(n:x) &= K(x) - K(x|n) \leq n - n = 0 \end{aligned}$$

Isto mostra que $|I(x:y) - I(y:x)|$ *pode ser da ordem do logaritmo das complexidades de* x *e de* y.

4.6 Complexidade de Kolmogorov auto-delimitada

A complexidade algorítmica definida na secção anterior apresenta alguns problemas tanto do ponto de vista teórico como prático, o que levou ao seu refinamento (10 anos depois ...e em particular por Chaitin [Cha87]) designado por complexidade de Kolmogorov auto-delimitada, K'. No entanto, como as duas diferem de um factor logarítmico, para muitas aplicações é indiferente qual a versão a usar.

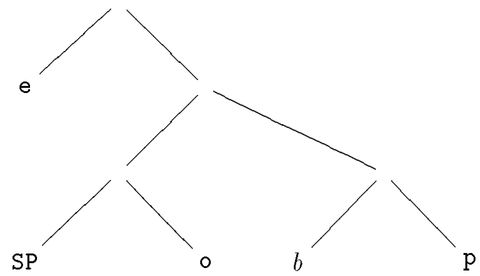
Definição 4.6.1 *Seja* Σ *um alfabeto. Uma linguagem* $A \subseteq \Sigma^\star$ *é* independente de prefixos *se nenhum elemento de* A *é prefixo de outro elemento de* A. *Uma codificação* $c : \Sigma^\star \longrightarrow \mathbb{N}$ *é uma* codificação prefixa *se o seu domínio é independente de prefixos.*

Exemplo 25 *O conjunto* $\{0^i 1 \;:\; i = 0, 1, 2, \ldots\}$ *é independente de prefixos. O único conjunto independente de prefixos que contém* ϵ *é* $\{\epsilon\}$.

Exemplo 26 *Seja* $c(0) = 1$, $c(10) = 2$, $c(110) = 3$ *e* $c(111) = 4$. *Então, o código* 0111100 *pode ser descodificado como* 1421 *duma maneira única!*

Exemplo 27 *O conjunto dos números de telefone (de alfabeto* $\{0, 1, \cdots, 9\}$*) é independente de prefixos. Não podem existir, por exemplo, telefones "5502631" e "550263".*

Exemplo 28 *As linguagens independentes de prefixos estão na base de um método de compressão de ficheiros que corresponde ao chamado código de Huffman. A cada caracter corresponde uma sequência de* $\{0, 1\}^\star$, *de modo a que aos caracteres mais frequentes no ficheiro correspondam sequências mais curtas. O algoritmo de Huffman examina o ficheiro a comprimir determinando a frequência relativa (número de ocorrências/número total de caracteres) de cada caracter existente. Em seguida constrói uma árvore binária óptima (ver, por exemplo, [Knu68]) – correspondente a um código que permite uma compressão máxima (utilizando este tipo de códigos!) do ficheiro. Por exemplo, se os caracteres que ocorrem forem* $\{\mathtt{SP}, \mathtt{b}, \mathtt{e}, \mathtt{o}, \mathtt{p}\}$ *(*SP *representa o caracter "espaço") e* *se as correspondentes frequências forem 0.15, 0.15, 0.4, 0.14, 0.16 uma árvore binária óptima é*



O código de um caracter é obtido a partir do caminho desde a raiz até ele, tomando-se um 0 cada vez que se segue pela sub-árvore esquerda e um 1 quando se segue pela sub-árvore direita. Neste exemplo, temos: $h(\texttt{SP}) = 100$, $h(\texttt{b}) = 110$, $h(\texttt{e}) = 0$, $h(\texttt{o}) = 101$, *e* $h(\texttt{p}) = 111$. *Note-se que o conjunto* $\{0, 100, 101, 110, 111\}$ *é independente de prefixos.*
Que representa a sequência 1110111010011010111010*1?*

Como é fácil de se verificar, uma vantagem destas linguagens é que uma qualquer concatenação de palavras pode ser descodificada de uma forma única nas palavras componentes: quando se concatenam palavras não se perde informação.

Uma codificação prefixa de $x \in \mathbb{N}$ é por exemplo 1^x0. Outros exemplos são as versões auto-delimitadas de x como $\overline{|x|}x$. A noção de auto-delimitação pode ser generalizada. Para $x \in \{0,1\}^\star \setminus \{\epsilon\}$, seja $E_0(x) = \overline{x}$ a versão auto-delimitada de ordem 0; $E_1(x) = \overline{|x|}x = x'$ a versão auto-delimitada de ordem 1 ("standard"); $E_i(x) = E_{i-1}(|x|)x$ para $i \geq 1$.

Exercício 4.6.1 Dado x, a versão auto-delimitada de ordem 0 pode ser $E_0(x) = 0^x1$. Obtenha $E_i(x)$ neste caso e determine um majorante das suas complexidades.

Uma das propriedades importantes das codificações (códigos) prefixas é a seguinte, denominada *Inequação de Kraft*:

Teorema 4.6.1 *Para qualquer sequência finita ou infinita de número naturais* l_1, l_2, $\ldots$, *existe uma codificação prefixa* c *tal que esta sequência corresponde aos tamanhos das sequências no domínio de* c *se e só se*

$$\sum_n 2^{-l_n} \leq 1$$

Dem. ($\Rightarrow$) Considere-se a injecção entre sequências binárias x e os intervalos reais, $\Gamma_x = [0.x, 0.x + 2^{-|x|}[$ em $[0,1[$. O comprimento de cada intervalo é $2^{-|x|}$. Basta ver que uma codificação prefixa corresponde a um conjunto de intervalos daquela forma disjuntos. Isto prova que a inequação se verifica para codificações prefixas.

