Computabilidade e Complexidade

Pedro Quaresma[1]

2011/2012

[1]Departamento de Matemática da Universidade de Coimbra.

Conteúdo

1 Introdução **3**

2 Funções Recursivas Parciais **5**
2.1 Funções Recursivas Primitivas 5
2.1.1 Exemplos e Propriedades de Fecho 6
2.2 Função de Ackermann 8
2.3 A função de Ackermann não é recursiva primitiva 9
2.4 Funções recursivas parciais 12

3 Máquinas e Modelos **13**
3.1 Máquinas de Turing 13
3.1.1 Máquinas de Turing deterministas 14
3.1.2 Toda a função recursiva parcial é T-computável 17
3.1.3 Toda a função T-computável é recursiva parcial. 21
3.1.4 Máquina de Turing Universal 24
3.2 Conjuntos Recursivos Enumeráveis 27
3.2.1 O Problema da Paragem 31
3.2.2 Os Teoremas do Ponto Fixo 35

4 Teoria da Complexidade **39**
4.1 Medindo a Complexidade 39
4.1.1 Notação $\mathcal{O}$-maiúsculo e o-minúsculo 40
4.1.2 Relações de Complexidade entre Modelos Diferentes 43
4.2 A Classe P 44
4.2.1 Exemplos de Problemas em P 45
4.3 A classe NP 46
4.3.1 Exemplos de Problemas em NP 47
4.3.2 P versus NP 48
4.4 Completude NP 49

Capítulo 1

Introdução

O advento da «Era dos Computadores» veio permitir automatizar métodos de resolução, ou seja, construir algoritmos capazes de solucionar problemas em tempo útil. Sabemos, no entanto, que há problemas para os quais não existem algoritmos que os resolvam (como o Problema da Paragem - Turing 1936, ou o 10º Problema de Hilbert - Matijasevic, 1970), enquanto outros há que, apesar de ser possível construir algoritmos para a sua resolução, esses algoritmos possuem «ordens de complexidade» tão grandes que originam a denominação de «ineficientes».

Um algoritmo é um método de resolução de um dado problema, no sentido de que, quando aplicado a uma particularização do problema, garante a obtenção de uma solução. Ao se pretender utilizar uma máquina para tratar problemas reais, procura-se, em geral, construir algoritmos que, não só garantam a obtenção da solução desejada, mas que resolvam de um modo «eficiente» o problema proposto. Mas o que ´ um algoritmo «eficiente»? O sentido geral da resposta a esta pergunta é, apesar de haver outros recursos envolvidos, o de relacionar eficiência com tempo (embora o espaço ocupado na resolução seja igualmente importante). Este tempo de resposta pode ser expresso em termos do tamanho do problema ou, melhor dizendo, da quantidade de informação que descreve a particularização do problema que se pretende ver resolvido. De facto, a descrição fornecida à máquina que vai resolver o problema é, no fundo, uma sequência finita de símbolos de um dado alfabeto (também finito). A complexidade, em termos temporais, de um algoritmo vai ser calculada em função do número de símbolos, comprimento, dessa sequência, fornecendo um limite superior para o tempo ocupado na resolução do problema pelo algoritmo, o que levanta a questão da classificação de problemas relativamente à sua dificuldade computacional intrínseca.

Não se pretende de modo algum apresentar aqui uma compilação exaustiva a de resultados da Teoria da Complexidade mas apenas cimentar algumas noções necessárias para uma compreensão mais profunda das questões que se levantam quando formulamos um problema e pretendemos resolvê-lo. De facto, não basta a identificar um problema e garantir que existe pelo menos uma solução para ele, é e ainda necessário saber se ele pode ser efectivamente resolvido em tempo e espaço de memória finitos.

As presentes notas seguem de muito perto [CL01] e [Sip97] e estão (fortemente) baseados nos apontamentos para esta disciplina (2007/08) do professor Reinhard Kahle.

Capítulo 2

Funções Recursivas Parciais

Antes de mais algumas questões introdutórias.

- Seja p um inteiro. O conjunto de todas as aplicações $\mathbb{N}^p$ em $\mathbb{N}$ designar-se-á por $\mathcal{F}$. Por convenção se $p = 0$ então o único elemento de $\mathbb{N}^p$ é a sequência vazia e, em consequência disso, os elementos de $\mathcal{F}_0$ podem ser identificados com os elementos de $\mathbb{N}$. O conjunto $\bigcup_{p \in \mathbb{N}} \mathcal{F}_p$ denotar-se-á por $\mathcal{F}$.

- Se i é um natural de 1 a p inclusive, a projecção de ordem i, P_p^i, é a função em $\mathcal{F}_p$ definida por
$$P_p^i(x_1, x_2, \ldots, x_p) = x_i$$

- Por definição a função sucessor S é a função em $\mathcal{F}_\infty$, $S(n) = n + 1$.

- Se $f_1, f_2, \ldots, f_n$ pertencem a $\mathcal{F}_p$ e g pertence a $\mathcal{F}_n$ então a composição das funções $h = g(f_1, f_2, \ldots, f_n)$ é um elemento de $\mathcal{F}_p$ definido por:
$$h(x_1, x_2, \ldots, x_p) = g(f_1(x_1, x_2, \ldots, x_p), f2(x_1, x_2, \ldots, x_p), \ldots, f_n(x_1, x_2, \ldots, x_p)).$$

2.1 Funções Recursivas Primitivas

Definição 2.1 (Definição por Recursão) *Se existem funções $g \in \mathcal{F}_p$ e $h \in \mathcal{F}_{p+2}$ então existe uma e uma só função $f \in \mathcal{F}_{p+1}$ que satisfaz as seguintes condições:*

Para todos os $x_1, x_2, \ldots, x_p$ e y em $\mathbb{N}$ tem-se,

$$f(x_1, x_2, \ldots, x_p, 0) = g(x_1, x_2, \ldots, x_p) \quad (2.1)$$
$$f(x_1, x_2, \ldots, x_p, y + 1) = h(x_1, x_2, \ldots, x_p, y, f(x_1, x_2, \ldots, x_p, y)). \quad (2.2)$$

Diz-se que f é definida por recursão a partir de g (o caso de base) e de h (o passo indutivo).

Definição 2.2 (Funções Recursivas Primitivas) *O conjuntos das funções recursivas primitivas é o menor sub-conjunto E de $\mathcal{F}$ que satisfaz as seguintes condições:*

1. *para todo o p, E contém todas as funções constantes de $\mathbb{N}^p$ em $\mathbb{N}$;*

2. *a função sucessor, $S(x) = x + 1$, pertence a E;*

3. *as projecções,* $P_n^i(x_1, \ldots, x_n) = x_i$, $1 \leqslant i \leqslant n$, *pertencem a* E;
4. *O conjunto* E *é fechado para a composição de funções. Isto é se* $g, h_1, h_2, \ldots h_n$ *são funções em* E *então a função composição* $f = g(h_1, h_2, \ldots, h_n)$ *pertence a* E;
5. *O conjunto* E *é fechado para a recursão. Isto é se* $p \in \mathbb{N}$ *e se* $g \in \mathcal{F}_p$ *e* $h \in \mathcal{F}_{p+2}$ *são elementos de* E *então a função* f *definida recursivamente por* g *e* h *pertence a* E.

Relembrando a noção de função característica χ_A do conjunto A, a qual é definida do seguinte modo:

$$\chi_A(n_1, n_2, \ldots, n_p) = \begin{cases} 1 & \text{se } (n_1, n_2, \ldots, n_p) \in A \\ 0 & \text{no caso contrário} \end{cases}$$

Definição 2.3 (Conjunto Recursivo Primitivo) *Um sub-conjunto* $A \subseteq \mathbb{N}^p$ *é designado recursivo primitivo se a sua função característica é uma função recursiva primitiva.*

2.1.1 Exemplos e Propriedades de Fecho

Exemplos 2.4

1. *Adição com 2: A função* $f(x) = x + 2$ *é recursiva primitiva. De facto, a definição precisa só a função de sucessor e composição:* $f(x) = S(S(x))$. *Isto é uma instância do esquema de composição com* $g(x)$ *e* $h_1(x)$ *igual* $S(x)$.
2. *Adição:* $f(y, x) = y + x$.

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

Isto é uma instância do esquema de recursão primitiva com

$$\begin{aligned} g(x) &= P_1^1(x) \\ h(z, y, x) &= P_1^3(S(z), y, x) \end{aligned}$$

Nota-se que h *é definida por composição (com alguma pedantearia mais exacto como* $h(z, y, x) = P_1^3(S(z), P_1^1(y), P_1^1(x))$*).*

O Esquema da Definição por Casos Sejam f e g duas funções recursivas primitivas em $\mathcal{F}_p$ e seja A um sub-conjunto recursivo primitivo de $\mathbb{N}^p$, então a função h definida por:

$$h(x_1, x_2, \ldots, x_n) = \begin{cases} f(x_1, x_2, \ldots, x_n) & \text{se } (x_1, x_2, \ldots, x_n) \in A \\ g(x_1, x_2, \ldots, x_n) & \text{no caso contrário} \end{cases}$$

é recursiva primitiva.

É suficiente observar que:

$$h = f \cdot \chi(A) + g \cdot \chi(\mathbb{N}^p - A).$$

O mesmo pode-se generalizar facilmente para comportar a definição de uma função por múltiplos casos.

Somas e Produtos Limitados Se $f \in \mathcal{F}_{p+1}$ é uma função recursiva primitiva, então as funções:

$$g(x_1, x_2, \ldots, x_p, y) = \sum_{t=0}^{t=y} f(x_1, x_2, \ldots, x_p, t)$$

e

$$h(x_1, x_2, \ldots, x_p, y) = \prod_{t=0}^{t=y} f(x_1, x_2, \ldots, x_p, t)$$

são também funções primitivas recursivas.

As funções g e h são facilmente definidas por recursão. Por exemplo, para o caso da função g ter-se-ia:

$$\begin{aligned} g(x_1, x_2, \ldots, x_p, 0) &= f(x_1, x_2, \ldots, x_p, 0) \\ g(x_1, x_2, \ldots, x_p, y+1) &= g(x_1, x_2, \ldots, x_p, y) + f(x_1, x_2, \ldots, x_p, y+1) \end{aligned}$$

A função h definir-se-ia de igual forma.

O operador μ limitado Seja A um sub-conjunto recursivo primitivo de $\mathbb{N}^{p+1}$. Então a função $f \in \mathcal{F}_{p+1}$ definida como se segue, é uma função recursiva primitiva:

$$\begin{aligned} f(x_1, x_2, \ldots, x_p, z) &= 0 && \text{se não existe um inteiro } t \leqslant z \text{ tal que } (x_1, x_2, \ldots, x_p, t) \in A; \\ f(x_1, x_2, \ldots, x_p, z) &= t && \text{com } t \text{ o menor inteiro } t \leqslant z \text{ tal que } (x_1, x_2, \ldots, x_p, t) \in A, \\ & && \text{no caso contrário.} \end{aligned}$$

A definição de f usa a recursão primitiva, o esquema da definição por casos e as somas limitadas.

$$\begin{aligned} & f(x_1, x_2, \ldots, x_p, 0) = 0 \\ & f(x_1, x_2, \ldots, x_p, z+1) = \\ & \quad = \begin{cases} f(x_1, x_2, \ldots, x_p, z) & \text{se } \sum_{y=0}^{y=z} \chi_A(x_1, x_2, \ldots, x_p, y) \geqslant 1; \\ z+1 & \text{no caso contrário e se } (x_1, x_2, \ldots, x_p, z+1) \in A; \\ 0 & \text{em todos os outros casos} \end{cases} \end{aligned}$$

Para designar esta função usar-se-á a seguinte notação:

$$f(x_1, x_2, \ldots, x_p, z) = \mu t \leqslant z[(x_1, x_2, \ldots, x_p, t) \in A].$$

A qual se pode ler da seguinte forma $f(x_1, x_2, \ldots, x_p, z)$ é o menor inteiro t tal que $t \leqslant z$ tal que $(x_1, x_2, \ldots, x_p, t) \in A$ para o caso em que um tal t existe. No caso contrário $f(x_1, x_2, \ldots, x_p, z) = 0$.

A função $\pi(n)$ é Recursiva Primitiva A função π cujo valor em n é dado pelo $(n+1)$-ésimo primo é recursiva primitiva.

A função π pode ser definida por recursão usando o operador μ limitado da seguinte forma:

$$\begin{array}{rcl} \pi(0) & = & 2 \\ \pi(n+1) & = & \mu z \leqslant (\pi(n)!+1)[z > \pi(n) \text{ e } z \text{ é primo}] \end{array}$$

Utiliza-se aqui o resultado que nos garante que, para um dado p primo, existe pelo menos um outro primo entre p e $p!+2$.

2.2 Função de Ackermann

A função de Ackermann constitui um caso exemplar de uma é função efectivamente computável, no sentido intuitivo da palavra, mas que não é primitiva recursiva.

A definição tal como é aqui feita é uma variante, um pouco diferente daquela que usualmente é referida, mas que torna a demonstração do resultado principal desta secção, o facto de que esta função não é primitiva recursiva, mais fácil.

Definição 2.5 *A* função de Ackermann *(numa forma conveniente para os argumentos seguintes*[1]*) é definida por:*

$$\begin{array}{rrcl} \forall_{x \in \mathbb{N}} & ack(0,x) & = & 2^x \\ \forall_{y \in \mathbb{N}} & ack(y,0) & = & 1 \\ \forall_{x,y \in \mathbb{N}} & ack(y+1,x+1) & = & ack(y, ack(y+1,x)) \end{array}$$

Nota-se que a última cláusula não é uma instância do esquema da recursão primitiva.

Lema 2.6 *Para todo o $n, m \in \mathbb{N}$, $ack(n,m) \in \mathbb{N}$.*

Dem.: Indução sobre n:

$$\forall_{m \in \mathbb{N}} . ack(n,m) \in \mathbb{N}$$

a) caso base: $\forall_{m \in \mathbb{N}} . ack(0,m) = 2^m \in \mathbb{N}$.

b) passo indutivo:

hipI: $\forall_{m \in \mathbb{N}} . ack(n,m) \in \mathbb{N}$.

Tese: $\forall_{m \in \mathbb{N}}, ack(n+1,m) \in \mathbb{N}$

b1) caso base: $ack(n+1,0) = 1 \in \mathbb{N}$.

[1] Na literatura existem variantes diferentes da função de Ackermann. A versão usada aqui é uma variante duma simplificação da função original por Rósza Péter.

b2) passo indutivo:
hipII: $ack(n+1,m) \in \mathbb{N}$

$$ack(n+1,m+1) = \underbrace{ack(n, \underbrace{ack(n+1,m)}_{\in\mathbb{N} \text{ por hipII}})}_{\in\mathbb{N} \text{ por hipI}}$$

□

2.3 A função de Ackermann não é recursiva primitiva

Lema 2.7 *Definam-se,*

$$ack_n(x) = ack(n,x)$$
$$ack_n^k(x) = \underbrace{ack_n(ack_n(\dots ack_n(x)\dots))}_{k \text{ vezes}}$$

A definição das funções $ack_n(x)$ mostram-nos que efectivamente existe uma única função ack verificando as condições acima, e que por outro lado podemos ver que $ack_0(x) = 2^x$ e que para todo o $n > 0$ a função ack_n é definida a partir de ack_{n-1} por recursão:

$$ack_n(0) = 1 \quad e \quad ack_n(x+1) = ack_{n-1}(ack_n(x))$$

Pode-se provar, por indução em n, que as funções $ack_n(x)$ são primitivas recursivas sem que isso permita afirmar no entanto que a função de Ackermann o seja.

