O problema P versus NP

Fernando Ferreira
Departamento de Matemática da FCUL

1 Abertura

Por volta de 300 a.C., Euclides descreveu um algoritmo de recorrência para calcular o máximo divisor comum entre dois números inteiros positivos. Ei-lo:

EUCLIDES(a, b)
1 **se** $b = 0$
2 **então o resultado é** a
3 **caso contrário o resultado é** EUCLIDES$(b, a \bmod b)$

Como exemplo do processo computacional descrito pelo algoritmo EUCLIDES, consideremos a computação do máximo divisor comum entre os números 30 e 21:

$$\begin{aligned} \text{EUCLIDES}(30,21) &= \text{EUCLIDES}(21,9) \\ &= \text{EUCLIDES}(9,3) \\ &= \text{EUCLIDES}(3,0) = 3 \end{aligned}$$

O algoritmo EUCLIDES descreve uma sequência de passos computacionais que, a partir das entradas a e b, produz um determinado resultado: a saber, o máximo divisor comum entre a e b. Não descrevemos exaustivamente todos os passos computacionais que subjazem à execução do algoritmo. Por exemplo, não descrevemos os passos computacionais que permitem calcular $a \bmod b$, i.e., o resto da divisão inteira de a por b. Supusemos que esta computação não é problemática e que é por demais conhecida dos leitores — de facto, conhecida desde os primeiros tempos de escola.

Palestra proferida no dia 19 de Outubro de 2000, integrada no ciclo "2000 matemática $\sqrt{\text{radical}}$" do Departamento de Matemática da Faculdade de Ciências da Universidade de Lisboa.

Cada ciclo do algoritmo EUCLIDES calcula um resto. Estes restos vão diminuindo sucessivamente, uma vez que o resto duma divisão é sempre inferior ao divisor. Assim, mais tarde ou mais cedo, o resto anula-se, entrando o algoritmo na cláusula de escape 2 e terminando com o resultado final. O facto de o resultado estar correcto, i.e., de ser realmente o máximo divisor comum (mdc) dos dois números dados, advém da seguinte igualdade: $\mathrm{mdc}(a, b) = \mathrm{mdc}(b, a \bmod b)$.

Se quisermos saber se um número inteiro a, com $a > 1$, é um número composto (i.e., não primo), podemos procurar exaustivamente por um divisor de a, desde 2 até à sua raiz quadrada:

```
EXAUSTÃO(a)
 1  k ← 2
 2  enquanto k² ⩽ a
 3      se a mod k = 0
 4          então o resultado é SIM
 5          caso contrário k ← k + 1
 6  o resultado é NÃO
```

O algoritmo EXAUSTÃO é iterativo e, dado a, a iteração pode ocorrer um número de vezes da ordem de $\sqrt{a} - 1$. O número de ciclos de EUCLIDES é muito menor: dados a e b, EUCLIDES tem, no máximo, $2(1 + \log_2 a)$ ciclos. A razão para este facto não é difícil de explicar. Basta observar que a cada dois ciclos de EUCLIDES o resto diminui para, pelo menos, metade. Com efeito, se $a \bmod b$ já fosse, quanto muito, metade de b então, num só ciclo, o resto já se teria reduzido o suficiente. Caso contrário, o resto seguinte seria $b \bmod r$, com $r = a \bmod b > \frac{b}{2}$. Logo, o resto seria $b - r < \frac{b}{2}$.

Os dois algoritmos acima dizem respeito a números inteiros positivos. Do ponto de vista computacional, estes números são dados em notação posicional binária, i.e., como sequências apropriadas de 0s e 1s. O *comprimento* $|a|$ dum número inteiro positivo a é o número de *bits* que entra na sua representação binária. Não é difícil ver que esse número difere de $\log_2 a$ por, no máximo, uma unidade. Se n for o máximo dos comprimentos das entradas a e b, então o número de ciclos computacionais por que passa a execução de EUCLIDES está limitado pelo valor $2(n + 2)$: na gíria matemática, é $\mathcal{O}(n)$. Por sua vez, em cada ciclo, EUCLIDES efectua uma divisão inteira. Ora, o algoritmo usual da divisão inteira é quadrático, i.e., efectua-se em $\mathcal{O}(n^2)$ passos, onde n é o máximo dos comprimentos do dividendo e do divi-

sor. Em suma, EUCLIDES trabalha em $\mathcal{O}(n^3)$ passos. EUCLIDES é um algoritmo em tempo cúbico e, portanto, em tempo polinomial. Por outro lado, para uma entrada a, o algoritmo EXAUSTÃO pode ter que sofrer $\sqrt{a}-1$ iterações. Se n for o comprimento de a, estas iterações corresponderiam a cerca de $2^{\frac{n}{2}}$ tarefas (e, portanto, a pelo menos tantos passos computacionais). EXAUSTÃO não é um algoritmo que trabalhe em tempo polinomial.