($\Leftarrow$) Suponhamos que $l_1, l_2, \ldots$ verificam a inequação. Podemos supor que a sequência é não decrescente. Escolham-se intervalos disjuntos adjacentes I_1, I_2, $\ldots$ de comprimentos 2^{-l_1}, 2^{-l_2}, $\ldots$, a partir do extremo inferior de $[0,1[$. Deste modo, para cada $n \geq 1$ o extremo inferior de I_n é $\sum_{i=1}^{n} 2^{-l_i}$ e o extremo superior de I_n coincide com o extremo inferior de I_{n+1}. Dado que os l_i são não decrescentes, cada intervalo $I_n = \Gamma_x$ para alguma sequência binária x com tamanho $|x| = l_n$. Tome-se $c(x) = n$ para cada I_n. $\square$

Exercício 4.6.2 Construa uma codificação prefixa associada à sequência $2, 3, 6, 10$ e verifique a Inequação de Kraft.

Exemplo 29 *O Teorema 4.6.1 pode-se enunciar em termos de linguagens independentes de prefixos. Para cada* $L \subseteq \{0,1\}^\star$ *define-se* $FI(L) = \sum_{w \in L} 2^{-|w|}$*, onde para cada* $w \in L$*,* $2^{-|w|}$ *é o indicador de frequência (ou a probabilidade) de* w*. Se* L *é independente de prefixos então* $FI(L) \leq 1$*.*

Por exemplo, para $\{0^i 1 \;:\; i = 0, 1, 2, \ldots\}$ *temos*

$$FI(L) = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 1$$

Voltemos às complexidades ...

A complexidade de Kolmogorov auto-delimitada requere que os programas sejam codificações prefixas. Isto é, as máquinas de Turing usadas e nomeadamente a máquina de Turing de referência U, só aceitam programas p tal que nenhum prefixo de p é um programa aceite por essas máquinas. Isto é, o conjunto de programas para os quais estas máquinas param é livre de prefixos. Formalmente,

Definição 4.6.2 *Uma função* $\phi : \Sigma^\star \longrightarrow \mathbb{N}$ *parcial recursiva é* prefixa *se* $\phi(x)\downarrow$ *se e só se* $\phi(y)\uparrow$ *para todo o* y *prefixo próprio de* x*.*

Exemplo 30 *Pode-se obter uma enumeração das funções recursivas prefixas usando o algoritmo a seguir apresentado. Considere-se uma enumeração de máquinas de Turing* T_1*,* T_2*,.... Vamos considerar uma enumeração apenas das máquinas de Turing que calculam funções prefixas. Seja* T *uma máquina de Turing tal que os seus dados podem ser apenas lidos da esquerda para a direita sem nunca voltar a trás. Suponhamos que* $x = b_1 b_2 \ldots$ *são os seus dados. Então, modificámos* T *para executar o seguinte algoritmo:*

Passo 1: *Seja* $x := \epsilon$

Passo 2 *Consideram-se as computações de* T *pelo método de "Dovetail" de* $\phi(xy)$ *para todo o* $y \in \Sigma^\star$*. Se* $\phi(xy)\downarrow$ *e é a primeira computação que pára então vá para* **Passo 3.**

Passo 3: *Se* $y = \epsilon$ *então produza* $\phi(x)$ *e pare; senão leia o bit seguinte de* x*;* $x := xb$*; vá para* **Passo 2***.*

Esta construção produz uma enumeração das máquinas T_1'*, ... que calculam funções recursivas prefixas. (Verifique!)*

Exemplo 31 *Uma forma alternativa de definir uma máquina de Turing* prefixa *é a seguinte: Considere-se a classe de máquinas de Turing* PM *com uma fita semi-infinita de leitura, uma fita semi-infinita de escrita e uma fita duplamente infinita de "trabalho". Suponhamos que a fita de leitura apenas contém* 0*'s e* 1*'s (não contém brancos). No estado inicial os dados são escritos na fita de leitura, e a cabeça de leitura encontra-se na primeira célula. Inicialmente as restantes fitas apenas contêm brancos. Cada passo da computação — transição da máquina de Turing consiste numa de três acções:*

- *A máquina lê o símbolo seguinte da fita de dados movendo a cabeça da fita de dados de uma unidade para a direita e alterando o seu estado de acordo com o símbolo lido.*
- *A máquina escreve um símbolo (dependente do seu estado) na fita de resultados, movendo a cabeça da fita de resultados de uma unidade para a direita.*

– *A máquina executa uma transição relativa à fita de trabalho semelhante às transições nas máquinas de uma só fita.*

Quando a máquina parar o resultado x é o conteúdo da fita de escrita. Denominamos esta máquinas de Turing auto-delimitadas, porque o conjunto de dados para as quais cada máquina pára é independente de prefixos. Uma sequência p é um programa para T se T pára com a cabeça de leitura no bit mais à direita de p (isto é, p é um prefixo maximal duma sequência infinita de dados). Qualquer programa da forma px é exactamente equivalente ao programa p.

Exercício 4.6.3 Mostrar que a classe de máquinas de Turing auto-delimitadas PM, definidas no exemplo anterior, calculam precisamente o conjunto de funções parciais recursivas prefixas.

O Teorema da Invariância continua válido se nos restringirmos a este tipo de máquinas.

Exercício 4.6.4 Mostre a afirmação anterior.

Podemos então fixar uma máquina de Turing U' auto-delimitada de referência. A *complexidade auto-delimitada (prefixa)* de x é definida por $K'(x) = K_{U'}(x)$ isto é o tamanho do menor programa p de U' tal que $U'(p) = x$. Analogamente se definem complexidades auto-delimitadas condicionais.

Sempre que for explícito pelo contexto, qual a complexidade ou qual a máquina de referência a que nos referimos, designaremos estes valores apenas por K ou U.

A complexidade auto-delimitada K' verifica várias propriedades sem o factor logarítmico que afectava os resultados para K.