Então, para todos os n, k e h,

$$ack_n^k \circ ack_n^h = ack_n^{k+h}.$$

Lema 2.8 (5.6²) *Para todo o n e todo o x,*

$$ack_n(x) > x.$$

A demonstração é similar à demonstração de que lema 2.6, «Para todo o $n, m \in \mathbb{N}$, $ack(n,m) \in \mathbb{N}$». Podemos substituir na demonstração $\cdot \in \mathbb{N}$ por $\cdot > x$.

Exercício 1 *Demonstrar os lemas que se seguem.*

Lema 2.9 (5.7) *Para todo o n e para todo o x,*

$$ack_n(x+1) > ack_n(x).$$

Lema 2.10 (5.8) *Para todo o $n \geqslant 1$ e para todo o x,*

$$ack_n(x) \geqslant ack_{n-1}(x).$$

Lema 2.11 (5.12) *Para todo o n, k e para todo o x,*

$$ack_n^k(x) \leqslant ack_{n+1}(x+k).$$

Definição 2.12 *Sejam f e g duas funções. Diz-se que f domina g se e só se existe um natural A tal que para todos os $x_1, x_2, \ldots, x_n$ se tem que $g(x_1, x_2, \ldots, x_n) \leqslant f(\max(x_1, x_2, \ldots, x_n, A))$.*

Definição 2.13 $C_n = \{g \mid$ *existe um k tal que ack_n^k domina g*$\}$

Exemplos 2.14

- $Z(x) \in C_0$
- $S(x) \in C_0$
- $P_i^n(\vec{x}) \in C_0$
- *A função constante* $K_k(x) = k \in C_0$
- $\max(x_1, \ldots, x_n) \in C_0$
- $x + y \in C_0$
- $k \cdot x \in C_0$, *com* $k \in \mathbb{N}$
- $ack_n \in C_n$
- *Se* $g \in C_n$ *e para todos os* $x_1, \ldots, x_n$ $f(x_1, \ldots, x_n) \leqslant g(x_1, \ldots, x_n)$ *então* $f \in C_n$

Lema 2.15 *C_n é fechado para a composição.*

Dem.: Quer provar-se que dadas as funções $g, h_1, \ldots, h_m$ pertencentes a C_n, $g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$ é ainda uma função em C_n.

Tem-se que $g, h_1, \ldots, h_m \in C_n$, logo, existem $A, A_1, \ldots, A_m, k, k_1, \ldots, k_m$ tais que

$$g(\vec{y}) \leqslant ack_n^k(\max(\vec{y}, A))$$
$$h_i(\vec{x}) \leqslant ack_n^{k_i}(\max(\vec{x}, A_i))$$

Defina-se $\widetilde{A} = \max(A, A_1, \ldots, A_m)$ e defina-se $\widetilde{k} = \max(k_1, \ldots, k_m)$. Ora,

$$\begin{aligned} g(h_1(\vec{x}), \ldots, h_m(\vec{x})) &\leqslant ack_n^k(\max(h_1(\vec{x}), \ldots, h_m(\vec{x}), A)) \\ &\leqslant ack_n^k(\max(ack_n^{k_1}(\max(\vec{x}, A_1)), \ldots, ack_n^{k_m}(\max(\vec{x}, A_m)), A)) \\ &\leqslant ack_n^k(ack_n^{\widetilde{k}}(\max(\vec{x}, \widetilde{A}))) \\ &\leqslant ack_n^{k+\widetilde{k}}(\max(\vec{x}, \widetilde{A})) \end{aligned}$$

□

Lema 2.16 *Se $g, h \in C_n$, então f com $f(0, \vec{x}) = g(\vec{x})$ e $f(y + 1, \vec{x}) = h(f(y, \vec{x}), y, \vec{x})$ pertence ao conjunto C_{n+1}.*

Dem.: Existem A_1, A_2, k_1, k_2 tais que

$$\begin{aligned} g(\vec{x}) &\leqslant ack_n^{k_1}(\max(\vec{x}, A_1)) \\ h(y, z, \vec{x}) &\leqslant ack_n^{k_2}(\max(y, z, \vec{x}, A_2)) \end{aligned}$$

Vamos demonstrar que

$$f(y, \vec{x}) \leqslant ack_n^{k_1 + y \cdot k_2}(\max(\vec{x}, y, A_1, A_2))$$

Demonstração por indução em y:
Caso base: $y = 0$

$$\begin{aligned} f(0, \vec{x}) &= g(\vec{x}) \\ &\leqslant ack_n^{k_1}(\max(\vec{x}, A_1)) \\ &\leqslant ack_n^{k_1 + 0 \cdot k_2}(\max(\vec{x}, 0, A_1, A_2)) \end{aligned}$$

Hipótese de Indução:

$$f(y, \vec{x}) \leqslant ack_n^{k_1 + y \cdot k_2}(\max(\vec{x}, y, A_1, A_2))$$

$$\begin{aligned} f(y + 1, \vec{x}) &= h(f(x, \vec{x}), x, \vec{x}) \\ &\leqslant ack_n^{k_2}(\max(\vec{x}, f(y, \vec{x}, y, A_2)) \\ &\leqslant ack_n^{k_2}(ack_n^{k_1 + y \cdot k_2}(\max(\vec{x}, y, A_1, A_2))) \\ &\leqslant ack_n^{k_1 + y \cdot k_2 + k_2}(\max(\vec{x}, y, A_1, A_2)) \end{aligned}$$

Mas, pelo Lema 2.11, $ack_n^k(x) \leqslant ack_{n+1}(x + k)$, ou seja,

$$\begin{aligned} f(y, \vec{x}) &\leqslant ack_n^{k_1 + yk_2}(\max(\vec{x}, y, A_1, A_2)) \\ &\leqslant ack_{n+1}(\underbrace{\underbrace{\max(\vec{x}, y, A_1, A_2)}_{\in C_0} + \underbrace{k_1 + yk_2}_{\in C_0}}_{\in C_0}) \end{aligned}$$

Conclui-se assim que $f(y, \vec{x}) \in C_{n+1}$. □

Corolário 2.17 (5.14) *Todas as funções recursivas primitivas pertencem ao conjunto* $C = \bigcup_{n \in \mathbb{N}} C_n$.

Teorema 2.18 (5.15) *A função ack não é uma função recursiva primitiva.*

Dem.: (por absurdo)
Suponhamos que ack é RPrim.
Então, pelo Lema 2.15 $f(x) = ack(x, 2x)$ é RPrim.
Logo, pelo Corolário 2.17 existem n, k e A tais que para todos os $x > A$, $f(x) \leqslant ack_n^k(x)$.

Ou ainda,

$$\begin{aligned} ack(x,2x) &\leqslant ack_n^x(x) \\ &\leqslant ack_{n+1}(x+k) \end{aligned}$$

Para $x > \max(A, k, n+1)$ vem que

$$ack_{n+1}(x+k) < ack_{n+1}(2x) < ack_x(2x) = ack(x,2x).$$

Contradição! □

2.4 Funções recursivas parciais

Definição 2.19 *Uma função f de números naturais diz-se* recursiva parcial *se puder ser gerada num número finito de passos através das regras:*

1. $Z(x) = 0$, *a função* zero *é recursiva parcial (*RPar*).*
2. $S(x) = x+1$, *a função* sucessor *é* RPar*.*
3. $P_n^i(x_1, x_2, \ldots, x_n) = x_i$, *a função* projecção *é* RPar*.*
4. Composição. *Se $g, h_1, h_2, \ldots, h_n$ são funções* RPar*, então a função $f(\vec{x}) = g(h_1(\vec{x}), h_2(\vec{x}), \ldots, h_n(\vec{x}))$ é* RPar*.*
5. Recursão Primitiva. *Se g e h são funções* RPar*, então a função f definida recursivamente por*

$$\begin{aligned} f(0,\vec{x}) &= g(x), \\ f(y+1,\vec{x}) &= h(f(y,\vec{x}), y, \vec{x}) \end{aligned}$$

   é RPar*.*

6. Recursão μ. *Se g é* RPar*, então a função f definida por $f(\vec{x}) = \mu\, y[g(\vec{x}, y) = 0]$ é RPar, onde a expressão $\mu\, y[g(\vec{x}, y) = 0]$ é definida por: y é o menor valor para o qual $g(\vec{x}, y) = 0$, no caso em que tal y existe e para todo o $z < y$ $g(\vec{x}, z)$ está definido. Caso contrário, $\mu\, y[g(\vec{x}, y) = 0]$ não tem um valor bem definido.*

Capítulo 3

Máquinas e Modelos

A característica básica de uma função computável é a disponibilidade de um algoritmo, uma sequência finita de instruções bem definidas e não ambíguas, cada uma das quais pode ser executada mecanicamente num período de tempo finito e com uma quantidade de esforço finita, que descreve como computá-la.

De acordo com a tese de Church-Turing, funções computáveis são as que podem ser calculadas usando um dispositivo mecânico dado uma quantidade ilimitada de espaço de armazenamento e de tempo de execução. A mesma tese define que qualquer função que possui um algoritmo é computável.

A interpretação do significado de algoritmo e como ele é usado cabe ao modelo de computação, mas cada interpretação compartilha algumas propriedades.

No que se segue vamos estudar um dos possíveis modelos de computabilidade, as *Máquinas de Turing*[1].

3.1 Máquinas de Turing

1936 Alan Turing define uma máquina formal a partir de princípios simples (ler , apagar e escrever símbolos numa fita) e define o conceito de Máquina Universal.

1936 Alonzo Church define o λ-Cálculo e mostra que define qualquer função para as qual exista uma Máquina de Turing.

Uma Máquina de Turing (MT) define um modelo matemático de computação, simples e universal.

- Semelhante a um autómato com memória ilimitada é um modelo mais próximo de um computador generalista.
- Principal modelo no estudo do que é ou não computável.
- Até hoje todo o procedimento pode ser implementado por (algum tipo de) Máquina de Turing.

[1] Alan Turing (23 de Junho de 1912 – 7 de Junho de 1954)

3.1.1 Máquinas de Turing deterministas

Definição 3.1 *Uma* máquina de Turing *é composta por:*

- *«Parâmetros genéricos»*[2]
  - *Uma fita com um número finito de bandas paralelas limitadas à esquerda e ilimitadas à direita. Cada banda é dividida em caixas.*
  - *Uma cabeça de leitura que pode ler, escrever e apagar símbolos nas bandas. A cabeça pode ser deslocada para o lado esquerdo até ao limite da fita ou para o lado direito.*
  - *Define-se o alfabeto* $S = \{d, 1, b\}$ *com os três símbolos:*
    - d *(o fim da fita no lado esquerdo),*
    - 1 *(ou de forma equivalente, ' | '),*
    - b *(o símbolo vazio).*

  Exemplo 3.2 *O conteúdo duma banda pode ser:* $d11b111bb1b111bbb\cdots$

- *«Parâmetros específicos», i.e., para uma máquina de Turing concreta têm-se as seguintes variáveis:*
  - *o número* n *das bandas*
  - *um conjunto finito de estados* E *com pelo menos dois elementos* e_i *(estado inicial) e* e_f *(estado final)*
  - *uma tabela* M *(tabela ou função de transição) que representa uma função* $S^n \times E \to S^n \times E \times \{-1, 0, 1\}$. *De notar que* M *é finita já que está definida entre conjuntos finitos.*

A definição de uma MT comporta a possibilidade de diferentes variantes para os parâmetros genéricos:

- *várias fitas;*
- *fitas extensíveis para ambos os lados;*
- *fitas com finalidade específica (fitas para entrada e fitas para saída);*
- *a máquina pode ter mais de uma cabeça de leitura por fita;*
- *em vez de fita, a máquina pode usar como memória um reticulado bi-dimensional;*

[2] Estes parâmetros são fixos para todas as máquinas de Turing que *nós* consideramos aqui. Existem variantes de definição da máquina de Turing que variam nestes parâmetros.

Funcionamento duma MT *O funcionamento duma máquina pode ser descrito por:*

- *No momento $t = 0$ a cabeça da máquina está no extremo esquerdo da fita com o símbolo d em todas as bandas e a máquina está no estado inicial e_i.*
- *Em cada momento t a máquina lê os símbolos $S_1, \ldots, S_n$. No caso em que a máquina está no estado e calcula-se*

$$M(S_1, \ldots, S_n, e) = (S'_1, \ldots, S'_n, e', \epsilon) \quad \text{com } \epsilon \in \{-1, 0, 1\}$$

  - *Neste caso, a máquina escreve nas posições de $(S_1, \ldots, S_n)$ os respectivos $(S'_1, \ldots, S'_n)$.*
  - *A cabeça da máquina desloca-se um passo para a esquerda se $\epsilon = -1$, desloca-se um passo para a direita se $\epsilon = 1$ e permanece na mesma posição se $\epsilon = 0$.*
  - *O estado da máquina muda para e' e a computação passa para o instante $t + 1$.*
- *No caso em que a máquina está no estado final, e_f, a computação termina.*

Para este funcionamento temos as seguintes restrições formais:

- *Para todos os $(S_1, \ldots, S_n) \in S^n$, $M(S_1, \ldots, S_n, e_f) = (S_1, \ldots, S_n, e_f, 0)$.*

  Isto garante que a máquina «pára» no estado final e_f.
- *Para todos os $e \in E$, $M(d, \ldots, d, e) = (d, \ldots, d, e', \epsilon)$ com $\epsilon \neq -1$.*

  Isto garante que a cabeça da máquina não desloca-se para o lado esquerdo no fim da banda.
- *Para todos os $S_1, \ldots, S_n, S'_1, \ldots, S'_n$ se $(S_1, \ldots, S_n) \neq (d, \ldots, d)$ e $M(S_1, \ldots, S_n, e) = (S'_1, \ldots, S'_n, e', \epsilon)$ então pelo menos um $S'_i \neq d$, isto é, $(S'_1, \ldots, S'_n) \neq (d, \ldots, d)$.*

  Isto garante que o símbolo d é só usado para marcar o fim da banda.

Definição 3.3 *Uma banda representa um número natural n (no momento t) se e só se (no momento t) o conteúdo da banda é $(d, \underbrace{1, 1, \ldots, 1}_{n \text{ vezes}}, b, b, \ldots)$*

Nota que o *zero* é representado por $(d, b, b, \ldots, b, b, \ldots)$.

Nota também que $(d, 1, b, 1, b, b, \ldots)$ é um conteúdo «legal» duma banda, mas não representa um número.

Definição 3.4 *Seja $f(x_1, \ldots, x_p)$ uma função parcial e $\mathcal{M}$ uma máquina de Turing com pelo menos $p + 1$ bandas. Diz-se que $\mathcal{M}$ calcula f se e só se:*

- *As bandas $1, \ldots, p$ representam $x_1, \ldots, x_p$ no estado inicial e_i e todas as outras bandas estão vazias (b).*
  - *No caso em que $f(x_1, \ldots, x_p)$ não está definido, a máquina $\mathcal{M}$ não pára.*
  - *No caso em que $f(x_1, \ldots, x_p) = y$, a máquina pára após um número finito de passos e a banda $p + 1$ representa y (as bandas $1, \ldots, p$ representam $x_1, \ldots, x_p$ e as bandas $p + 2, \ldots$ estão vazias).*

Definição 3.5 *Uma função f diz-se T-*computável *se e só se existe uma máquina de* Turing *$\mathcal{M}$ que calcula f.*

Exemplo 3.6 $f(x) = x + 1$.

Considere-se a máquina $\mathcal{M}$ definida da seguinte forma:

- *A máquina $\mathcal{M}$ tem duas bandas.*
- $E = \{e_i, e_f\}$
- *A função de transição M é dado da seguinte forma:*

$$\begin{aligned} M(d, d, e_i) &= (d, d, e_i, +1) \\ M(1, b, e_i) &= (1, 1, e_i, +1) \\ M(b, b, e_i) &= (b, 1, e_f, 0) \end{aligned}$$

(Não é necessário especificar os outros casos.)

$\mathcal{M}$ é a MT que calcula f.