Um algoritmo como EXAUSTÃO não é eficiente do ponto de vista prático. Um computador que trabalhe ininterruptamente a 1GHz durante um ano efectua cerca de $3,2 \times 10^{16}$ passos computacionais. Por sua vez, para $n = 128$, $2^{\frac{n}{2}}$ é, aproximadamente, $1,8 \times 10^{19}$. Por conseguinte, um computador a 1GHz demoraria cerca de 563 anos a executar 2^{64} passos computacionais. Para $n = 256$, $2^{\frac{n}{2}}$ é cerca de $3,4 \times 10^{38}$. O nosso sistema solar tem, aproximadamente, 6×10^9 anos. Consequentemente, todo esse tempo está ainda muitíssimo longe de chegar para executar 2^{128} passos a 1GHz!

2 O enunciado do problema

Seja Σ um alfabeto finito (i.e., um conjunto finito e não vazio) e seja Σ^* o conjunto de todas as sequências finitas de elementos de Σ (as *palavras* de Σ). Um *problema* é um subconjunto de Σ^*. Por exemplo, o conjunto de todos as palavras do alfabeto $\{0,1\}$ que representam — em notação posicional binária — um número composto forma um problema: o problema COMPOSTO. Outro exemplo é o problema de saber se o máximo divisor comum entre dois números inteiros positivos é 1. Podemos formular este problema no alfabeto $\{0,1,\#\}$ como sendo o conjunto constituído pelas palavras da forma $x\#y$, onde x e y são palavras do alfabeto $\{0,1\}$ que representam números cujo máximo divisor comum é 1. Chamamos a este problema PRIMOS-ENTRE-SI.

Um problema L num alfabeto Σ diz-se *decidível em tempo polinomial* se existir um algoritmo que trabalhe em tempo polinomial e que termine em SIM sempre que a entrada x estiver em L, e que termine em NÃO caso contrário. A classe P é a classe dos problemas decidíveis em tempo polinomial. Não definimos matematicamente nem a noção de algoritmo nem a noção de passo computacional. Por conseguinte, a caracterização que acabámos de descrever é meramente

indicativa. A definição matemática, rigorosa, da noção de algoritmo apareceu nos anos trinta do século passado, fruto de investigações em lógica matemática (por, entre outros, Jacques Herbrand, Kurt Gödel, Alonzo Church, Emil Post, Stephen Kleene e Alan Turing). A definição de algoritmo devida a Turing não toma muito espaço. Fornecemo-la no anexo a este artigo, tornando assim completa e rigorosa a formulação do problema P versus NP aqui apresentada.

Um problema L num alfabeto Σ diz-se decidível *não deterministicamente* em tempo polinomial se houver uma relação binária R em Σ^*, decidível em tempo polinomial, e houver um polinómio $p(X) \in \mathbb{N}[X]$ tal que, para todas as palavras $x \in \Sigma^*$,

$$x \in L \Leftrightarrow \exists y\,(|y| \leqslant p(|x|)\,\&\,xRy), \qquad (*)$$

onde $|x|$ denota o comprimento da sequência (palavra) x. A classe NP é a classe dos problemas decidíveis em tempo polinomial não determinista.

Enunciado do problema: Será que P = NP?

É claro que COMPOSTO $\in$ NP, pois um número inteiro positivo a é composto se, e somente se,

$$\exists b\,(|b| \leqslant |a|\,\&\,1 < b < a\,\&\,a \bmod b = 0).$$

Neste exemplo, a relação binária pertinente (dita *relação de verificação*) é a relação $aRb \Leftrightarrow 1 < b < a\,\&\,a \bmod b = 0$. Note o leitor que se trata duma relação decidível em tempo polinomial. Todo o problema em P (e.g., o problema PRIMOS-ENTRE-SI) está em NP por razões triviais: se $L \in$ P então $L \in$ NP através da relação de verificação $xRy \Leftrightarrow y = y\,\&\,x \in L$ e tomando para $p(X)$ o polinómio de valor constante 1. O problema P versus NP é o problema de saber se NP $\subseteq$ P.