Exemplo 32 *Seja $K'(x,y) = K'(\langle x,y\rangle)$. Neste caso, como sabemos que os programas são auto-delimitados (as codificações são prefixas), podemos concatenar descrições sem ter de marcar as suas extremidades. Então,*

$$K'(x,y) \leq K'(x) + K'(y) + O(1)$$

Nomeadamente, seja U' a máquina de referência. Sejam $U'(x^\star) = x$ com $K'(x) = |x^\star|$ e $U'(y^\star) = y$ com $K'(y) = |y^\star|$. Seja V uma máquina universal auto-delimitada que primeiro lê $x^\star$ e calcula x, depois lê $y^\star$ e calcula y e finalmente calcula $\langle x,y\rangle$. Seja $n = e(V)$ para uma enumeração de máquinas de Turing auto-delimitadas. Então U' com dados $0^n1x'y'$ calcula $\langle x,y\rangle$.

Exemplo 33 *As funções K e K' são assimptoticamente iguais. Para todo o x, y tem-se a menos duma constante aditiva*

$$K(x|y) \leq K'(x|y) \leq K(x|y) + 2\log K(x|y)$$

A primeira inequação é trivial (Porquê?). Seja $K(x|y) = |p|$ tal que U calcula x com dados $\langle y,p\rangle$. A codificação $p' = \overline{|p|}p$ é uma codificação auto-delimitada de p. Por isso,

$$K'(x|y) \leq K(x|y) + 2|K(x|y)| + O(1)$$

Em consequência do Teorema 4.3.1 podemos concluir que para todo o x, $|x| = n$

$$K'(x) \leq n + 2\log n + O(1)$$

embora usando outras codificações prefixas este majorante possa ser melhorado.

O Teorema da Incompressibilidade pode-se enunciar do seguinte modo:

Teorema 4.6.2 *(i)* $\forall n, max\{K'(x) \,:\, |x| = n\} = n + K'(n) + O(1)$

(ii) $\forall c, \#\{x : |x| = n \text{ e } K'(x) \leq n + K'(n) - c\} \leq 2^{n-c+O(1)}$

Diz-se que $x \in \Sigma^\star$ com $|x| = n$ é incompressível se $K'(x) \geq n$.

Exercício 4.6.5 Relacione as noções de incompressibilidade para K e K'.

Teorema 4.6.3 *A menos duma constante aditiva tem-se*

$$K'(x, y) = K'(x) + K'(y|(x, K'(x)))$$

e

$$K'(x, K'(x)) = K'(x)$$

Então ,

Teorema 4.6.4 *A menos duma constante aditiva tem-se*

$$K'(y) - K(y|(x, K'(x))) = K'(x) - K(x|(y, K'(y)))$$

isto é,

$$I'(\langle x, K'(x)\rangle : y) = I'(\langle y, K'(y) >: x) + O(1)$$

O problema da paragem resolvido a partir de $K'(x)$

Vamos ver que o conhecimento de $K'(x)$ permitiria resolver o problema da paragem ("halting problem"); uma consequência deste facto é que a função $K'(x)$ não é computável. Comecemos pelo seguinte Lema.

Lema 4.6.1 *Seja p um programa com n bits.*

a) Se o programa pára, o tempo t que demora a parar satisfaz (onde c é uma constante)

$$K(t) \leq n + c$$

b) Como consequência, se $t \geq T$ implica $K'(t) > n + c$, e se o programa já correu o tempo T sem parar, então o programa nunca vai parar.

Dem. a) Consideremos uma máquina de Turing M, semelhante à máquina universal U, mas com as seguintes diferenças: M conta o número t de passos que executa e no fim (se parar) dá t como resultado. A simulação de M em U permite calcular t com $n + c$ bits.

b) Se o programa parasse, tinhamos uma descrição de $t \geq T$ com $n+c$ bits o que contraria $K(t) > n + c$.

□

Seja n um inteiro e consideremos uma palavra x_{max} de comprimento n com complexidade K' máxima (isto é, se $|x| = n$, então $K'(x) \leq K'(x_{max})$). Sabemos do Teorema 4.6.2 que

$$K'(x_{max}) = n + K'(n) + O(1)$$

donde

$$K'(x_{max}) > n + K'(n) - c'$$

para uma constante apropriada c'.

Vamos agora procurar um majorante de $K'(x_{max})$. Seja n um inteiro fixo. O conjunto dos majorantes de $K'(x)$ com $|x| = n$ é recursivamente enumerável. A prova deste facto pode basear-se na utilização da técnica do "dovetailing" para a máquina de referência U e para todos os x de comprimento n e programas possíveis (λ, 0, 1, 00,...). Quando (e se) a máquina simulada parar para um certo programa p com resultado x então temos um majorante, $K'(x) \leq |p|$.

Seja então dado o inteiro n e consideremos a enumeração referida. Se $K'(x)$ fosse computável, seria possível terminar a computação quando todos os valores de $K(x)$ para x de comprimento n (em particular para x_{max}) tivessem sido atingidos. Seja $\beta(n)$ o correspondente tempo. A partir de n e de $\beta(n)$ ou de qualquer valor $t \geq \beta(n)$ seria possível determinar x_{max}. Teríamos (recordemos que estaamos a utilizar complexidades auto-delimitadas)

$$K'(x_{max}) \leq K'(t) + K'(n) + c''$$

Obtemos então

$$n + K'(n) - c' < K'(x_{max}) \leq K'(t) + K'(n) + c''$$

donde

$$K'(t) > n - c' - c''$$

Concluimos que, se $t \geq \beta(n)$ então $K'(t) > n - c' - c''$. Poderíamos então utilizar $\beta(n)$ (que, como vimos, seria computável se K' o fosse) e o Lema anterior para decidir da seguinte forma o problema da paragem: um programa p com n bits pára sse pára antes do tempo $\beta(n)$.