Exemplo 3.7 $g(x) = 2x$

Considere-se a máquina $\mathcal{M}$ definida da seguinte forma:

- *A máquina $\mathcal{M}$ tem duas bandas*
- $E = \{e_i, e_1, e_2, e_f\}$
- *A função de transição M é dado da seguinte forma:*

$$\begin{array}{rcll} M(d, d, e_i) &=& (d, d, e_i, +1) & \}\text{ Estado Inicial} \\ M(1, b, e_i) &=& (1, 1, e_1, +1) & \rbrace\text{ Adiciona um à segunda banda (caso} \\ M(b, b, e_i) &=& (b, b, e_f, 0) & \rbrace\ x \neq 0) \\ M(1, b, e_1) &=& (1, b, e_1, +1) & \rbrace\text{ Mover até ao fim da palavra} \\ M(b, 1, e_1) &=& (b, 1, e_1, +1) & \\ M(b, b, e_1) &=& (b, 1, e_2, -1) & \}\text{ Adiciona um à segunda banda} \\ M(b, 1, e_2) &=& (b, 1, e_2, -1) & \rbrace\text{ Regressa ao ponto que acabou de} \\ M(1, b, e_2) &=& (1, b, e_2, -1) & \rbrace\text{ ser duplicado} \\ M(1, 1, e_2) &=& (1, 1, e_i, +1) & \rbrace\text{ Repete até que todas as barras te-} \\ M(b, 1, e_i) &=& (b, 1, e_f, 0) & \rbrace\text{ nham sido duplicadas} \end{array}$$

$\mathcal{M}$ é a MT que calcula g.

d	1	1	1					$\cdots$
d	1	1	1	1	1	1		$\cdots$

3.1.2 Toda a função recursiva parcial é T-computável

Teorema 3.8 *Toda a função recursiva parcial é T-computável.*

Dem.: (por indução estrutural sobre a definição de funções recursivas parciais)

3 casos básicos.

- Zero: $M(d, d, e_i) = (d, d, e_f, 0)$.
- Sucessor: $f(x) = x + 1$ (ver exemplo 3.6).
- Projecção: $P_p^k(x_1, x_2, \ldots, x_k, \ldots, x_p) = x_k$.

$$M(\vec{d}, e_i) = (\vec{d}, e_i, +1)$$
$$M(\vec{s}, b, e_i) = \begin{cases} (\vec{s}, 1, e_i, +1), & \text{se } s_k = 1, \\ (\vec{s}, b, e_f, 0), & \text{se } s_k = b. \end{cases}$$

3 passos indutivos.

- Composição:

$$f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_n(\vec{x})),$$

com $\vec{x} = x_1, \ldots, x_p$, $f \in \mathcal{F}_p^*$, $g \in \mathcal{F}_n^*$ e $h_i \in \mathcal{F}_p^*$.
Hipótese de Indução:

> Existem $n + 1$ máquinas de *Turing*, $\mathcal{N}, \mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ que calculam, respectivamente, $g, h_1, h_2, \ldots h_n$.

Pretende-se encontrar uma máquina $\mathcal{M}$ que calcula f. Ora,

- $\mathcal{N}$ tem n' bandas ($n' \geqslant n + 1$) e um conjunto E_0 de estados.
- Cada $\mathcal{M}_i$ tem p_i bandas ($p_i \geqslant p + 1$) e um conjunto de estados E_i

Defina-se $\mathcal{M}$ do seguinte modo:

- $\mathcal{M}$ tem $p' = p + \sum_{i=1}^{n}(p_i - p) + n' - n$ bandas;
- O conjunto de estados de $\mathcal{M}$ é $E = E_0 \cup \displaystyle\bigcup_{i=1}^{n} E_i$, sendo que $\forall_{i,j}\ i \neq j \Rightarrow E_j \cap E_i = \varnothing$;
- O estado inicial de $\mathcal{M}$ é o estado inicial de $\mathcal{M}_1$;
- Do estado final da $\mathcal{M}_i$, $i < n$, mudamos a máquina no estado inicial de $\mathcal{M}_{i+1}$.
- Do estado final da $\mathcal{M}_n$, mudamos a máquina no estado inicial de $\mathcal{N}$.
- O estado final de $\mathcal{M}$ é o estado final de $\mathcal{N}$.

O funcionamento da máquina deve ser óbvio: Usamos $\mathcal{M}_1$ para $\mathcal{M}_n$ com as bandas re-numeradas; depois usamos $\mathcal{N}$ para calcular o resultado final.

- Recursão Primitiva.

$$f(\vec{x}, 0) = g(\vec{x})$$
$$f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}, y))$$

Com $f \in \mathcal{F}^*_{n+1}$, $g \in \mathcal{F}^*_n$ e $h \in \mathcal{F}^*_{n+2}$.

Como hipótese da indução, existem máquinas $\mathcal{M}_g$ e $\mathcal{M}_h$ que calculam g e h respectivamente.

Podemos supor que $\mathcal{M}_g$ tem $n+1+k$ bandas e um conjunto de estados E_g e que $\mathcal{M}_h$ tem $n+3+k'$ bandas e um conjunto de estados E_h com $E_g \cap E_h = \varnothing$.

Vamos «construir» uma máquina $\mathcal{N}$ com $n+4+k+k'$ bandas e conjunto de estados $E_g \cup E_h \cup \{e_0, \ldots, e_7\}$ com $E_g \cap E_h = \varnothing$ e $(E_g \cup E_h) \cap \{e_0, \ldots, e_7\} = \varnothing$.

O estado inicial de $\mathcal{N}$ é o estado inicial de $\mathcal{M}_g$. No instante inicial ($t = 0$) as bandas 1 a p e $p+1$ representam x_1 a x_p e y respectivamente. No instante final, na banda $n+3$ ter-se-á o resultado pretendido $f(\vec{x}, y)$.

De facto, vamos só descrever informalmente o funcionamento de $\mathcal{N}$:

1. Calcular $f(\vec{x}, 0)$ usando $\mathcal{M}_g$, com as bandas 1 a $n+1$ e k mas colocando o resultado na banda $n+4$;
2. Mudar a cabeça para o início da fita e mudar o estado para e_0;
3. No estado e_0 comparar o valor da banda $n+1$ (o y original) com o valor da banda $n+2$ (um contador que guarda os sucessivos valores de y para cada passagem do ciclo, o seu valor inicial é 0).
   (a) Se os valores comparados são iguais, copiar o valor da banda $n+4$ para a banda $n+3$ e a máquina termina;
   (b) Caso contrário, copiar o valor da banda $n+4$ para a banda $n+3$, apagando de seguida a banda $n+4$. Calcular $f(\vec{x}, y+1)$ através da máquina $\mathcal{M}_h$ colocando o resultado na banda $n+4$. No fim incrementa-se o valor da banda $n+2$.

   Voltar ao passo 2.

Nota 1 *Quando calculamos $f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}, y))$ temos:*

- *o valor de $\vec{x}$ nas bandas $1, \ldots, n$;*
- *o valor de y na banda $n+1$;*
- *o contador incremental (de 0 a y) na banda $n+2$;*
- *o valor de $f(\vec{x}, y)$ na banda $n+3$.*
- *o valor de $g(\vec{x}, 0)$ e de $h(\vec{x}, y, f(\vec{x}, y))$ na banda $n+4$.*

- Recursão μ.

$$f(\vec{x}) = \mu\, y[g(\vec{x}, y) = 0]$$

Por hipótese da indução existe uma máquina $\mathcal{M}_g$, com $n+2+k$ bandas e o conjunto de estados E, que calcula g.

Vamos «construir» uma máquina $\mathcal{N}$ com também $n+2+k$ bandas e o conjunto de estado $E \cup \{e_0, e_1, e_2, e_3\}$ que calcula f.

O funcionamento de $\mathcal{N}$ é o seguinte:

(0. Inicializar $y = 0$ — é feito automaticamente, porque a banda $n+1$ está vazia no inicio da computação.)

1. Calcular $g(\vec{x}, y)$.
2. Mudar a cabeça para o início da banda e mudar o estado para e_0.
3. Testar se a banda $n+2$ representa 0:
   (a) Se sim, mudar para o estado final.
   (b) Senão, incrementar o valor da banda $n+1$ e voltar ao passo 1.

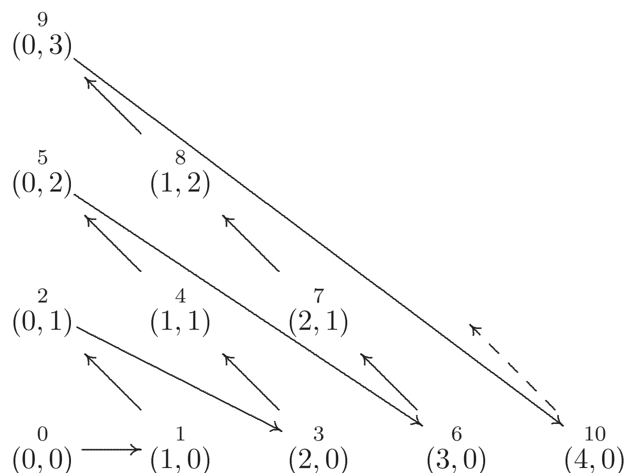
□

Antes de se poder avançar para a demonstração do resultado inverso deste que acabámos de demonstrar é necessário estudar alguns resultados referente à codificação de sequências de inteiros.

Codificações

Lema 3.9 *Para todo o $n > 0$ existem funções recursivas primitivas $\alpha^n, \beta_1^n, \beta_2^n, \ldots, \beta_n^n$ tal que: α^n é uma bijecção de $\mathbb{N}^n$ para $\mathbb{N}$ com funções inversas $\beta_1^n, \beta_2^n, \ldots, \beta_n^n$.*

Dem.: Começa-se por ilustrar o caso α_2 que permite compreender a forma como a demonstração é feita.



Isto é, vai-se enumerar os pares (x, y) seguindo as diagonais em sentido ascendente para os valores de $x+y$ constantes. Começa-se na diagonal cujo valor de $x+y$ é zero, daí passa-se para a diagonal $x + y = 1$, procedendo sempre de igual forma. O valor de $\alpha_2(x, y)$ é exactamente o número de predecessores de (x, y) na enumeração.

Sendo assim tem-se que para o par $(p+n,0)$, existem:

$$1+2+\cdots+(n+p)=\frac{1}{2}(n+p+1)(n+p)$$

elementos. O par (p,n) está na mesma diagonal que o par $(p+n,0)$ e está exactamente n posições após ele. Consequentemente tem-se:

$$\alpha_2(p,n)=\frac{1}{2}(n+p+1)(n+p)+n$$

A função α_2 é claramente recursiva primitiva e o seu valor é sempre igual ou maior do que n ou p. Dado que, por definição, α_2 é bijectiva. pode-se recuperar n e p de $\alpha_2(n,p)$, definindo as seguintes funções:

$$\begin{aligned}\beta_2^1(x) &= \mu z \leqslant x[\exists_{t\leqslant x}\ \alpha_2(z,t)=x]\\ \beta_2^2(x) &= \mu z \leqslant x[\exists_{t\leqslant x}\ \alpha_2(t,z)=x]\end{aligned}$$

Por definição β_2^1 e β_2^2 são também recursivas primitivas.

A partir daqui podemos definir $\alpha_3(x,y,z)=\alpha_2(x,\alpha_2(y,z))$ e $\beta_3^1=\beta_2^1$, $\beta_3^2=\beta_2^1\circ\beta_2^2$, $\beta_3^3=\beta_2^2\circ\beta_2^2$.

Mais genericamente temos:

$$\alpha_{p+1}(x_1,x_2,\ldots,x_p,x_{p+1})=\alpha_p(x_1,x_2,\ldots,\alpha_2(x_p,x_{p+1}));$$

$$\beta_{p+1}^1=\beta_p^1,\ \beta_{p+1}^2=\beta_p^2,\ \ldots,\beta_{p+1}^{p-1}=\beta_p^{p-1},\ \beta_{p+1}^p=\beta_2^1\circ\beta_p^p,\ \beta_{p+1}^{p+1}=\beta_2^2\circ\beta_p^p.$$

Conclui-se definindo $\alpha_1(x)=x$ e $\beta_1^1(x)=x$. □

Vai-se usar a notação S para designar o conjunto das sequências finitas de naturais.

Vejamos agora uma codificação possível para sequências de inteiros.

Definição 3.10 *A função Ω é a função de $\mathcal{S}$ em $\mathbb{N}$ definida da seguinte forma:*

$$\begin{aligned}\Omega((x_0,x_1,\ldots,x_p)) &= \pi(0)^{x_0}\cdot\pi(1)^{x_1}\cdots\pi(p)^{x_p}\\ \Omega(s) &= 1,\ \textit{com } s \textit{ a sequência vazia}\end{aligned}$$

Em sentido inverso temos a função δ.

Definição 3.11 *A função δ é a função de $\mathcal{F}_2$ definida por:*

$$\delta(i,x)=\mu z\leqslant x[x\ \textit{não é divisível por}\ \pi(i)^{z+1}]$$

$\delta(i,x)$ é o expoente de $\pi(i)$ na decomposição de x num produto de primos.

3.1.3 Toda a função T-computável é recursiva parcial.

Relembrando resultados anteriores temos que:

- Seja $f(x_1, \ldots, x_p, y)$ uma função recursiva primitiva, então a soma limitada é uma função recursiva primitiva (secção 2.1.1):

$$g(x_1, \ldots, x_p, y) = \sum_{t=0}^{t=y} f(x_1, \ldots, x_p, t).$$

- Para todo o $n > 0$ existem funções recursivas primitivas $\alpha_n, \beta_n^1, \beta_n^2, \ldots, \beta_n^n$ tal que: α_n é uma bijecção de $\mathbb{N}^n$ para $\mathbb{N}$ com funções inversas $\beta_n^1, \beta_n^2, \ldots, \beta_n^n$ (lema 3.9).

  Quando n é claro do contexto deixamos este índice e usamos só $\alpha, \beta^1, \beta^2, \ldots$.

Questões de notação: os nomes dos estados não são relevantes e como tal podemos considerar a seguinte codificação para os mesmos $\{0, 1, \ldots, m\}$ com 0 o estado inicial e 1 o estado final; o alfabeto dos símbolos da máquina de Turing podem, por sua vez ser codificados como inteiros, 0 para 'b', 1 para 'd' e 2 para |.

Definição 3.12 (Configuração e Situação de *M*) *Suponha-se que* M *é uma máquina de Turing com n bandas. A sequência infinita $C = (s_0, s_1, s_2, \ldots)$ aonde para todo o n, u e v com $0 \leqslant v < n$, s_{u+v} é o símbolo contido na célula $u + 1$ da banda $v + 1$, é designada por* Configuração de M no instante t. *A* situação de M no instante t *é o triplo $S(t) =< e, k, C(t) >$ aonde, para um dado instante t, 'e' é o estado da máquina, k é o número de células acima da posição corrente da cabeça da máquina e $C(t)$ é a configuração da máquina.*

A configuração da máquina de Turing no instante t, $C(t)$ é a sequência obtida ao justapor as sequências de símbolos nas diferentes da células da máquina de Turing.

$$C(t) = (\underbrace{s_1^1, s_1^2, \ldots, s_1^p}_{\delta_1}, \underbrace{s_2^1, s_2^2, \ldots, s_2^p}_{\delta_2}, \ldots \underbrace{s_k^1, s_k^2, \ldots, s_k^p}_{\delta_k}, \ldots$$

onde s_i^j é o i-ésimo símbolo da banda j.

Dado que $C(t)$ define-se como sendo uma configuração de uma dada máquina de Turing concreta a sequência de símbolos, qualquer sub-sequência de símbolos não nulos será sempre vazia.

Uma sequência infinita $C = (s_0, s_1, s_2, \ldots)$ mas com só um número finito de elementos não nulos ($\neq 0$) pode ser codificada da seguinte forma:

$$\Gamma(C) = \sum_{i \leqslant 0} s_i \cdot 3^i.$$

que é um número bem-definido.

No caso de sequências finitas, isto é nos casos em que k é o maior número tal que $s_k \neq 0$, tem-se:

$$\Gamma(C) = \sum_{0 \leqslant i \leqslant k} s_i \cdot 3^i.$$

Sabendo-se o código de $\Gamma(C)$ é possível recuperar os símbolos das diferentes posições. Sejam: $q(x,y)$, o quociente da divisão inteira de x por y e $r(x,y)$ o resto da divisão.

O símbolo contido na célula u da banda v é dado por:

$$r(q(\Gamma(C), 3^{n(u-1)+v-1}), 3)$$

A situação de M no instante t, $S(t) =< e, k, C(t) >$, pode ser codificada pelo inteiro:

$$\Gamma(S) = \alpha_3(e, k, \Gamma(C))$$

Lema 3.13 *Existe uma função recursiva primitiva $g \in \mathcal{F}_\infty$ tal que se x é o código da situação de uma dada máquina de Turing no instante t, então $g(x)$ é o código da situação da mesma máquina no instante $t+1$.*

A função g é definível por casos usando funções recursivas primitivas e conjuntos também recursivos parciais, como tal g será uma função recursiva primitiva (ver demonstração em [CL01].

Existem funções recursivas primitivas γ_i tal que $\gamma_i(C(s_0, s_1, \ldots)) = s_i$.

Pretende-se então formalizar a noção intuitiva de que se sabemos a situação da máquina num dado instante e sabemos como ela evolui, então podemos determinar qualquer situação

Definição 3.14 *Define-se, por recursão primitiva a função* $\mathrm{Sit}(t, x_1, x_2, \ldots, x_p)$ *como sendo:*

$$\begin{aligned} \mathrm{Sit}(0, x_1, x_2, \ldots, x_p) &= \alpha^3(0, 1, \Gamma(C)) \\ \mathrm{Sit}(t+1, x_1, x_2, \ldots, x_p) &= g(S(t, x_1, x_2, \ldots, x_p)) \end{aligned}$$

Aonde C é a configuração da máquina de Turing na qual a banda 1 representa x_1, a banda 2 representa x_2, e assim por adiante até a banda p que represente x_p, e com todas as outras bandas vazias.

Lema 3.15 *A função* $\mathrm{Sit}(t+1, x_1, x_2, \ldots, x_p)$ *é recursiva primitiva.*

Dem.: Dado a definição da função Sit é suficiente demonstrar que $\mathrm{Sit}(t+1, x_1, x_2, \ldots, x_p)$ é uma função recursiva primitiva de $\vec{x}$. Tem-se que:

$$\mathrm{Sit}(0, x_1, x_2, \ldots, x_p) = \alpha^3(0, 1, \Gamma(C))$$

aonde C é a configuração inicial das bandas, isto é contém os valores de $x_1, x_2, \ldots, x_p$ nas bandas $1, 2, \ldots, p$ respectivamente. Basta então mostrar que $\Gamma(C)$ é uma função recursiva primitiva em $\vec{x}$.

Seja $\rho(i,x)$ a função definida como sendo igual a 2 sempre que $i \leqslant x$ e igual a zero no caso contrário. Se $C = (s_0, s_1, \ldots, s_i, \ldots)$ é a configuração inicial da máquina, então s_i é o símbolo escrito na banda $r(i,n)+1$ na célula $q(i,n)$ temos então:

$$s_i = \begin{cases} 1 & \text{se } o \leqslant i \leqslant n-1, \\ \rho(q(i,n), x_{r(i,n)+1}) & \text{se } i \geqslant n; \end{cases}$$

consequentemente a definição de s_i pode ser feito por uma função recursiva primitiva, assim como a função $\Gamma(C)$ a qual é igual a:

$$\Gamma(C) = \sum_{i=0}^{i=(n+1)\cdot\sup(x_1, x_2, \ldots, x_p)} 3^i \cdot s_i.$$

□

Podemos agora enunciar e demonstrar o teorema que nos estabelece a relação entre funções T-computáveis e funções recursivas parciais.

Teorema 3.16 *Toda a função T-computável é recursiva parcial.*

Dem.: Seja $\mathcal{M}$ a máquina de Turing $\mathcal{M}$ que calcula $f(x_1, \ldots, x_p)$, isto significa que no momento em que $\mathcal{M}$ está no estado e_f, a banda $p+1$ representa $f(x_1, \ldots, x_p)$.

Vai-se calcular o *tempo de computação*, isto é o primeiro instante em que a máquina atinge o estado final.

$$T(x_1, \ldots, x_p) = \mu t[\beta_3^1(\mathrm{Sit}(t, x_1, \ldots, x_p)) = 1]$$

que está definido se e só se $f(x_1, \ldots, x_p)$ está definido; sabe-se então a situação da máquina no instante neste instante $T(x_1, \ldots, x_p)$, depois é só uma questão de contar o número de $|$ (barras/uns) escritos na banda $p+1$.

Seja α a função definida por:

$$\alpha(x) = \mu y[r(q(\beta_3^3(x), 3^{n\cdot(y+1)+p}, 3) = 0];$$

isto dado que se x é o código da situação da máquina $r(q(\beta_3^3(x), 3^{n\cdot(y+1)+p}), 3) = 0$ significa que o símbolo na célula $y+2$ é um nulo («blanck»). Isto é $\alpha(x)$ é o número de uns (ou $|$) no início da banda $p+1$.

Podemos então calcular f:

$$f(x_1, \ldots, x_p) = \alpha(\mathrm{Sit}(T(x_1, \ldots, x_p), (x_1, \ldots, x_p))).$$

Dado que tanto T como α são funções recursivas parciais então f é uma função recursiva parcial. □

Alguma observações acerca deste resultado que se acabou de demonstrar.

- Se a função $f \in \mathcal{F}_p$ é computável por uma máquina de Turing em tempo $T(x_1, \ldots, x_p)$ sendo que esta última função é uma função recursiva primitiva, então também a função f é recursiva primitiva.

  Este facto deve-se ao facto de que a função $\alpha(x)$ pode ser definida recorrendo simplesmente ao operador μ limitado, mais precisamente:

  $$\alpha(x) = \mu y \leqslant x[r(q(\beta_3^3(x), 3^{n\cdot(y+1)+p}, 3) = 0];$$

- O conjunto da funções recursivas parciais é quanto muito igual ao menor sub-conjunto $\mathcal{A}$ de $\mathcal{F}_p$ que contêm as funções recursivas primitivas e que seja fechado pela composição e o operador μ, ou dito de outra forma, caso se tenha já todas as funções recursivas primitivas as definições por recursão já não são necessárias. Seja $\mathcal{M}$ a máquina de Turing que calcula a função parcial $f \in \mathcal{F}_p$, a função parcial T que foi definida acima claramente pertence a $\mathcal{A}$ (a função Sit é recursiva primitiva) assim como f dado que esta é definida através de T e de funções recursivas primitivas.

- Seja $\mathcal{B}$ o sub-conjunto de $\mathcal{F}_p$ que contêm as funções recursivas primitivas e é fechado para a composição e o operador μ total (isto é, o operador μ somente para funções totais). Este conjunto é exactamente o conjunto das funções recursivas totais; seja f uma função recursiva total calculada pela máquina $\mathcal{M}$, a função T correspondente pertence a $\mathcal{B}$ e de igual modo f.

Nota 2 *Seja dada uma codificação das funções recursivas parciais (por exemplo como índices de máquinas de Turing). Então existe um predicado recursivo primitivo*[3] *T (o* predicado de Kleene*) e uma função recursiva primitiva U, tal que*

$$T(e, \vec{x}, y) \to U(y) = f(\vec{x}),$$

se e é o código da função f. Em T, y codifica o tempo da computação e *o resultado (parcial, se a computação ainda não terminou; final, se a computação já terminou).*

De facto, este predicado T é mais poderoso do que as funções consideradas na demonstração do teorema 3.16 porque é um *predicado para todos as funções recursivas parciais (que são dadas pelo argumento e): T é* universal *para as funções recursivas parciais. No seguinte parágrafo introduzimos um a função universal paras máquinas de Turing.*

3.1.4 Máquina de Turing Universal

Até agora falou-se sempre de uma máquina de Turing específica para cada uma das funções que se pretende calcular. Vai-se explorar agora a possibilidade de definir uma máquina única para toda e qualquer função T-computável.

Pretende-se então definir uma máquina que inclua, além dos valores dos parâmetros de entrada as instruções que ela tem de seguir. Para o fazer é essencial estabelecer uma codificação de uma máquina de Turing genérica.

Uma máquina de Turing é determinada por

- o número de bandas;
- O conjunto finito de estados E;
- M representa a tabela de transições $M : S^n \times E \to S^n \times E \times \{-1, 0, 1\}$.

Vai-se assumir que o conjunto dos estados E é da forma $\{0, 1, 2, \ldots, m\}$, com $e_i = 0$ e $e_f = 1$.

Para codificar a máquina de Turing é necessário então codificar M.

Para toda a sequência $\sigma = (s_1, s_2, \ldots, s_n, e)$ vai-se definir:

$$\begin{aligned} r_1 &= \alpha_2(\Gamma(s_1, s_2, \ldots, s_n), e); \\ r_2 &= \alpha_3(\Gamma(t_1, t_2, \ldots, t_n), e', \epsilon + 1) \\ &\quad \text{aonde } (t_1, t_2, \ldots, t_n), e', \epsilon + 1) = M(\sigma); \\ n(\sigma) &= [\pi(r_1)]^{r_2}. \end{aligned}$$

[3] Mais tarde, vamos estudar *predicados* em mais detalhe. Mas já podemos dizer que um predicado pode ser identificado com uma *função característica* desse predicado. Então chamamos um predicado *recursivo primitivo* se a sua função característica é recursiva primitiva.

Relembrar que $\pi(i)$ é a função que nos dá o i-ésimo+1 número primo.

O código da tabela M é dado pelo inteiro u:

$$u = \prod_{p \in S^n \times E} n(p)$$

É fácil recuperar M a partir da sua codificação. Se se quer saber $(t_1, t_2, \ldots, t_n, e', \epsilon) = M(s_1, s_2, \ldots, s_n, e)$, começa-se por calcular $c = \Gamma(s_1, s_2, \ldots, s_n)$ e $r = \alpha_2(c, e)$, depois usa-se a função δ previamente definida (3.11):

$$\delta(r, u) = \alpha_3(c', e', \epsilon + 1) \qquad \text{aonde } c' = \Gamma(t_1, t_2, \ldots, t_n)$$

e a descodificação pode então ser calculada sem dificuldade.

Definição 3.17 (Indice de $\mathcal{M}$) *O índice da máquina $\mathcal{M}$ é um inteiro $\alpha_3(n, m, u)$ com:*

- *n - o número de bandas de $\mathcal{M}$;*
- *$m + 1$ - o número de estados de $\mathcal{M}$;*
- *u - é o código da tabela de transições de $\mathcal{M}$.*

Define-se também o conjunto:

$$I_p = \{x \mid x \text{ é o índice de } \mathcal{M} \text{ com pelo menos } p + 1 \text{ bandas}\}$$

Agora vamos definir para qualquer n um conjunto de índices de máquinas de Turing que tem (pelo menos) n argumentos:

I_n é um conjunto *recursivo primitivo*, isto é, a sua função característica é recursiva primitiva.

Função Recursiva Parcial Universal (para Máquinas de Turing)

Definição 3.18 *Pretende-se definir a função que caracteriza a situação da máquina de índice i:*

$$ST^p(i, t, x_1, \ldots, x_p) = \begin{cases} \Gamma(S(t)), & i \in I_p \\ 0, & i \notin I_p \end{cases}$$

Relembrando que $\Gamma(S(t))$ é o código no instante t da situação de uma máquina com índice i que começa a operar no instante $t = 0$ contendo nesse instante $x_1, x_2, \ldots, x_p$ nas bandas $1, 2, \ldots, p$, respectivamente, sendo que todas as outras bandas estão vazias.

Teorema 3.19 *Para todo o inteiro p a função $ST^P(i, t, x_1, \ldots, x_p)$ é primitiva recursiva.*

Para a demonstração ver [CL01].

Pretende-se de seguida demonstrar que a situação de uma máquina de Turing no instante t é uma função recursiva primitiva $ST^p(i, t, x_1, \ldots, x_p)$ do seu índice, a situação inicial e t.

Definição 3.20 *Para todo o $p > 0$ define-se:*

$$\Phi^n(i, x_1, \ldots, x_p) = \begin{cases} \bot, & i \notin I_p; \\ \bot, & i \in I_p \text{ e a máquina de Turing com índice } i \text{ não pára}; \\ n, & i \in I_p \text{ a máquina de Turing com índice } i \text{ pára} \\ & \text{e } n \text{ é o número representado na banda } p+1. \end{cases}$$

Teorema 3.21 (Enumeração) *Para todo o natural p, Φ^p é uma função recursiva parcial e se f é uma função recursiva parcial de p variáveis então existe um i tal que para todos os $x_1, x_2, \ldots, x_p$ se tem $f(x_1, x_2, \ldots, x_p) = \Phi^n(i, x_1, x_2, \ldots, x_p)$.*

Dem.: Vamos introduzir uma função recursiva parcial T^p e predicados recursivos primitivos B^p e C^p.

$T^p(i, x_1, \ldots, x_p)$ é o tempo de computação da máquina de índice i e com argumentos $x_1, \ldots, x_p$ se $i \in I_p$, é indefinido se a computação não para.

$$T^p(i, x_1, \ldots, x_p) = \mu t[\beta_3^1(ST^p(i, t, x_1, \ldots, x_p)) = 1].$$

Note-se que se $i \notin I_p$ então $T^p(i, x_1, \ldots, x_p)$ é indefinido. Para todo o natural p faz-se:

$$\begin{aligned} B^p &= \{(i, t, x_1, \ldots, x_p) \mid ST^p(i, t, x_1, \ldots, x_p) =\} \\ B^p(i) &= \{(t, x_1, \ldots, x_p) \mid (i, t, x_1, \ldots, x_p) \in B^p\} \end{aligned}$$

Estes conjuntos são recursivos primitivos e $(i, t, x_1, \ldots, x_p) \in B^p$ significa que esta máquina, com índice i e argumentos $x_1, \ldots, x_p$ nas bandas $1, \ldots, p$ e todas as outras bandas vazias completa o seu funcionamento no tempo t.

Continuando com a definição dos elementos necessários à demonstração, temos:

$$C^p = \{(i, y, t, x_1, \ldots, x_p) \mid i \in I_p, (i, t, x_1, \ldots, x_p) \in B^p\}$$

sendo que y é o número representado no banda $p+1$ com a máquina nas condições já referidas acima.

$$C^p(i) = \{(y, t, x_1, \ldots, x_p) \mid (i, y, t, x_1, \ldots, x_p) \in C^p\}$$

De novo é fácil de mostrar que estes conjuntos são recursivos primitivos.

Estamos agora em condições de definir a função parcial φ^p:

$$\varphi^p(i, x_1, \ldots, x_p) = \mu y[(i, y, T^p(i, x_1, \ldots, x_p), x_1, \ldots, x_p) \in C^p].$$

A função φ^p é recursiva. □

Dado que a função φ^p é recursiva é então computável por uma dada máquina de Turing e temos então que essa máquina pode calcular todas as funções recursivas parciais de p variáveis.

Para todo o natural i seja $\varphi_i^p(x_1, \ldots, x_p) = \varphi^p(i, x_1, \ldots, x_p)$, podemos então definir o conjunto $\{\varphi_i^p \mid i \in \mathbb{N}\}$ o conjunto de todas as funções recursivas parciais de p argumentos.