4.7 Aplicações da Compressibilidade

Sendo ω uma sequência infinita de 0's e 1's denota-se por $\omega_{1:n}$ o seu prefixo de tamanho n. Diz-se que ω é *aleatória* se

$$\exists c\ \forall n K'(\omega_{1:n}) \geq n - c$$

Um número real $r \in \mathbb{R}$ pode ser visto como a sequência, possivelmente infinita, dos seus dígitos (numa dada base). Dizemos que um número é *computável* se existe um algoritmo que permite calcular a sua sequência de dígitos. Formalmente uma sequência infinita ω é *recursiva* sse existe uma função recursiva $f : \mathbb{N} \longrightarrow \Sigma^\star$ tal que $\omega_{1:n} = f(n)$. Pode-se mostrar que ω é recursiva sse $K(\omega_{1:n}) \leq K(n) + c$. Neste sentido podemos concluir que não existem números reais computáveis que sejam aleatórios (isto é que a sua sequência de dígitos seja aleatória).

Isto aplica-se por exemplo a números como e ou π. Em conclusão, não devíamos usar a sequência de dígitos de π como base dum gerador de números aleatórios. Uma outra caracterização de números aleatórios é o de serem *normais*, isto é, se, para cada $n \geq 1$, qualquer sequência de dígitos de 0 a 9 (se a base for 10) ocorrer com a mesma probabilidade assimptótica; por exemplo, a probabilidade da sequência "`87`" é, no limite, 1/100. Nenhum número racional é *normal* e números como o e ou o π não se sabe. Contudo, existem números normais (na base 10) que não são "nada" aleatórios. Um desses números é o número de Champernowne:

$$0.1234567891011121311415\ldots$$

Contudo este número não é normal, por exemplo na base 2.

Por outro lado, quase todos os números reais são aleatórios (no sentido de Kolmogorov) (Porquê?). Mas como obter um tal número?

Considere-se o seguinte número não computável. Seja U uma máquina universal de Turing e $T_1, T_2, \ldots$ uma enumeração de máquinas de Turing. Define-se $k = 0.k_1k_2\ldots$ tal que $k_i = 1$ se T_i

pára com a fita vazia e $k_i = 0$ caso contrário. Pela insolubilidade do problema da paragem k não é computável. Vamos ver que também não é aleatório. O conjunto $B = \{i : U(T_i) \downarrow\}$ é r.e., logo, pelo Lema de Barzdin, k é compressível: cada prefixo $k_{1:n}$ de tamanho n, para todo $n \geq n_0$, pode ser calculado por uma sequência de tamanho no máximo $2 \log n$ (Verifique porquê!).

O número Ω

Considerando a máquina universal de referência, para cada $x \in \Sigma^\star$ podemos definir a sua *probabilidade* por $m(x) = \sum_{U'(p)=x} 2^{-|p|}$

A função $m(x)$ pode ser interpretada como a probabilidade de U' parar e produzir x, tendo como dados um programa p obtido por $|p|$ lançamentos de uma moeda ao ar.

Lema 4.7.1 *Para todo o* $x \in \Sigma^\star$

(i) $m(x) \geq 2^{-K'(x)}$

(ii) $0 < m(x) < 1$

(iii) $0 < \sum_{x \in \Sigma^\star} m(x) < 1$

Dem. A alínea (i) verifica-se porque $2^{-K'(x)}$ é um dos termos de $m(x)$. Pela inequação de Kraft e porque $L = \{p : U'(p) = x\}$ é uma linguagem livre de prefixos, tem-se que $m(x) \leq 1$. O mesmo argumento se aplica para o limite superior de (i) Como todos os termos dos somatórios são não negativos tem-se $m(x) \geq 0$ e analogamente para (iii). A primeira inequação de (ii) é estrita por (i) e a segunda inequação é estrita porque $\sum_{x \in \Sigma^\star} m(x) \leq 1$. A segunda inequação de (iii) é estrita porque existem programas para os quais U' não pára. □

Em particular pode-se provar que $K'(x) = -\log m(x) + O(1)$ (Solomonoff-Levin).

O número Ω denominado *probabilidade da paragem* da máquina de referência U' (ou "número mágico") é definido por:

$$\Omega = \sum_{U'(p)\downarrow} 2^{-|p|} = \sum_x m(x)$$

Pelo Lema 4.7.1:

$$0 < \Omega < 1$$

Seja

$$\Omega = 0.b_1 b_2 \ldots$$

a representação de Ω na base[1] 2. Denote-se por $B = b_i b_2 \ldots$ a sequência (possivelmente infinita ...) dos seus bits e $B_{1:n} = b_1 \ldots b_n$ o seu prefixo de tamanho n, para $n \geq 1$. Para todo $n \geq 1$ define-se o número racional

$$\Omega_n = 0.b_1 \ldots b_n$$

O número Ω tem as seguintes propriedades e as seguintes implicações para sistemas formais:

- Ω codifica o problema da paragem.
- B é aleatória.
- Qualquer sistema formal axiomático apenas pode produzir um número finito de bits de Ω.

[1] Se Ω for racional considera-se uma representação com um número infinito de zeros

- Para todo o sistema formal axiomático T que possa ser expresso em n bits, $B_{1:n}$ pode ser usada para decidir se uma qualquer fórmula bem formada, em T é um teorema, um não teorema ou independente.
- Um sistema formal só pode deduzir $B_{1:n}$ se $B_{1:n}$ for dado como axioma.
- Em conclusão, podemos definir Ω formalmente mas não existe nenhum algoritmo que permita calcular uma infinidade de bits de Ω.
- Existe uma equação diofantina $A(n, x_1, \ldots, x_m) = 0$ que tem um número infinito de soluções se e só se o n-ésimo bit de Ω é 1.
- Podemos "codificar" Ω na teoria elementar dos números.
- A aleatoriedade é inerente à Matemática.

O conjunto $D = \{p \ : \ U'(p) \downarrow\}$ é r.e.. Consideremos uma enumeração $g : \mathbb{N} \longrightarrow \Sigma^\star$ de D obtida pelo método de "dovetail" das possíveis computações. Recapitulando, para $i = 1, \ldots$ a fase i consiste em U' executar o j-ésimo passo de computação nos k-ésimos dados, para todos os j e k tal que $j + k = i$. Define-se $g(1)$ como o primeiro programa para o qual a computação de U' pára, $g(2)$ o segundo, ...

Para todo o $i \geq 1$ seja

$$\omega_i = \sum_{j=1}^{i} 2^{-|g(j)|}$$

É óbvio que a sucessão ω_i é monótona crescente e $\lim_{i\to\infty} \omega_i = \Omega$

Lema 4.7.2 *Se $\omega_i > \Omega_n$ então*

$$\Omega_n < \omega_i < \Omega \leq \Omega + 2^{-n}$$

Dem. A última inequação resulta de que

$$2^{-i} \geq \sum_{j=i+1}^{\infty} b_j 2^{-j}$$

para quaisquer $b_j \in \{0, 1\}$. $\square$

Teorema 4.7.1 *Dado um qualquer n, se se conhecer $B_{1:n}$ então é possível decidir o problema da paragem para qualquer programa p com tamanho $|p| \leq n$.*

Dem. Conhecendo $B_{1:n}$ pode-se construir Ω_n. Determina-se i tal que ω_i verifica $\omega_j > \Omega_n$. Pelas propriedades da sucessão ω_i, tal i existe sempre. Seja p_1 um programa tal que $|p_1| = n_1 \leq n$. Então $U'(p_1) \downarrow$ se e só se p_1 é um dos $g(1), \ldots, g(i)$. Que a condição é necessária é trivial. Suponhamos que $U'(p_1) \downarrow$ mas que $p_1 = g(m)$ e $m > i$. Então

$$\Omega > \omega_m \geq \omega_i + 2^{-n_1} \geq \omega_i + 2^{-n} > \Omega_n + 2^{-n} \geq \Omega$$

Absurdo! $\square$

Pelo Teorema 4.7.1, o conhecimento dum prefixo suficientemente grande $B_{1:n}$ permite resolver qualquer problema da paragem e n pode ser determinado pelo . Em particular, permitiria resolver qualquer problema que possa ser refutado por contra-exemplos finitos, o que é o caso de muitas das conjecturas em matemática.

Exercício 4.7.1 O Teorema 4.7.1 contradiz a indecidibilidade do problema da paragem? Justifica a resposta. Porque é que Ω é não computável?

Teorema 4.7.2 *A sequência B é aleatória.*

Dem. Recorde que B é aleatória se

$$\exists c \forall n \; K'(B_{1:n}) \geq n - c$$

Dado $B_{1:n}$ é possível calcular todos os programas p de comprimento não superior a n para os quais U' converge. Então,

$$\{U'(g(j)) \; : \; 1 \leq j \leq i \text{ e } |g(j)| \leq n\} = \{x \; : \; K'(x) \leq n\}$$

Se x não pertence a este conjunto, tem-se necessariamente que $K'(x) > n$. Então é possível construir uma função recursiva ϕ tal que $\phi(B_{1:n}) = x$. Vem

$$K'(\phi(B_{1:n})) \leq n$$

Dada uma descrição de ϕ em c bits, para cada n pode-se calcular $\phi(B_{1:n})$ de $B_{1:n}$, o que significa que:

$$K'(B_{1:n}) + c \leq n$$

□

Exercício 4.7.2 Diga como se pode construir a função ϕ referida na demonstração anterior. Sugestão: Sendo $x = x_1 \ldots x_t$ considere m o menor inteiro, se existir algum, tal que $\omega_m = \sum_{j=1}^{t} x_j 2^{-j}$. Se tal m existir considere $f(x)$ a primeira sequência que não pertence a $\{g(j) \; : \; 1 \leq j \leq m\}$.

Teorema 4.7.3 *(Chaitin,[Cha87]) Existe uma equação diofantina $A(n, x_1, \ldots, x_m) = 0$ que tem um número infinito de soluções $x_1, \ldots, x_m$ se e só se o n-ésimo bit de Ω é 1.*

Chaitin [Cha87, Cha94] construiu explicitamente a equação mencionada no teorema anterior, com cerca de 17000 variáveis.

Exercício 4.7.3 Justifica as implicações da existência do número Ω acima referidas.

4.8 Minorantes de Complexidade

A existência de sequências incompressíveis, aleatórias segundo Kolmogorov, permite estabelecer minorantes da complexidade de vários problemas e algoritmos. As demonstrações por contagem para estabelecer minorantes, envolvem todas as instâncias (sequências dum dado tamanho) duma classe de problemas e prova-se que o minorante deve verificar-se para algumas dessas instâncias.

Numa demonstração típica, usando o método da incompressibilidade, escolhe-se uma sequência aleatória duma dada classe (que existe, mas que não se pode exibir...). Essa sequência é incompressível. Depois prova-se que a propriedade desejada (por exemplo, um minorante de complexidade) é verificada por essa sequência. O argumento é que se a propriedade não se verificasse, então essa sequência era compressível.

Dado que apenas se tem de manipular um objecto, as demonstrações são normalmente mais simples e curtas.

Vamos apenas considerar três exemplos.

4.8.1 Máquina de Turing com uma fita

Uma máquina de Turing M com uma fita semi-infinita necessita de $\Omega(n^2)$ passos para reconhecer $L = \{xx^R : x \in \Sigma^\star\}$, isto é, a sua complexidade temporal $T_M(n)$ é $\Omega(n^2)$.

Já vimos que existe uma máquina de Turing M que reconheça L em $O(n^2)$ passos. Vamos ver que qualquer máquina de Turing deste tipo necessita $\Omega(n^2)$ passos para reconhecer L.

Durante uma computação de M podemos associar a cada célula da fita[2] c_i uma *sequência de passagem* $sp_i = (q_1, \ldots, q_m)$ constituída pelos estados de M nos passos em que a cabeça passa dessa célula para a célula à sua direita, primeiro da esquerda para a direita, e depois alternadamente em ambas as direcções. Denota-se por $|sp_i|$ o tamanho duma sequência de passagem para a célula c_i.