Definição 3.22 *Seja $f \in \mathcal{F}_p^*$ uma função recursiva parcial. Diz-se que $i \in \mathbb{N}$ é um índice de f se $f = \varphi_i^p$.*

Definição 3.23 (igualdade parcial) *$t \simeq s$ se sempre que t está bem definido ou s está bem definido, então $t = s$.*

Lema 3.24 *No caso em que t não está bem definido e $t \simeq s$, então s não está também bem definido.*

Outra forma de escrever o teorema da Enumeração, é a seguinte: defina-se, $\{i\}(x_1, \ldots, x_p) = \Phi^n(i, x_1, \ldots, x_n)$. Para cada função recursiva parcial f existe um natural i, tal que qualquer que sejam os $x_1, \ldots, x_p$ se tem $f(x_1, \ldots, x_p) \simeq \{i\}(x_1, \ldots, x_p)$.

3.2 Conjuntos Recursivos Enumeráveis

Definição 3.25 (Conjuntos Recursivos & Recursivos Enumeráveis) *Seja A um sub-conjunto de $\mathbb{N}^n$ ($A \subseteq \mathbb{N}^n$).*

A é recursivo *se a sua função característica χ_A é recursiva (total).*

A é recursivo enumerável *se A é o domínio de uma função recursiva parcial.*

A segunda definição pode ser motivada na forma que existe um teste (uma função recursiva parcial) que responde *em tempo finito* positivamente se $x \in A$; no caso negativo ($x \notin A$) este teste pode não terminar.

O domínio de uma função parcial com índice x, i.e. ϕ_x^p, vai-se designar por W_x^p. O conjunto $\{W_x^p \mid x \in \mathbb{N}\}$ é o conjunto de todos os sub-conjuntos de $\mathbb{N}^p$ que são recursivos enumeráveis. Se $A = W_x^p$ diz-se que x é o índice de A.

Lema 3.26 *Todo o conjunto recursivo é recursivo enumerável.*

Dem.: Seja $f(x) = \mu y[y + 1 = x]$. A função f é recursiva parcial, f é indefinida para $x = 0$ e definida para todos os outros casos.

Então a função $f \circ \chi_A$ é uma função recursiva parcial cujo domínio é o conjunto A.

□

Lema 3.27 *Para cada p, o conjunto dos sub-conjuntos recursivos de $\mathbb{N}^p$ é fechado para as operações Booleanas ($\cap, \cup, \backslash$).*

Definição 3.28 *Temos que:*

$$\chi_{A \cap B} = \chi_A \cdot \chi_B$$

$$\chi_{A \cup B} = \mathrm{sg}(\chi_A + \chi_B)$$

com sg *a função recursiva primitiva definida do seguinte modo:*

$$\mathrm{sg}(x) = \begin{cases} 0, & x = 0 \\ 1, & x \neq 0 \end{cases}$$

$$\chi_{\mathbb{N}^p \backslash A} = 1 \dot{-} \chi_A$$

com $\dot{-}$ a função recursiva primitiva definida do seguinte modo:

$$\dot{-}(x,y) = \begin{cases} x - y, & x \geqslant y \\ 0, & x < y \end{cases}$$

Lema 3.29 *A união e a intersecção de dois conjuntos recursivos enumeráveis é recursivo enumerável.*

Dem.: Sejam A_1 e A_2 sub-conjuntos recursivos enumeráveis de $\mathbb{N}^p$, domínios das funções parciais f_1 e f_2 respectivamente, sendo que estas funções são calculadas por máquinas de Turing com índices i_1 e i_2 respectivamente.

É evidente que $A_1 \cap A_2$ é o domínio de $f_1 + f_2$. Por outro lado $A_1 \cup A_2$ é o domínio da função parcial

$$\mu t[(t, x_1, x_2, \ldots, x_p) \in B^p(i_1) \cup B^p(i_2)]$$

que é recursiva dado que $B^p(i)$ é recursiva primitiva. □

Teorema 3.30 *Seja $A \subseteq \mathbb{N}^p$. A é recursivo se e só se A e $\mathbb{N}^p \backslash A$ são ambos recursivos enumeráveis.*

Dem.: Num dos sentidos é quase que imediato.

A é recursivo então também $\mathbb{N}^n \backslash A$ (ver lema 3.27), então pelo lema 3.26 são ambos recursivos enumeráveis.

Vejamos agora no sentido oposto.

Seja i o índice da máquina que calcula a função parcial cujo domínio é A, seja, i' o índice da máquina que calcula a função parcial cujo domínio é $\mathbb{N}^p \backslash A$. Então:

$$h(x_1, \ldots, x_p) = \mu t[(t, x_1, \ldots, x_p) \in B^p(i) \cup B^p(i')]$$

é uma função recursiva total e

$$(x_1, \ldots, x_p) \in \text{Asse}(h(x_1, \ldots, x_p), x_1, \ldots, x_p) \in B^p(i)$$

então se $\chi_{(t,x_1,\ldots,x_p)}$ é a função característica de $B^p(i)$, então a função característica de A é:

$$\chi_(h(x_1, \ldots, x_p), x_1, \ldots, x_p),$$

o que mostra que A é recursivo.

□

Teorema 3.31 *A projecção de um conjunto recursivo enumerável é um conjunto recursivo enumerável.*

Dem.: Por projecção entende-se: se $A \subseteq \mathbb{N}^{p+1}$ então, $B = \{(x_1, \ldots, x_n) \mid$ existe x_0 tal que $(x_0, x_1, \ldots, x_n) \in A\}$, é uma projecção de A.

Seja i o índice da máquina que calcula a função cujo domínio é A, sendo assim temos que:

$(x_0, x_1, \ldots, x_p) \in A$ sse existe um t tal que $(t, x_0, \ldots, x_p) \in B^p(i)$ e $(x_1, \ldots, x_p) \in B$ sse existe um t e um x_0 tais que $(t, x_0, x_1, \ldots, x_n) \in B^p(i)$.

Isto mostra que B é o domínio de uma função recursiva parcial:

$$g(x_1, \ldots, x_p) = \mu z[(\beta_2^1(z), \beta_2^2(z), x_1, \ldots, x_n) \in B^p(i)]$$

□

Teorema 3.32 *Todo o sub-conjunto recursivo enumerável $A \subseteq \mathbb{N}^p$ é a projecção de um conjunto recursivo* primitivo $B \subseteq \mathbb{N}^{p+1}$.

Dem.: Isto significa que se $A \subseteq \mathbb{N}^p$ é recursivo enumerável então existe um sub-conjunto recursivo primitivo $B \subseteq \mathbb{N}^{p+1}$ tal que:

$$(x_1, \ldots, x_p) \in A$$

sse

$$\text{existe um } x_0 \text{ tal que } (x_0, x_1, \ldots, x_p) \in B$$

É suficiente tomar para B o conjunto $B^p(i)$ aonde i é o índice da máquina que calcula a função cujo domínio é A.

□

Corolário 3.33 *O grafo de uma função recursiva parcial é um conjunto recursivo enumerável.*

Dem.: Por grafo de uma função $f \in \mathcal{F}_1$ entende-se o conjunto formado pelos pares $(x, f(x))$.
Seja $f \in \mathcal{F}_p^*$.
Temos que demonstrar que

$$G = \{(x_1, \ldots, x_p, y) \mid y = f(x_1, \ldots, x_p)\}$$

é recursivo enumerável.

Seja i o índice da máquina que calcula f, vê-se então que $(x_1, \ldots, x_p, y) \in G$ se e só se existe um t tal que $(y, t, x_1, \ldots, x_p) \in C^p(i)$; isto mostra que G é a projecção de um conjunto primitivo recursivo, e como tal é recursivo enumerável (3.31).

□

O resultado recíproco é também verdadeiro. Se G, o grafo da função parcial f, é recursivo enumerável então f é recursiva parcial: existe um conjunto recursivo primitivo tal que $(x_1, x_2, \ldots, x_p, y) \in G$ se e só se existe um t tal que $(x_1, x_2, \ldots, x_p, y, t) \in A$, temos então que:

$$f(x_1, x_2, \ldots, x_p) = \beta_2^1(\mu t[(x_1, x_2, \ldots, x_p, \beta_2^1(t), \beta_2^2(t)) \in A])$$

O contra-domínio de f é ele próprio a projecção do grafo de f; em consequência disso tem-se,

Corolário 3.34 *O contradomínio de uma função recursiva parcial é um conjunto recursivo enumerável.*

O recíproco é também verdadeiro; de facto temos o seguinte resultado mais forte.

Corolário 3.35 *Cada conjunto recursivo enumerável e não vazio de* $\mathbb{N}$ *é o contradomínio de uma função recursiva primitiva em* $\mathcal{F}_1$.

Dem.: Seja $A \neq \emptyset$ um sub-conjunto recursivo enumerável; escolha-se $n \in A$ e seja i o índice de A.

Tem-se então que: $x \in A$ sse existe $t \in \mathbb{N}$ tal que $(t, x) \in B^1(i)$.

É fácil de verificar que A é o contradomínio da função recursiva primitiva g definida por:

$$g(z) = \begin{cases} \beta_2^2(z), & \text{se } (B_2^1(z), \beta_2^2(z)) \in B^1(i) \\ n, & \text{se } (B_2^1(z), \beta_2^2(z)) \notin B^1(i) \end{cases}$$

□

O resultado seguinte generaliza o princípio da definição por casos para o contexto das funções recursivas parciais.

Teorema 3.36 *Sejam* $g(x_1, \ldots, x_p)$ *e* $g'(x_1, \ldots, x_p)$ *duas funções recursivas parciais e seja* A *um conjunto recursivo. Então a função* f *definida por:*

$$f(x_1, \ldots, x_p) = \begin{cases} g(x_1, \ldots, x_p), & \textit{se } x_1, \ldots, x_p \in A, \\ g'(x_1, \ldots, x_p), & \textit{no caso contrário} \end{cases}$$

é recursiva parcial.

Nota 3 *Se* $(x_1, \ldots, x_p) \in A$ *então* $f(x_1, \ldots, x_p)$ *está definida se e somente se* $g(x_1, \ldots, x_p)$ *está definida e nesse caso são iguais. O mesmo comentário aplica-se para o caso* g'.

Isto é, a igualdade $f = g \cdot \chi_A + g' \cdot \chi_{\mathbb{N} \setminus A}$ *só é verdadeira quando* g *e* g' *estão ambas bem definidas.*

Dem.: Sejam i e i' os índices de de g e de g'. Considere-se o seguinte conjunto $C \subseteq \mathbb{N}^{p+2}$.

$$\begin{aligned} C \;=\; \big\{(y, t, x_1, \ldots x_p) \mid & ((y, t, x_1, \ldots, x_p) \in C^P(i) \quad \text{e} \quad (x_1, \ldots x_p) \in A) \\ & \text{ou} \quad ((y, t, x_1, \ldots, x_p) \in C^P(i') \quad \text{e} \quad (x_1, \ldots x_p) \notin A)\big\} \end{aligned}$$

o conjunto C assim definido, é recursivo.

O significado de $(y, t, x_1, \ldots, x_p) \in C$ é que, ou $(x_1, \ldots, x_p) \in A$ e a máquina de índice i finaliza a computação no instante t com valor y, ou então $(x_1, \ldots, x_p) \notin A$ e a máquina com índice i' finaliza a computação no instante t com valore y.

Temos então que $f(x_1, \ldots, x_p)$ é igual ao menor y tal que existe um t para os quais $(y, t, x_1, \ldots, x_p) \in C$. Isto implica que:

$$f(x_1, \ldots, x_p) = \beta_2^1(\mu z[(\beta_2^1(z), \beta_2^2(z), x_1, \ldots, x_p) \in C]).$$

o que mostra que f é uma função recursiva parcial. □

3.2.1 O Problema da Paragem

Com o problema de paragem pretende-se responder à questão: dado um índice de uma máquina de Turing (i.e. o seu conjunto de instruções) e a sua configuração inicial, é possível saber se esta termina em tempo finito, ou não termina sequer?

Se fosse possível, para todos os casos, responder a esta questão afirmativamente dir-se-ia que o problema de paragem era decidível. Infelizmente vamos verificar já de seguida que o mesmo é indecidível, isto é, não é possível dizer se a máquina pára, ou não.

Antes de poder estabelecer este resultado é necessário verificar outros resultados preliminares.

Existem conjuntos recursivos enumeráveis que não são recursivos? Ou doutro modo, existem conjuntos recursivos enumeráveis cujo complemento não seja um conjunto recursivo enumerável?

A resposta é positiva, considere-se a função $\phi^1(i,x)$ e fixemos $g(x) = \phi^1(x,x)$. Seja então A o domínio de g.

O conjunto A é recursivo enumerável, mas o seu complementar não. Vejamos que assim é.

Para todo o $x \in \mathbb{N}$, $x \in A \equiv x \in W^1_x$, o qual é recursivo enumerável.

Vejamos então que $\mathbb{N} \backslash A$ não é recursivo enumerável.

Suponha-se o contrário: existe um natural n tal que $\mathbb{N} \backslash A = W^1_n$ isto é, tal que para todo o natural x:

$$x \neq A \quad sse \quad x \in W^1_n$$

mas então tem-se que para $x = n$ verifica-se:

$$n \in A \quad sse \quad n \in W^1_n$$

o que é um absurdo, logo $\mathbb{N} \backslash A$ não é recursivo enumerável, logo A é recursivo enumerável mas não é recursivo.

Corolário 3.37 *O conjunto* $\left\{(m,x) \mid \phi^1(m,x) \text{ está definido}\right\}$ *não é recursivo.*

Dem.: Se o conjunto em questão fosse recursivo então o conjunto

$$\left\{(x,x) \mid \phi^1(x,x) \text{ está definido}\right\}$$

seria recursivo. Já se viu que tal não é o caso. □

Podemos então dizer que para $A \subseteq \mathbb{N}$, A é recursivo se existe um algoritmo que permita decidir se um dado inteiro pertence a A ou não.

A é recursivo enumerável se existe um algoritmo que enumera A. Se se coloca a questão de saber se um dado natural pertence a A, podemos dizer que se n está na sequência que enumera A então $n \in A$, por outro lado enquanto n não surge na sequência, nada se pode afirmar sobre a pertença, ou não, desse natural ao conjunto A.

Definição 3.38 (Decidível) *Seja* $B(x_1, x_2, \ldots, x_p)$ *uma propriedade que se aplica aos naturais* $x_1, x_2, \ldots, x_p$. *O problema «a sequência* $(x_1, x_2, \ldots, x_p)$ *satisfaz* B*?» é dito decidível se o conjunto das sequências* $(x_1, x_2, \ldots, x_p)$ *para os quais* $B(x_1, x_2, \ldots, x_p)$ *é verdade é um conjunto recursivo.*

Podemos então afirmar que: o problema de paragem para máquinas de Turing é indecidível.

O Teorema *smn*

Sejam $f \in \mathcal{F}^*_{m+n}$ com índice i e $a_1, a_2, \ldots, a_n$ (fixos). E seja $g \in \mathcal{F}^*_m$ com $g(y_1, \ldots, y_m) = f(a_1, \ldots, a_n, y_1, \ldots, y_m)$. Será possível calcular um índice para g?

Teorema 3.39 (Teorema snm**)** *Para todo o par de naturais m, n existe uma função recursiva primitiva s^m_n de $n+1$ variáveis tal que para todo o $i, x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m$ tem-se:*

$$\phi^{n+m}(i, x_1, \ldots, x_n, y_1, \ldots, y_m) = \phi^m(s^m_n(i, x_1, \ldots, x_n), y_1, \ldots, y_m).$$

Dem.: O valor de $s^m_n(i, x_1, \ldots, x_n)$ vai ser definido por casos de acordo com o $i \in I_{n+m}$, ou não.