A demonstração é feita por redução ao absurdo. Suponhamos que $T_M(n)$ é $o(n^2)$. Suponhamos que M pára na primeira célula depois dos dados (marcador). Seja $|M|$ o tamanho duma descrição de M. Fixemos x uma sequência de tamanho $|x| = n$ tal que $K(x|(M,n)) \geq n$. Pelo Teorema da Incompressibilidade x existe. Consideremos uma computação de M com dados $x0^{2n}x^R$. Se cada sequência de passagem associada a uma célula no segmento 0^{2n} é maior que $n/(2|M|)$, então M executa pelo menos $n^2/|M|$ passos. Caso contrário, existiria uma sequência de passagem sp_i de tamanho menor que $n/(2|M|)$, associada a uma célula c_i. Esta sequência de passagem sp_i é completamente descrita por no máximo $n/2$ bits[3]. Usando sp_i, é possível reconstruir x por verificação exaustiva de todas as sequências binárias de tamanho n. Para cada y sequência candidata de tamanho n, coloca-se a sequência $y0^{2n}$ nas primeiras $3n$ células mais à esquerda da fita e simula-se a computação de M desde o estado inicial. Cada vez que a cabeça passa da célula c_i para a célula à sua direita, interrompe-se a computação de M com a cabeça à direita de c_i e recomeça-se a computação q obtido de sp_i e com a cabeça da fita na célula c_i. Assim M não executa "realmente" nenhum movimento à direita de c_i

Suponhamos que nesta computação com y, cada vez que a cabeça se movimenta de c_i para a sua direita, o estado corrente de M é o estado seguinte indicado por sp_i. Então M aceita os dados $y0^{2n}x^R$. Isto é, a computação à direita da célula c_0 é idêntica à computação correspondente com dados $x0^{2n}x^R$. Dado que M pára à direita de c_0, ou aceita ambas as sequências $y0^{2n}x^R$ e $x0^{2n}x^R$ ou rejeita as duas. Como M aceita L tem-se que $y = x$. Assim, dado sp_i podemos reconstruir x com um programa cujos dados são a informação de M, n e sp_i. Isto significa que

$$K(x|(M,n)) \leq |sp_i| + O(1) \leq n/2 + O(1)$$

o que contradiz $K(x|(M,n) \geq n$, para n grande.

4.8.2 Linguagens Regulares

Recorde que uma linguagem é regular se e só se for aceite por um autómato finito, $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ onde $\delta : \Sigma \times Q \longrightarrow Q$ é a função de transição e $F \subseteq Q$ são os estados finais. A linguagem aceite por $\mathcal{A}$ é $L = \{x : \delta'(x, q_0) \in F\}$, onde δ' é δ estendida a $\Sigma\star$ por $\delta'(q, ax) = \delta'(q', x)$ se $\delta(q, a) = q'$ e $\delta'(q, \epsilon) = q$. Usando a complexidade de Kolmogorov vamos mostrar que $L = \{0^k 1^k : k \geq 1\}$ não é regular. Suponhamos que era regular. Fixe-se k com $K(k) \geq \log k$. Seja q o estado dum autómato $\mathcal{A}$ que aceite L, depois de processar 0^k. Então q e A são uma descrição de k. Nomeadamente, executando $\mathcal{A}$ iniciando em q e com uma sequência de 1's, $\mathcal{A}$ chega a um estado de aceitação exactamente após k uns consecutivos. A descrição de $\mathcal{A}$ e q estão limitados por uma constante, seja

[2] que contém inicialmente os dados

[3] Note que cada estado é descrito com menos de $|M|$ bits!

c, independente de k. Então, $K(k) \leq c + O(1)$ o que contradiz a hipótese, para k suficientemente grande.

O lema seguinte generaliza este exemplo

Lema 4.8.1 *(Regularidade-KC) Seja $L \subseteq \Sigma^\star$ uma linguagem regular e $L_x = \{y \ : \ xy \in L\}$. Existe uma constante c, tal que para cada x, se y é a n-ésima*[4] *sequência em L_x, então $K(y) \leq K(n) + c$*

Dem. Seja L regular. A n-ésima sequência y tal que $xy \in L$ pode ser descrita por:

- por pertencer a L_x e pela descrição do autómato finito que descreve L
- o estado do autómato depois de processar x e pelo número n

O primeiro item requere $O(1)$ bits. Então $K(y) \leq K(n) + O(1)$ □

Exercício 4.8.1 Prove que $\{1^p \ : \ p \text{ é primo}\}$ não é regular.
Sugestão: considere $xy = 1^p$ com p o $k+1$-ésimo primo. Seja $x = 1^{p'}$ (do lema) com p' o k-ésimo primo. Então, $y = 1^{p-p'}$ e $n = 1$. Use o facto de que a diferença entre primos consecutivos crescer de forma não limitada.

Exercício 4.8.2 Prove que $\{0^i 1^j \ : \ i \neq j\}$ não é regular.
Sugestão: Considere $x = (0)^m$ e $K(m) \geq \log m$.

Exercício 4.8.3 Prove que $\{xx^R \ : \ x \in \Sigma^\star \setminus \{\epsilon\}\}$ não é regular.
Sugestão: Considere $x = (01)^m$ e $K(m) \geq \log m$. O primeiro elemento de L_x é $y = (10)^m 0$. Qual é $K(y)$?

Usando a complexidade de Kolmogorov pode ainda caracterizar-se completamente as linguagens regulares.

Definição 4.8.1 *Considere-se um enumeração de $\Sigma^\star$ (por exemplo, a lexicográfica) sendo y_i o seu i-ésimo elemento. Seja $L \subseteq \Sigma^\star$ e $x \in \Sigma^\star$, considere-se a sequência característica $\chi = \chi_1\chi_2\ldots$ de $L_x = \{y \ : \ xy \in L\}$, tal que $\chi_i = 1$ se $xy_i \in L$, caso contrário $\chi_i = 0$.*

Teorema 4.8.1 *(Caracterização de regularidade (KC)) Seja $L \subseteq \Sigma^\star$. Existe uma constante c_L dependente de L, tal que as seguintes afirmações são equivalentes:*

(i) *L é regular*

(ii) *para todo o $x \in \Sigma^\star$, para todo o n, $K(\chi_{1:n}|n) \leq c_L$*

(iii) *para todo o $x \in \Sigma^\star$, para todo o n, $K(\chi_{1:n}) \leq K(n) + c_L$*

(iv) *para todo o $x \in \Sigma^\star$, para todo o n, $K(\chi_{1:n}) \leq \log n + c_L$*

Exercício 4.8.4 *Mostre o Teorema anterior.*
Sugestão: Mostre que (i) $\to$ (ii) e (ii) $\to$ (iii) $\to$ (iv). A parte (iv) $\to$ (i) usa o facto de, para cada c, apenas um número finito de sequências infinitas terem para todo o n, $K(\chi_{1:n}) \leq \log n + c$ (porquê?). Considera-se ainda uma relação invariante à direita R tal que xRx' se $\chi = \chi'$ e usa-se o Teorema de Myhill-Nerode para linguagens regulares.

[4] na ordem lexicográfica

4.8.3 Análise da complexidade do caso médio do Heapsort

Seja $A[1] \ldots A[n]$ uma sequência de números inteiros a ordenar por ordem crescente. Uma árvore binária diz-se *essencialmente completa* se cada nó tiver dois filhos, excepto eventualmente um nó no nível 1 que só possui um filho à esquerda. Um `heap` é uma árvore binária essencialmente completa em que para cada nó interno o seu valor é maior ou igual aos valores dos seus filhos. Formalmente, uma sequência de valores $A[1], \ldots, A[n]$ é um `heap` se

$$A[\lfloor j/2 \rfloor] \geq A[j] \text{ para } 1 \leq \lfloor j/2 \rfloor < j \leq n$$

Qualquer sequência A pode ser vista como uma árvore, considerando a raiz $A[1]$ e para o nó $A[i]$ os seus filhos são $A[2i]$ e $A[2i+1]$, com $2i, 2i+1 \leq n$.

Na figura 4.1 apresenta-se um algoritmo de ordenação baseado na construção dum `heap`. A função `makeheap` constrói um `heap` na sequência $a[n]$. Se $a[n]$ for um `heap` no qual um elemento foi modificado (e cujo valor decresceu) a função `siftdown` permite tornar a de novo num `heap`. Esta função executa a seguinte operação: Se $a[i] < a[2i]$ troca-se esse nó pelo filho maior e continua-se esse processo descendo a árvore (`siftdown`). O algoritmo do `heapsort` tem complexidade no pior caso de $O(n \log n)$ sendo a complexidade da função `makeheap` de $O(n)$. Note-se que a profundidade dum `heap` é $\log n$.

Vamos ver que a complexidade no caso médio é também $O(n \log n)$. Suponhamos que as sequências de n valores são igualmente prováveis, isto é, que têm uma distribuição de probabilidade uniforme.

Teorema 4.8.2 *(I. Munro) Em média o* `Heapsort` *faz* $n \log n + O(n)$ *atribuições e* $2n \log n - O(n)$ *comparações.*

Dem. Vamos dar apenas uma ideia da demonstração.

Dados n valores as suas permutações são $n! \approx n^n e^{-n} \sqrt{2\pi n}$. Pelo Teorema da Incompressibilidade existe uma permutação p dos n valores, tal que

$$K(p|n) \geq \log n! - c \geq n \log n - n \log e \geq n \log n - 2n \tag{4.1}$$

Facto Seja h o `heap` construído por `makeheap` com dados p satisfazendo a equação (4.1). Então,

$$K(h|n) \geq n \log n - 6n \tag{4.2}$$

Dem. Suponhamos por absurdo que $K(h|n) < n \log n - 6n$. Então, vamos mostrar que se pode descrever p, usando h e n, com menos de $n \log n - 2n$ bits. Podemos codificar o processo de construção do `heap` h a partir de p. Em cada ciclo, quando o valor $v = A[i]$ é "empurrado" para baixo na árvore podemos memorizar o caminho que v percorre: 0 indica um ramo esquerdo, 1 um ramo direito e 2 significa que pára. No total este processo pode ser memorizado em $n \log 3 \sum_{i=1} \lfloor n/2 \rfloor i/2^{i+1} \leq 2n \log 3$ bits[5]. Dado o `heap` h final e a descrição dos caminhos, podemos inverter a função `makeheap` e reconstruir p. Então,

$$K(p|n) < K(h|n) + 2n \log 3 + O(1) < n \log n - 2n$$

Obtemos uma contradição. □

[5] Analisando a função `siftdown`

```
void swap (int a[],int i,int k){
  int aux;
    aux=a[k];
    a[k]=a[i];
    a[i]=aux;
}
/* ``peneira´´ o nodo i ate a heap estar restabelecida */
void siftdown(int a[],int n, int i){
  int k,j;
  k=i;
  do{
    j=k;
  if(2*j<= n && a[2*j]>a[k])
    k=2*j;
  if(2*j< n && a[2*j+1]>a[k])
    k=2*j+1;
  swap(a,j,k);
  }while (j!=k);
}

void makeheap(int a[],int n){
  int i;
  for(i=n/2;i>=1;i--){
    siftdown(a,n,i);
  }
}

/* para cada i a[1..i] nao esta ordenada e a[i-1...n] esta ordenada por
ordem crescente. A posição a[0] não é usada. */
void heapsort(int a[],int n) {
  int i;
  makeheap(a,n);
/* passo de ordenacao */
  for(i=n;i>=2;i--){
    swap(a,1,i);
    siftdown(a,i-1,1);
  }
}
```

Figura 4.1: `Heapsort`

Podemos descrever h usando a história dos $n-1$ rearranjos do `heap` durante o passo de ordenação do `heapsort`. Apenas necessitamos de guardar, para $i = n-1, \ldots, 2$ a posição final em que $A[i]$ é inserida. Essa posição pode ser codificada descrevendo o caminho da raiz até essa posição. O caminho pode ser representado por uma sequência binária, com 0 indicando um ramo à esquerda e 1 um ramo à direita. Cada caminho é codificado como um par (l, s), onde $l = |s|$ e l é auto-delimitada e s é a codificação do caminho. Se o i-ésimo caminho tem tamanho d_i, então esta descrição requere no máximo

$$d_i + 2\log d_i \tag{4.3}$$

Seja H a sequência que resulta da concatenação das descrições de todos os caminhos.