Primeiro caso: $i \notin I_{n+m}$. Faz-se $s^m_n(i, x_1, \ldots, x_n) = i_0$, para este caso temos que: nem $\phi^{n+m}(i, x_1, \ldots, x_n, y_1, \ldots, y_m)$, nem $\phi^m(s^m_n(i, x_1, \ldots, x_n), y_1, \ldots, y_m)$ são definidos.

Segundo caso: $i \in I_{n+m}$. Seja $\mathcal{M}$ a máquina cujo índice é i e sejam $a_1, a_2, \ldots, a_n$ inteiros fixos. É possível então conceber a máquina $\mathcal{M}'$ que tem o mesmo número de fitas que $\mathcal{M}$ e que tem o seguinte comportamento:

- começa por escrever a_1 `1s` na fita $m+2$, a_2 `1s` na fita $m+3$, etc., e finalmente a_n `1s` na fita $m+n+1$.
- depois o seu comportamento é idêntico ao comportamento de $\mathcal{M}$ embora com os papéis das fitas 1 a n trocados com os papéis das fitas de $m+2$ a $m+n+1$. O resultado final ficará na fita $m+n+1$.
- finalmente apaga os conteúdos das bandas $m+2, m+3, \ldots, m+n$.

Para esta máquina é claro que.

Em primeiros lugar a descrição de $\mathcal{M}'$ é completamente explícita e efectivamente baseada na descrição de $\mathcal{M}$ e dos $a_1, a_2, \ldots, a_n$ dados. Seria possível encontrar a função recursiva primitiva de $n+1$ variáveis, $s^m_n(i, x_1, x_2, \ldots, x_n)$, cujo valor é o índice da máquina $\mathcal{M}'$.

Em segundo lugar, se se colocarem as máquinas $\mathcal{M}$ e $\mathcal{M}'$ em funcionamento com as seguintes configurações iniciais:

- para k de 1 até n inclusive, os conteúdos da fita k de $\mathcal{M}'$ é igual ao conteúdo da fita $n+k$ de $\mathcal{M}$;
- todas as outras fitas de $\mathcal{M}'$ estão vazias;
- a_k, com $1 \leqslant k \leqslant n$ é representado na fita k de $\mathcal{M}$.

então, a menos de uma permutação da ordem das fitas, as duas máquinas operam exactamente da mesma maneira, nomeadamente uma pára se se somente se a outra também pára. Após terminarem o conteúdo da fita $m+1$ de $\mathcal{M}'$ será igual ao conteúdo da fita $n+m+1$ de $\mathcal{M}$. Consequentemente tem-se que para todos os $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m$,

$$\phi^{n+m}(i, x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m) = \phi^m(s_n^m(i, x_1, x_2, \dots, x_n), y_1, y_2, \dots, y_m).$$

□

Vejamos agora algumas aplicações deste teorema.

Exemplo 3.40 *Existe uma função recursiva primitiva* $pl(i, j)$ *tal que, se* $f = \phi_i^1$ *e* $g = \phi_j^1$*, então* $pl(i, j)$ *é um índice para a função parcial* $f + g$*.*

Dem.: Considere-se a função parcial

$$h(i, j, x) = \phi^1(i, x) + \phi^1(j, x).$$

que é obviamente recursiva, então existe um inteiro k tal que esta função é igual a ϕ_k^3. Então, para todo o i, j e x tem-se,

$$\phi_k^3(i, j, x) = \phi^3(k, i, j, x) = \phi^1(s_2^1(k, i, j), x) = \phi^1(i, x) + \phi^1(j, x)$$

basta então tomar $pl(x, y) = s_2^1(k, i, j)$. □

Ter-se-ia um resultado semelhante se se considera-se a multiplicação ou uma outra qualquer função recursiva parcial.

Exemplo 3.41 *Sejam* n *e* p *inteiros. Existe uma função recursiva primitiva* $\mathrm{comp}(i_1, i_2, \dots, j)$ *tal que, se para* $1 \leqslant k \leqslant n$ $f_k \in \mathcal{F}_p^*$ *é a função parcial cujo índice é* i_k*, e se* $g \in \mathcal{F}_p^*$ *é a função parcial cujo índice é* j*, então* $\mathrm{comp}(i_1, i_2, \dots, j)$ *é o índice para a função parcial* $h = g(f_1, f_2, \dots, f_n)$*.*

Dem.: A demonstração é idêntica ao exemplo anterior. Considere-se a função parcial

$$\begin{aligned} & l(i_1, i_2, \dots, i_n, j, x_1, x_2, \dots, x_p) = \\ & \quad \phi^n(j, \phi^p(i_1, x_1, x_2, \dots, x_p), \phi^p(i_2, x_1, x_2, \dots, x_p), \dots, \phi^p(i_n, x_1, x_2, \dots, x_p)) \end{aligned}$$

é recursiva, então existe um inteiro k tal que esta função parcial é igual a ϕ_k^{n+p+1}.

Temos então

$$\phi_k^{n+p+1}(k, i_1, i_2, \dots, i_n, j, x_1, x_2, \dots, x_p) = \phi^p(s_{n+1}^p(k, i_1, i_2, \dots, i_n, j), x_1, x_2, \dots, x_p),$$

. e podemos definir a função comp da seguinte forma.

$$\mathrm{comp}(i_1, i_2, \dots, i_n, j) = s_{n+1}^p(k, i_1, i_2, \dots, i_n, j)$$

□

O teorema que vamos enunciar e demonstrar de seguida, o teorema de Rice, é ainda uma outra aplicação do teorema *smn*.

Teorema 3.42 (Teorema de Rice) *Seja χ um conjunto de funções recursivas parciais de uma variável que se vai assumir não vazio e não igual ao conjunto de todas as funções recursivas parciais. Então o conjunto $A = \{x \mid \phi^1_x \in \chi\}$ não é recursivo.*

Dem.: É equivalente mostrar que A, ou o seu complementar, é não recursivo: sendo assim intercambiando esses dois conjuntos, se necessário, e trocando χ pelo seu complementar em relação ao conjunto de todas as funções recursivas parciais de uma só variável, podemos assumir que a função parcial ϕ_0 cujo domínio é vazio é um elemento de χ.

Fixe-se um inteiro b que não pertence a A, e defina-se a seguinte função recursiva parcial $\psi \in \mathcal{F}^*_3$:

$$\psi(x, y, z) = \phi^1(b, z) + \phi^1(x, y) - \phi^1(x, y)$$

Faça-se também

$$\psi_{x,y}(z) = \psi(x, y, z).$$

If $\phi^1(x, y)$ não está definido, a função parcial $\psi_{x,y}$ nunca está definida (e como tal é igual a ϕ_0, e consequentemente está em χ; caso contrário, $\psi_{x,y}$ é igual a ϕ^1_b, sendo assim não está em χ. Em consequência podemos afirmar que $\psi_{x,y}$ pertence a χ se e só se $\phi^1(x, y)$ é indefinida. Aplicando o teorema *smn*, existe um inteiro k tal que:

$$\psi(x, y, z) = \phi^3(k, x, y, z) = \phi^1(s^1_2(k, x, y), z).$$

A função $h(x, y) = s^1_2(k, x, y)$ é recursiva primitiva e $h(x, y)$ é um índice de $\psi_{x,y}$.

Considerando o conjunto $W = \{(x, y) \mid \phi^1(x, y)$é indefinido$\}$ não é recursivo (ver lema 3.37) e constatando que $(x, y) \in W$ se e só se $h(x, y) \in A$ temos que A não é recursivo, porque se o fosse, então também W o era. □

Eis alguns resultados que resultam directamente do teorema de Rice.

- Se $f \in \mathcal{F}^*_p$ é uma função recursiva parcial, o conjunto dos índices de f não é recursivo. Basta fazer $\chi = \{f\}$ no teorema de Rice. Em particular não é finito.

  Intuitivamente, se uma função parcial é computável, então existem infinitas máquinas que a calculam. Além disso podemos ainda afirmar que não é possível ter uma descrição efectiva para o conjunto de todas as máquinas que calculam f.

- O problema de decidir quando é que duas máquinas diferentes calculam a mesma função parcial é indecidível. Para todo o inteiro p o conjunto

  $$X = \{(i, j) \mid \phi^p_i = \phi^p_j\}$$

  não é recursivo.

  Se o conjunto X fosse recursivo, então o conjunto

  $$\{i \mid (i, 0) \in X\} = \{i \mid \phi^p_i = \phi^p_0\}$$

  também seria recursivo, o que se provou acima que não é o caso.

- O conjunto $\{n \mid \phi_i^p \text{é total}\}$ não é recursivo.

  Basta tomar para χ o conjunto de todas as funções recursivas totais. De acordo com o primeiro destes resultados, se uma função parcial tem um índice i, tem um outro índice que é maior do que i. O teorema que se segue estabelece isso mesmo.

Teorema 3.43 *Para todo o inteiro p existe uma função recursiva primitiva α de duas variáveis tal que:*

- *para todo o i e n, $\phi_i^p = \phi_{\alpha(i,n)}^p$;*
- *para todo o i, a função $f(n) = \alpha(i,n)$ é estritamente crescente.*

A demonstração deste resultado encontra-se em [CL01].

3.2.2 Os Teoremas do Ponto Fixo

Os resultados que se vão apresentar de seguida são devidos a S. Kleene e são também designados por teoremas da recursão.

Teorema 3.44 (Teorema do Ponto Fixo (1ª versão)) *Seja p um natural não nulo e seja α uma função de uma variável, recursiva e total. Então existe um inteiro i tal que:*

$$\phi_i^p = \phi_{\alpha(i)}^p.$$

Dem.: Considere-se a função parcial

$$f(y, x_1, \dots, x_p) = \phi^p(\alpha(s_1^p(y,y)), x_1, \dots, x_p)$$

esta função é recursiva, então tem um índice a e tem-se que, para todo o $x_1 \dots, x_p$ e y

$$\begin{aligned} \phi^{p+1}(a, y, x_1, \dots, x_p) &= \phi^p(\alpha(s_1^p(y,y)), x_1, \dots, x_p) \\ &= \phi^p(s_1^p(a,y), x_1, \dots, x_p) \end{aligned}$$

fazendo $y = a$ nas igualdades acima e fixando $i = s_1^p(a,a)$ obtêm-se

$$\phi_i^p = \phi_{\alpha(i)}^p$$

como pretendíamos. □

Teorema 3.45 (Teorema do Ponto Fixo (2ª versão)) *Para todo o natural não nulo p, existe uma função de uma variável recursiva e primitiva h_p tal que para todo o j, se $\alpha = \phi_j^i$ é uma função total, então*

$$\phi_{h_p(j)}^p = \phi_{\alpha(h_p(j))}^p.$$

Dem.: Suponha-se que $\alpha = \phi_j^p$. Começa-se por calcular um índice a da seguinte função parcial:

$$f(y, x_1, \dots, x_p) = \phi^p(\alpha(s_q^p(y,y), x_1, \dots, x_p)$$

Seja b um índice da seguinte função parcial

$$g(j, y, x_1, \ldots, x_p) = \phi^p(\phi^1(j, s_1^p(y, y)), x_1, \ldots, x_p)$$

então tem-se que para todo o $x_1, \ldots, x_p$

$$\begin{aligned}\phi^p(\alpha(s_1^p(y,y)), x_1, \ldots, x_p) &= \phi^p(\phi^1(j, s_1^p(y,y), x_1, \ldots, x_p)\\ &= \phi^{p+2}(b, j, y, x_1, \ldots, x_p)\\ &= \phi^{p+1}(s_1^{p+1}(b,j), y, x_1, \ldots, x_p)\end{aligned}$$

basta então fixar $a = s_1^{p+1}(b, j)$ e $i = s_1^p(a, a)$ e obtêm-se o resultado pretendido. □

Teorema 3.46 (Teorema do Ponto Fixo (3ª versão)) *Seja α uma função recursiva total de $p+1$ variáveis e seja n e p inteiros com $n > 0$. Então existe uma função recursiva primitiva h de p variáveis tal que para todo o $x_1, \ldots, x_p$ tem-se*

$$\phi^n_{h(x_1,\ldots,x_p)} = \phi^n_{\alpha(x_1,\ldots,x_p,h(x_1,\ldots,x_p))}.$$

Dem.: Seja a um índice da função parcial

$$f(z, x_1, \ldots, x_p, y_1, \ldots, y_n) = \phi^n(\alpha(x_1, \ldots, x_p, s^n_{p+1}(z, z, x_1, \ldots, x_p)), y_1, \ldots, y_n)$$

para todo os $x_1, \ldots, x_p, y_1, \ldots, y_n, z$ tem-se

$$\begin{aligned}\phi^n(\alpha(x_1, \ldots, x_p, s^n_{p+1}(z, z, x_1, \ldots, x_p)), y_1, \ldots, y_n) &= \phi^{n+p+1}(a, z, x_1, \ldots, x_p, y_1, \ldots, y_n)\\ &= \phi^n(s^n_{p+1}(a, z, x_1, \ldots, x_p), y_1, \ldots, y_n)\end{aligned}$$

fixando $z = a$ obtêm-se

$$\phi^n(\alpha(x_1, \ldots, x_p, s^n_{p+1}(z, z, x_1, \ldots, x_p)), y_1, \ldots, y_n) = \phi^n(s^n_{p+1}(a, a, x_1, \ldots, x_p), y_1, \ldots, y_n)$$

basta então fazer $h(x_1, \ldots, x_p) = s^n_{p+1}(a, a, x_1, \ldots, x_p)$ para se obter o resultado pretendido. □

Exemplo de aplicação destes resultados Seja $f(x, y)$ uma função parcial de duas variáveis que é definida por recursão em y.

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

aonde g e h são funções recursivas parciais. Então é possível calcular (de uma forma recursiva primitiva) um índice para f a partir de índices para g e h.
Dem.: Considere-se a aplicação de $\mathcal{F}_2^*$ para $\mathcal{F}_2^*$ que aplica ψ a função parcial ψ^* definida do seguinte modo

$$\psi^*(x, y) = \begin{cases} g(x), & y = 0\\ h(x, y-1, \psi(x, y-1)), & y \neq 0\end{cases}$$

f é o único ponto fixo desta atribuição, é a única função parcial que verifica $f = f^*$. Além disso se ψ é uma função recursiva então ψ^* também é uma função recursiva, pode-se então calcular o índice para ψ^* a partir dos índices i_1, i_2 e i_3 de g, h e ψ respectivamente, o que é possível pelo teorema *smn*.

Considere-se a função recursiva parcial $k(i_1, i_2, i_3, x, y)$ definida do seguinte modo

$$k(i_1, i_2, i_3, x, y) = \begin{cases} \phi^1(i_1, x), & y = 0 \\ \phi^3(i_2, x, y-1, \phi^2(i_3, x, y-1)), & y \neq 0 \end{cases}$$

A função parcial ψ^* é igual à função $l(x, y) = k(i_1, i_2, i_3, x, y)$. Se a é um índice para k tem-se

$$k(i_1, i_2, i_3, x, y) = \phi^5(a, i_1, i_2, i_3, x, y) = \phi^2(s_3^2(a, i_1, i_2, i_3), x, y)$$

Fixando $\alpha(i_1, i_2, i_3) = s_3^2(a, i_1, i_2, i_3)$ tem-se que α é uma função recursiva primitiva que calcula o índice de ψ^*

$$\left(\phi^2_{i_3}\right)^* = \phi^2_{\alpha(i_1, i_2, i_3)}$$

Por aplicação da terceira versão do teorema do ponto fixo temos que existe uma função recursiva primitiva $j \in \mathcal{F}_2$ tal que para todo o z e t

$$\phi^2_{\alpha(z,t,j(z,t))} = \phi^2_{j(z,t)}$$

o que mostra que

$$\left(\phi^e_{j(i_1,i_2)}\right)^* = \phi^2_{j(i_1,i_2)}$$

o que, pela unicidade acima referida, permite dizer que $\phi^2_{j(i_1,i_2)} = f$. □

Capítulo 4

Teoria da Complexidade

Nesta secção destes apontamentos o texto segue de perto o livro *Introduction to the Theory of Computation* de Michael Sipser [Sip97].