Facto É possível reconstruir h de H e de n.

Dem. Suponhamos que conhecemos H e que h é um `heap` com n valores diferentes. Podemos simular o passo de ordenação em sentido inverso. Inicialmente, $A[1..n]$ contém a sequência ordenada com pelo menos um elemento, $A[1]$.

para $i = 2, \ldots, n-1$ **faça:** Colocar o valor de $A[i]$ em $A[1]$ e deslocar todos os valores no caminho de ordem $n-i$ em H, uma posição para baixo a partir da raiz $A[1]$. O último valor neste caminho não tem para onde ir e é colocado na posição vazia $A[i]$.

terminação A sequência $A[1], \ldots, A[n]$ é o `heap` h.

□

Tem-se então que $K(h|n) \leq |H| + O(1)$ e pela equação (4.2) vem $|H| \geq n\log n - 6n$. Pela descrição da equação (4.3), isso só é possível se a média dos tamanhos dos caminhos é pelo menos $\log n - c'$, para alguma constante c'. Este valor dá o número médio de atribuições em cada volta do passo de ordenação do `heapsort`. Portanto, começando com um `heap` h, o `heapsort` executa pelo menos $n\log n - O(n)$ atribuições. Dado h, o número de comparações em `siftdown` é $2n\log n - O(n)$.

Dado que a maioria das permutações de n elementos é aleatória, segundo Kolmogorov, estes limites que se verificam para uma sequência aleatória p, também se verificam para todas as permutações em *média*. □

Bibliografia

[BC94] Daniel P. Bovet and Pierluigi Crescenzi. *Introduction to the Theory of Complexity*. Prentice Hall, 1994.

[BG89] Josep Díaz Balcázar, José Luis and Joaquim Gabarró. *Structural Complexity I*. Springer Verlag, 1989.

[BG90] Josep Díaz Balcázar, José Luis and Joaquim Gabarró. *Structural Complexity II*. Springer Verlag, 1990.

[Cha87] C. J. Chaitin. *Algorithmic Information Theory*. Cambridge University Press, 1987.

[Cha94] C. J. Chaitin. Limits of mathematics. Technical report, IBM Thomas J. Watson Research Center, 1994.

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In Association for Computing Machinery, editor, *Proc. 3rd Annual ACM Symposium on Theory of Computing*, 1971.

[Dew89] A. K. Dewdley. *The Turing Omnibus – 61 Excursions in Computer Science*. Computer Science Press, 1989.

[eAP95] Armando B. Matos e António Porto. Ackermann and the superpowers. Technical report, Faculdade de Ciências da Universidade do Porto, 1995.

[FB94] Robert Floyd and Richard Beigel. *The Language of Machines: An introduction to Computability and Formal Languages*. W.H. Freeman and Company, New York, 1994.

[GJ79] Michael Garey and David Johnson. *Computers and Intractability: a guide to the theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[Hof79] Douglas R. Hofstadler. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, 1979.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[J.S76] L. J.Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Tatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Knu68] Donald Knuth. *The Art of Computer Programming*, volume Vol. I: Fundamental Algorithms. Addison Wesley, 1968.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1981.

[LV90] Ming Li and Paul M. B. Vitànyi. Kolmogorov complexity and its applications. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 4, pages 188–254. Elsevier Science Publishers, 1990.

[LV94] Ming Li and Paul M. B. Vitàny. *An Introduction to Kolmogorov Complexity and its Application*. Texts and Monographs in Computer Science. Springer-Verlag, 1994?

[Man74] Zohar Manna. *Mathematical Theory of Computation*. Addison Wesley, 1974.

[Mat91] Armando B. Matos. Pesar sem pesos. Technical report, Centro de Informática da Universidade do Porto, 1991.

[Mat92] Armando B. Matos. A eficiência dos algoritmos. Technical report, Centro de Informática da Universidade do Porto, 1992.

[Mat95a] Armando B. Matos. Algoritmos probalísticos – algumas notas. Technical report, Faculdade de Ciências da Universidade do Porto, 1995.

[Mat95b] Armando B. Matos. Entropia, algoritmos e complexidade mínima. Technical report, Faculdade de Ciências da Universidade do Porto, 1995.

[Min74] M. Minsky. *Finite and Infinite Machines*. Addison Wesley, 1974.

[MS72] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proceedings of the 13th Annual Symp. on switching and Automata Theory*, pages 125–129. IEEE Computer Society, 1972.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[Phi92] I. C. C. Phillips. Recursion theory. In S. Abramsky D. M. Gabbay and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 79–187. Oxford University Press, 1992.

[Pra75] Vaughan Pratt. Every prime has a succinct certificate. *SIAM J. Computation*, 4(3):214–220, 1975.

[Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[RS93] Grzegorz Rozenberg and Arto Salomaa. *Cornerstones of Undecidability*. Prentice Hall International Series in Computer Science. Prentice Hall, 1993.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423,623–656, 1948. Part I,Part II.

[SvW88] R. Sommerhalder and S.C. van Westrhenen. *The Theory of Computability: Programs, Machines, Effectiveness and Feasibility*. Addison Wesley, 1988.

[Wei87] Klaus Weihrauch. *Computability*. Springer-Verlag, 1987.

[Wra76] C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3:23–33, 1976.

[Yas71] Ann Yasuhara. *Recursive Function Theory and Logic*. Computer Science and Applied Mathematics. Academic Press, 1971.