4.1 Medindo a Complexidade

Tradicionalmente as medidas de complexidade de algoritmos relacionam-se com o tempo e o espaço. Mede-se o tempo de CPU e/ou o espaço de memória RAM que um dado programa consome aquando da resolução de um dado problema.

Para o estudo teórico ir-se-á estudar linguagens, palavras, nomeadamente o seu comprimento, e máquinas de Turing capazes de processar as linguagens, decidindo se uma dada palavra pertence ou não a uma dada linguagem.

Exemplo 4.1 *Considere-se a linguagem* $A = \{0^k 1^k \mid k \geqslant 0\}$*. Qual é o tempo que uma dada máquina de Turing precisa para decidir se uma palavra de comprimento* n *pertence, ou não, à linguagem* A*?*

Seja M_1 *a seguinte máquina de Turing com uma só fita.*

Para uma dada entrada $\omega \in A$*:*

1. *lê toda a fita e* **rejeita** *se encontra um* 0 *à direita de um* 1*;*
2. *repete enquanto existem simultaneamente 0s e 1s:*
    (a) *«corta» todos os* 0 *que tenham (imediatamente) à sua esquerda um* 1*;*
3. *se ainda existem 0s após todos os 1s terem sido «cortados», ou se ainda existem 1s após todos os 0s terem sido «cortados»* **rejeita***. Caso contrário* **aceita***.*

Para podermos estudar a complexidade temporal deste algoritmo é necessário contar o número de passos que a cabeça da máquina necessita para decidir sobre uma sequência de comprimento n. Isto é a unidade de tempo será *um movimento da cabeça* da máquina de Turing

Pretende-se:

Análise do Pior Caso pretende-se saber qual é o tempo gasto para o pior caso, isto é, dada um comprimento n qual é o tempo máximo que a máquina necessita para decidir se a palavra pertence ou não à linguagem.

Análise do Caso Médio pretende-se saber qual é o tempo médio gasto para uma dada palavra de comprimento n.

Definição 4.2 (Complexidade Temporal) *Seja M uma máquina de Turing determinística que pára para todos os valores de entrada. O* tempo de execução *ou* complexidade temporal *de M é uma função $f : \mathbb{N} \to \mathbb{N}$, aonde $f(n)$ é o número máximo de passos que M utiliza, para uma qualquer entrada de comprimento n. Se $f(n)$ é o tempo de execução de M, diz-se que M executa em tempo $f(n)$, e que M é uma máquina de tempo $f(n)$.*

4.1.1 Notação $\mathcal{O}$-maiúsculo e o-minúsculo

Dado que se pretende obter uma estimativa (mais do que valor exacto) do tempo é conveniente usar uma análise assimptótica, isto é, pretende-se obter um valor que se irá aproximar do valor exacto à medida que o comprimento da sequência de entrada cresce.

- Pretende-se compreender o comportamento do algoritmo quando confrontado com entradas «grandes».
- Da expressão da complexidade temporal vai-se considerar somente os termos de ordem superior, desprezando todos os outros.

Exemplo, para um dada máquina de Turing M, obteve-se a seguinte expressão para a complexidades temporal: $f(n) = 6n^3+2n^2+20n+45$, temos então que o valor que pretendemos é $f(n) = 6n^3$, desprezando todos os termos de ordem inferior. Mais correctamente, vai-se considerar $f(n) = n^3$ dado que para valores de n suficientemente grandes também o facto 6 é desprezável.

Dir-se-ia então que a complexidade (temporal) assimptótica de M é $f(n) = \mathcal{O}(n^3)$.

Definição 4.3 (Notação $\mathcal{O}$-maiúsculo) *Sejam f e g duas funções $f, g : \mathbb{N} \to \mathbb{R}^+$. Diz-se que $f(n) = \mathcal{O}(g(n))$ se existem dois naturais não nulos c e n_0 tais que para todo o $n \geqslant n_0$ tem-se:*

$$f(n) \leqslant cg(n).$$

Quando $f(n) = \mathcal{O}(g(n))$, diz-se que $g(n)$ é um limite superior assimptótico para $f(n)$.

Intuitivamente $f(n) = \mathcal{O}(g(n))$ significa que f é menor ou igual a g a menos de um factor constante.

Exemplo 4.4 *Seja $f_1(n) = 5n^3 + 2n^2 + 22n + 6$. Então seleccionado o seu termo de ordem superior e desprezando o seu coeficiente, tem-se que $f_1(n) = \mathcal{O}(n^3)$.*

Vejamos que assim é. Façamos $c = 6$ e $n_0 = 10$, é fácil de verificar que então, para todo o $n \geqslant 10$ tem-se que $5n^3 + 2n^2 + 22n + 6 \leqslant 6n^3$.

Podemos ainda dizer que $f(n) = \mathcal{O}(n^k)$ para todo o $k \geqslant 4$. Para todos esses casos ter-se-ia $n^k \geqslant n^3$ e como tal as condições da definição verificar-se-iam.

Já não se pode dizer o mesmo da função n^2, f_1 não é $\mathcal{O}(n^2)$ isto dado que para quaisquer valores de c e n_0 que se considerem nunca se estaria nas condições da definição.

No caso de se considerarem funções logarítmicas a definição para a notação $\mathcal{O}$ leva a que não seja necessário ter em conta qual é a base do logaritmo. Vejamos que assim é.

No caso das funções logaritmos é verdadeira a seguinte igualdade que nos dá a conversão de bases:

$$\log_b n = \frac{\log_2 n}{\log_2 b} = \frac{1}{\log_2 b} \log_2 n$$

Dado que $\frac{1}{\log_2 b}$ é um factor constante e dado que na notação $\mathcal{O}$ se ignoram os factores constantes, podemos dizer algo como $f(n) = \mathcal{O}(\log n)$ sem que se diga nada sobre a base do logaritmo.

Por exemplo, dado $f_2(n) = 3n \log_2 n + 5n \log_2 \log_2 n + 2$, tem-se que $f_2(n) = \mathcal{O}(n \log n)$, isto dado que $\log n$ domina $\log \log n$.

Quando o símbolo $\mathcal{O}$ ocorre numa exponencial, por exemplo $f(n) = 2^{\mathcal{O}(n)}$ isso representa um limite superior para 2^{cn} para um dado c.

No caso de termos a expressão $f(n) = 2^{\mathcal{O}(\log n)}$ isso representa um limite superior para n^c para uma qualquer c. Podemos ver que assim é dado que $n = 2^{\log_2 n}$ implica $n^c = 2^{c \log_2 n}$.

Por fim a expressão $f(n) = \mathcal{O}(1)$ representa um valor que nunca é superior a uma qualquer constante c.

Fala-se de limites polinomiais quando se tem uma expressão do tipo $\mathcal{O}(n^i)$ e de limites exponenciais no caso de se ter $\mathcal{O}(2^{n^i})$, para um dado i natural positivo.

Há notação $\mathcal{O}$-maiúsculo junta-se a notação o-minúsculo. Se no primeiro caso dizemos que uma função não é assimptoticamente maior do que uma outra, no caso da segunda dir-se-á que uma função é assimptoticamente menor do que uma outra. As diferenças entre as notações $\mathcal{O}$-maiúsculo e o-minúsculo são semelhantes às diferenças entre $\leqslant$ e $<$.

Definição 4.5 (Notação o-minúsculo) *Sejam f e g duas funções $f, g : \mathbb{N} \to \mathbb{R}^+$. Diz-se que $f(n) = o(g(n))$ se*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Ou, de outra forma, $f(n) = o(g(n))$ significa que para um qualquer número real $c > 0$ existe um natural n_0 de tal forma que $f(n) < cg(n)$ para todo o $n \geqslant n_0$.

Exemplo 4.6 *É fácil de verificar que:*

- $\sqrt{n} = o(n)$.
- $n \log n = o(n^2)$.

No entanto $f(n)$ nunca é igual a $o(f(n))$.

Vejamos então qual é a complexidade temporal da máquina de Turing M_1 do exemplo 4.1.

1. percorre a fita a verificar se a entrada contém uma sub-sequência do tipo `10`: máximo n passos. Posicionar a cabeça de volta ao início: máximo n passos. No total temos $2n$ no máximo.

2. percorre a fita repetidas vezes eliminando a cada passagem pares do tipo `01`: $\frac{n}{2}$ vezes n passos. No máximo $\frac{1}{2}n^2$ passos.

3. um só percorrer da fita para decidir se aceita ou rejeita. No máximo n passos.

Temos então que $M_1(n) = 2n + \frac{1}{2}n^2 + n = \frac{1}{2}n^2 + 3n = \mathcal{O}(n^2)$

Definição 4.7 (Classes de Complexidade Temporal) *Seja $t : \mathbb{N} \to \mathbb{N}$ uma função. Define-se a classe de complexidade temporal* $\textsc{time}(t(n))$ *como sendo o seguinte conjunto*

$$\textsc{time}(t(n)) = \{L \mid L \textit{ é a linguagem decidida por uma maq. de Turing em tempo } \mathcal{O}(t(n))\}$$

Relembrando o exemplo 4.1, para este caso podemos afirmar que $A \in \textsc{time}(n^2)$ dado que M_1 decide A em tempo $\mathcal{O}(n^2)$ e $\textsc{time}(n^2)$ contêm todas as linguagens que podem ser decididas em tempo $\mathcal{O}(n^2)$.

No entanto isso não fecha o problema da classificação de A. Podemos colocar a questão. Existe uma outra máquina de Turing capaz de decidir A em tempo menor do que $\mathcal{O}(n^2)$?

Vejamos uma outra máquina de Turing para o problema de decidir A.

Exemplo 4.8 *Considere-se a linguagem $A = \{0^k1^k \mid k \geqslant 0\}$. Seja M_2 a seguinte máquina de Turing com uma só fita.*

Para uma dada entrada $\omega \in A$:

1. *lê toda a fita e* **rejeita** *se encontra um* 0 *à direita de um* 1*;*

2. *repete enquanto existem simultaneamente 0s e 1s:*

   (a) *percorre a fita verificando se o número total de 0s e 1s que restam na fita é par ou ímpar. Se é ímpar* **rejeita***;*

   (b) *percorre de novo a fita «apagando» os 0s alternadamente, começando pelo primeiro* 0*. Depois «apaga» os 1s alternadamente começando pelo primeiro 1.*

3. *se ainda existem 0s após todos os 1s terem sido «apagados», ou se ainda existem 1s após todos os 0s terem sido «apagados»* **rejeita***. Caso contrário* **aceita***.*

A exemplo do que já se fez acima vamos analisar esta máquina passo a passo.

1. no máximo $2n$ (ver exemplo 4.1);

2. a cada passagem do ciclo vão-se «apagar» metade dos 0s e 1s, sendo assim temos que no máximo este passo leva $2n(1 + \log_2 n)$.

3. no máximo n (ver exemplo 4.1);

Em conlusão $M_2(n) = 2n + 2n + 2n\log_2 n + n = \mathcal{O}(n \log n)$.

Ou seja obteve-se um melhor limite para a complexidade temporal de A, isto é acabou-se de verificar que $A \in \textsc{time}(n \log n)$.

Vejamos ainda uma outra máquina de Turing para A.

Exemplo 4.9 *Considere-se a linguagem $A = \{0^k1^k \mid k \geqslant 0\}$. Seja M_3 a seguinte máquina de Turing com duas fitas.*

Para uma dada entrada $\omega \in A$ carregada na fita 1:

1. *percorre a fita 1 e* **rejeita** *se encontra um* 0 *à direita de um* 1*;*

2. *percorre a fita 1 até ao primeiro 1, no processo copia os 0s para a fita 2.*

3. *percorre a fita 1 até ao fim da entrada. Para cada 1 lido na fita 1 «apaga» um 0 na fita 2. Se todos os 0s já foram «apagados» antes do último 1* **rejeita**.

4. *se ainda existem 0s na fita 2* **rejeita**. *Caso contrário* **aceita**.

Todos os passos têm uma complexidade $\mathcal{O}(n)$, temos então que a complexidade total é $M_3(n) = \mathcal{O}(n)$.

Os exemplos anteriores poêm em evidência o facto de que diferentes modelos de computação, por exemplo máquina de Turing de 1 fita versus máquinas de Turing com 2 fitas, levam a que um mesmo problema possa pertencer a diferentes classes de complexidade.

4.1.2 Relações de Complexidade entre Modelos Diferentes

Na teoria da complexidade queremos classificar os problemas de acordo com o tempo necessário à sua resolução, no entanto, como vimos acima, o modelo de computação escolhido interfere com a medida de tempo. O mesmo problema pode ter medidas de complexidade diferentes de acordo com diferentes modelos de computação.

Felizmente, se o sistema de classificação que se pretende estabelecer tem uma granularidade não muito fina vai-se chegar à conclusão que as diferenças entre diferentes modelos determinísticos não é significativa.

Vamos de seguida tentar estabelecer de que forma diferentes modelos computacionais podem alterar a complexidade de um problema. Vamos considerar três modelos diferentes: máquinas de Turing com uma só fita; máquinas de Turing com múltiplas fitas e máquinas de Turing não determinísticas

Teorema 4.10 *Seja $t(n)$ uma função com $t(n) \geqslant n$. Então toda a máquina de Turing multi-fitas de complexidade temporal $\mathcal{O}(t(n))$ tem uma máquina de Turing com uma só fita equivalente e com complexidade temporal $\mathcal{O}(t^2(n))$.*

A demonstração deste resultado encontra-se em [Sip97]. A demonstração passa pela contabilizar do «peso» da conversão de uma máquina multi-fitas numa máquina de uma só fita. A simulação de cada passo de uma máquina multi-fitas numa de uma só fita vai consumir no máximo $\mathcal{O}(t(n))$, temos então que para uma complexidade da máquina multi-fitas de $\mathcal{O}(t(n))$ obter-se-á a complexidade $\mathcal{O}(t^2(n))$ para a máquina mono-fita.

Definição 4.11 *Seja N uma máquina de Turing não determinística*[1]. *O tempo de execução de N é uma função $f : \mathbb{N} \rightarrow \mathbb{N}$, aonde $f(n)$ é número máximo de passos de N para todos os ramos da sua computação para um qualquer valor de entrada.*

Graficamente podemos representar a diferença entre o tempo de execução de uma máquina de Turing determinística e uma não determinística (ver Figura 4.1), pela diferença entre uma estrutura linear e uma outra em forma de árvore.

[1] Uma máquina de Turing não determinista «funciona» da mesma forma que uma máquina de Turing determinista com a única diferença que a «funcão» de transição M não é uma função, mas uma *relação*: $M \subseteq (E \times F) \times (E \times F \times \{-1, 0, 1\})$. Isto significa que para qualquer estado da máquina temos, em geral, mais do que uma possibilidade de transição para o próximo passo.

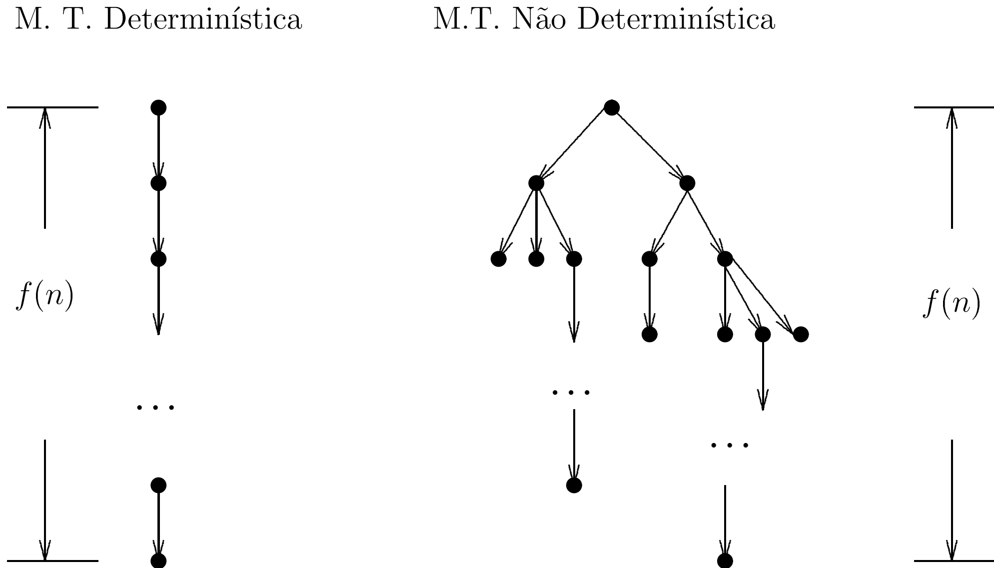


Figura 4.1: Determinístico vs Não Determinístico

Teorema 4.12 *Seja $t(n)$ uma função, aonde $t(n) \geqslant n$. Então toda a máquina de Tuting mono-fita não determinística com tempo de execução máximo de $t(n)$) tem uma equivalente máquina de Turing mono-fita determinística com tempo de execução máximo de $2^{\mathcal{O}(t(n))}$.*

4.2 A Classe P

No contexto do estudo da complexidade temporal de algoritmos as diferenças polinomiais no tempo de execução não são muito importantes, já diferenças exponenciais são muito significativas. Isto é podemos dizer que todos os problemas com complexidade são semelhantes, as diferenças ocorrem entre problemas, um com complexidade polinomial e outro com complexidade exponencial (ver Figura 4.2).

Os algoritmos com uma complexidade exponencial ocorrem muitas das vezes quando se tenta resolver um problema através de uma procura exaustiva no espaço das soluções, estas aproximações, muitas das vezes designadas por *força-bruta* ocorrem, por exemplo, quando se pretende factorizar um número em factores primos, testando todas as soluções possíveis. Estas aproximações *força-bruta*, sendo sempre possíveis, têm de ser vistas como soluções de recurso dado que rapidamente a sua aplicação se torna impraticável, por exemplo, em criptografia pretende-se sempre que uma cifra tenha a característica de ser inquebrável, a menos de uma solução do tipo *força-bruta*. Se isso se verificar basta confirmar que o espaço das soluções (chaves) é suficientemente grande para garantir que a aproximação por *força-bruta* é impraticável (ver Figura 4.2).

Tendo o que foi dito acima em conta vai-se considerar que todos os modelos determinísticos razoáveis são polinomialmente equivalentes. Isto é, pode-se sempre simular um modelo num outro com, no máximo, uma diferença polinomial no tempo de execução. Não se vai tentar definir o que se entende por modelos razoáveis, máquinas de Turing, cálculo-λ, etc.

Note-se que o facto de, em termos teóricos, não se considerarem diferenças polinomiais entre algoritmos não quer dizer que essas diferenças, do ponto de vista prático, não sejam

importantes. Para um dado problema a diferença entre um algoritmo com complexidade $\mathcal{O}(n)$ e um outro com complexidade $\mathcal{O}(n^3)$ é muito importante. Em muitas áreas de investigação tenta-se arduamente melhorar os algoritmos, mesmo que por factores constantes, para desse modo ir avançando na dimensão dos problemas que se conseguem resolver.

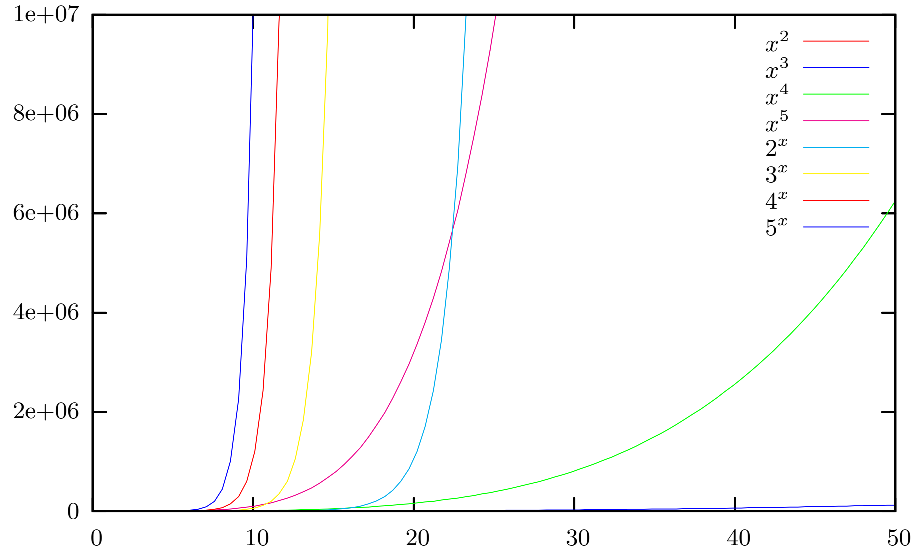


Figura 4.2: Polinomial vs Exponencial

Do ponto de vista teóricos esta escolha de uma maior granularidade prende-se com a necessidade de estudar o problema da complexidade temporal de uma forma global, ignorando portanto as diferenças locais.

Vejamos então uma primeira definição muito importante neste nosso estudo.

Definição 4.13 (P) *P é a classe das linguagens que são decidíveis em tempo polinomial numa máquina de Turing determinística e mono-fita. Isto é:*

$$P = \bigcup_k \text{TIME}(n^k)$$

A classe P tem um papel muito importante na teoria da complexidade dado que:

- P é invariante para todos os modelos de computabilidade que são polinomialmente equivalentes a uma máquina de Turing determinística mono-fita;
- P corresponde à classe de problemas realisticamente resolúveis por um computador.

4.2.1 Exemplos de Problemas em P

Vejamos alguns exemplos de problemas em P.

Determinar a existência de um caminho entre dois nós distintos de um grafo (ver Figura 4.3).

$$\text{Caminho} = \{(G, s, t) \mid G \text{ é um grafo dirigido que tem um caminho entre } s \text{ e } t\}$$

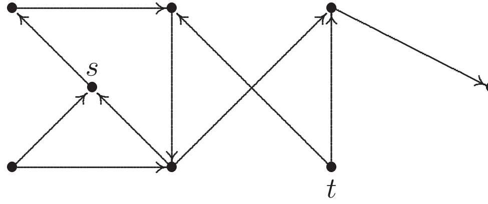


Figura 4.3: Caminho entre dois nós

Determinar se dois inteiros são primos relativos.

$$\text{PrimosRelativos} = \{(x, y) \mid x \text{ e } y \text{ são primos relativos}\}$$

Caminho e PrimosRelativos pertencem a P.

4.3 A classe NP

O algoritmo *força bruta* é sempre possível de aplicar, é no entanto em muitos casos, e do ponto de vista prático, não utilizável dado conduzir a soluções com uma complexidade temporal exponencial.

Por exemplo, para os dois casos anteriores ter-se-ia que o algoritmo *força bruta* levava a uma solução com complexidade temporal m^m, com m o número de nós do grafo, para Caminho e 2^n, com 2 a base da representação numérica usada e n o número a factorizar em PrimosRelativos.

Nos casos anteriores foi possível encontrar alternativas com uma complexidade temporal polinomial. Será isso sempre possível?

Vejamos um outro exemplo, a determinação de um caminho Hamiltoniano num dado grafo dirigido (ver Figura 4.4), isto é: dado dois nós s e t de um grafo G achar um caminho entre eles em que não se passe pelo mesmo nó mais do que uma vez.

$$\text{CaminhoHam} = \{(G, s, t) | G \text{ é um grafo dirigido com um caminho Hamiltoniano entre } s \text{ e } t\}$$

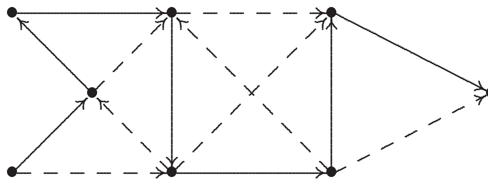


Figura 4.4: Caminho Hamiltoniano

A exemplo do problema Caminho a aproximação *força bruta* dá-nos um algoritmo com complexidade temporal exponencial, ao contrário do problema anterior, não se conhece nenhuma alternativa polinomial.

No entanto o problema CaminhoHam tem uma propriedade, que se vai designar por verificabilidade polinomial, que se vai revelar importante para perceber a sua complexidade.

Definição 4.14 (Verificador Polinomial) *Um* verificador *para uma linguagem A é um algoritmo V, aonde*

$$A = \{w \mid V \text{ aceita } (w, c) \text{ para uma dada sequência de caracteres } c\}.$$

Para um verificador o tempo é medido em termos do comprimento do w. Um verificador polinomial no tempo *corre em tempo polinomial no comprimento de w.*

Uma linguagem A é polinomialmente verificável se possuí um verificador polinomial. Na definição de verificador polinomial o c é dito um certificado (prova) de pertença a A.

- Para CaminhoHam o certificado é um dado caminho Hamiltoniano.
- Para o problema de factorização de um inteiro nos seus factores primos, um certificado é um dos divisores.

Definição 4.15 (NP) *NP é a classe das linguagens que possuem verificadores polinomiais.*

A designação NP advêm do facto de se poder caracterizar esta classe através de máquinas de Turing não-determinísticas com complexidade temporal polinomial.

Teorema 4.16 *Uma linguagem está em NP sse é decidível através de uma máquina de Turing não determinística em tempo polinomial.*

Definição 4.17 (NTIME) *Seja $t : \mathbb{N} \to \mathbb{N}$ uma função. Define-se a classe de complexidade temporal $\text{NTIME}(t(n))$ como sendo o seguinte conjunto:*

$$\text{NTIME}(t(n)) = \left\{ L \mid \begin{array}{l} L \text{ é a linguagem decidida por uma máquina de Turing não deter-} \\ \text{minística em tempo } \mathcal{O}(t(n)) \end{array} \right\}$$

Corolário 4.18

$$NP = \bigcup_k \text{NTIME}(n^k).$$

A classe NP é insensível à escolha de um modelo computacional não determinístico razoável, dado que todos esses modelos serão polinomialmente equivalentes.

4.3.1 Exemplos de Problemas em NP

Um «clique» num dado grafo não dirigido é um sub-grafo completo, isto é um sub-grafo para o qual todos os pares de nós estão ligados por um arco. Um «clique-k» é um clique que contém k nós.

O problema CLIQUE é o problema de determinar se um dado grafo contém, ou não, um clique da dimensão especificada.

$$\text{CLIQUE} = \{(G, k) \mid G \text{ é um grafo não dirigído com um clique-}k\}.$$

Teorema 4.19 CLIQUE *está em NP.*

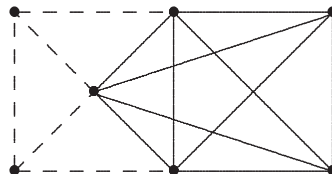


Figura 4.5: Um Grafo com um clique-5

Um outro exemplo é dado pelo problema da soma dos subconjuntos SOMASUBCONJ. Este problema pode ser caracterizado do seguinte modo: tem-se uma dada colecção de números inteiros $A = \{x_1, \ldots, x_k\}$ e um dado número inteiro t, designado por objectivo. Pretende-se determinar se a colecção acima dada contém uma sub-colecção cuja soma dos seus elementos dê o objectivo.

$$\begin{array}{rcl} \textsc{somaSubConj} & = & \{(S,t) \mid S = \{x_1, \ldots, x_k\} \text{ e para algum} \\ & & \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ tem-se } \sum y_i = t\} \end{array}$$

Note-se que na definição dada acima estão-se a considerar sacos, isto é o conjuntos que admitem elementos repetidos, e não conjuntos.

Por exemplo $(\{4, 11, 16, 21, 27\}, 25) \in$ SOMASUBCONJ dado que se tem $4 + 21 = 25$.

Teorema 4.20 SOMASUBCONJ *está em NP.*

4.3.2 P versus NP

Como se disse acima NP é a classe das linguagens que são resolúveis em tempo polinomial por uma máquina de Turing não-determinística, ou de forma equivalente, é a classe das linguagens para as quais a pertença à linguagem pode ser verificada em tempo polinomial. Por outro lado P é a classe das linguagens aonde a pertença pode ser testada em tempo polinomial.

De forma informal, podemos resumir esta informação dizendo que:

$$\begin{array}{rcl} P & = & \text{a classe das linguagens cuja pertença pode ser decidida de forma expedita} \\ NP & = & \text{a classe das linguagens cuja pertença pode ser verificada de forma expedita} \end{array}$$

Os problemas CAMINHOHAM e CLIQUE são membros de NP mas não se sabe se estão em P. A questão de saber se existe alguma linguagem em NP mas não em P é uma das grandes questões ainda em aberto na teoria da computação.

A questão de saber se $NP \subsetneq P$ ou $NP = P$ resiste ainda a todas as tentativas de demonstração.

Acredita-se que $NP \subsetneq P$ e é confiando nesse crença que os sistemas criptográficos de chave pública actuais se baseiam. Problemas cuja resolução é tida como muito difícil, e cuja verificação através de um certificado é fácil.

4.4 Completude NP

Um importante desenvolvimento na questão da relação entra as classes de complexidade temporal P e NP foi feito nos inícios da década de 70 por Stephen Cook e Leonid Levin [Coo71, Tra84]. De forma independentes eles descobriram que existem problemas cuja complexidade está relacionada com a complexidade de toda uma classe de complexidade. Caso exista um algoritmo com complexidade temporal polinomial para um desses problemas, então todos os problemas na classe NP seriam resolúveis em tempo polinomial. Estes problemas são designados por problemas **NP-completos.**

Este tipo de problemas é importante tanto do ponto de vista teórico como do ponto de vista prático. Do ponto de vista teórico a sua importância advêm do facto de permitir a investigação da relação entre as classes P e NP a este tipo concreto de problemas. Do ponto de vista prático as implicações que a descoberta de uma solução polinomial para um problema NP-completo, seriam enormes. Todos os problemas agora tidos como intractáveis passariam a ter uma solução polinomial, a questão da encontrar passaria a estar na ordem do dia.

Vejamos um primeiro problema nesta classe, o problema SAT, de *satisfiability problem* (problema da satisfação). O problema SAT pode ser caracterizado como o problema de saber se uma dada fórmula lógica é verdadeira para um dado conjunto de valores das suas variáveis lógicas. Uma fórmula lógica conterá somente variáveis lógicas (capazes de tomar o valor de *Verdade* ou *Falso*) ligadas pelos operadores lógicos de negação, conjunção e disjunção.

É sempre possível encontrar esse valor, por exemplo para a fórmula $\phi = (\neg x \wedge y) \vee (x \wedge \neg z)$, basta atribuir todos os valores possíveis (*Verdade*, *Falso*) às suas variáveis e verificar o valor obtido. No caso do exemplo anterior os valores $x = Falso$, $y = Verdade$, $z = Falso$, satisfaz a fórmula ϕ dando-lhe o valor lógico de *Verdade*. Temos então:

$$\text{SAT} = \{\phi \mid \phi \text{ é satisfeita para uma dada atribuição de valores às suas variáveis lógicas}\}$$

Podemos agora formular o teorema que relaciona o problema SAT com a classe de complexidade SAT.

Teorema 4.21 (Teorema Cook-Levin) *SAT $\in P$ sse $P = NP$.*

Bibliografia

[CL01] René Cori and Daniel Lascar. *Mathematical Logic, Part II*. Oxford University Press, 2001.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[Tra84] B.A. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *Annals of the History of Computing*, 6(4):384 –400, oct.-dec. 1984.