Dizemos que um problema L está na classe EXP se puder ser decidido por um algoritmo que trabalhe em tempo $\mathcal{O}(2^{p(n)})$, para um certo polinómio $p(X) \in \mathbb{N}[X]$. Dado $L \in$ NP como em $(*)$, uma busca exaustiva mostra que $L \in$ EXP:

FORÇABRUTA(x)
1 $y \leftarrow \epsilon$
2 **enquanto** $|y| \leqslant p(|x|)$
3 **se** xRy
4 **então o resultado é** SIM
5 **caso contrário** $y \leftarrow s(y)$
6 **o resultado é** NÃO

Aqui, ϵ denota a palavra vazia e $s(y)$ denota a palavra que se segue à palavra y na seguinte ordenação total de Σ^*: supondo que os elementos de Σ estão previamente ordenados, diz-se que uma palavra $x \in \Sigma^*$ precede outra palavra $y \in \Sigma^*$ se o comprimento da primeira for inferior ao comprimento da segunda ou se, tendo ambas o mesmo comprimento, então x precede y lexicograficamente. Seja m, com $m > 1$, o número de elementos do alfabeto Σ. O número de palavras de Σ^* cujo comprimento não excede n é $\frac{m^{n+1}-1}{m-1}$, o que permite concluir que FORÇABRUTA trabalha em tempo $\mathcal{O}(2^{r(n)})$, para certo polinómio $r(X)$.

A um valor y que testemunhe a verdade do segundo membro de $(*)$ chama-se um *certificado sucinto* da afirmação $x \in L$. Dado $x \in L$, é uma tarefa fácil verificar se um dado candidato a certificado sucinto é, ou não, genuíno; a dificuldade está em encontrá-lo no meio de um potencial número exponencial de candidatos! É precisamente nesta procura que reside a dificuldade de decidir a generalidade dos problemas de NP.

Não se deve confundir a questão $\mathsf{P} = \mathsf{NP}$ com a questão $\mathsf{P} = \mathsf{EXP}$. Esta última questão tem uma resposta pronta: a teoria da complexidade computacional mostra facilmente que há problemas em EXP que não estão em P (através de um argumento de diagonalização: ver adiante). Há pouco, vimos que $\mathsf{NP} \subseteq \mathsf{EXP}$, mas não se sabe se $\mathsf{NP} = \mathsf{EXP}$. Os especialistas acreditam que não.

3 Completude NP

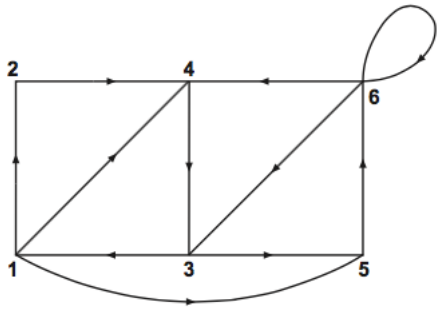
Sejam L_1 e L_2 dois problemas nos alfabetos Σ_1 e Σ_2, respectivamente. Diz-se que L_1 se *p-reduz* a L_2, e escreve-se $L_1 \leq_p L_2$, se existir uma função $f : \Sigma_1^* \mapsto \Sigma_2^*$, computável em tempo polinomial, tal que $x \in L_1 \Leftrightarrow f(x) \in L_2$, para todos os elementos x de Σ_1^*. Se $L_1 \leq_p L_2$ e

$L_2 \in \mathsf{P}$, conclui-se imediatamente que $L_1 \in \mathsf{P}$. Com efeito, decide-se se $x \in L_1$ calculando primeiro $f(x)$ e, em seguida, testando se $f(x) \in L_2$.

Um problema L diz-se NP-*completo* se $L \in \mathsf{NP}$ e se todo o problema em NP se p-reduz a L. Se os houver, os problemas NP-completos são os mais difíceis de entre os problemas em NP. Observe o leitor que se um problema NP-completo estiver em P, então $\mathsf{P} = \mathsf{NP}$.

Em 1971, Stephen Cook mostrou que existem problemas NP-completos. Eis um exemplo dum problema NP-completo.

Um grafo finito orientado G é um par (V, E), onde V é um conjunto finito e E é um subconjunto de $V \times V$. Devemos interpretar os elementos de V como sendo os vértices do grafo e cada par (x, y), tal que xEy, como sendo um arco que liga x a y neste sentido. Um *caminho Hamiltoniano* em G é uma permutação $x_1, x_2, \ldots, x_n$ de V tal que, para todo $1 \leqslant i < n$, se tem $x_i E x_{i+1}$. Por outras palavras, um caminho Hamiltoniano é um caminho através dos arcos do grafo que passa por cada um dos seus vértices uma, e uma única, vez. Por exemplo, o grafo



tem um caminho Hamiltoniano, e.g. $1 \to 2 \to 4 \to 3 \to 5 \to 6$. Chamamos HAMILTON ao problema de saber se um dado grafo tem, ou não, um caminho Hamiltoniano. Este problema é NP-completo.

Alguns leitores sentir-se-ão incomodados com a informalidade do exemplo. Qual é o alfabeto Σ que subjaz a HAMILTON? De que subconjunto L de Σ^* se trata exactamente? Este tipo de informalidade é muito usual em teoria da complexidade computacional (e, em geral, em matemática). Em ciência, a formalização é q.b. Quando se apresenta um problema deste modo informal, pressupõem-se que

as entradas do problema (neste caso, os grafos) estão representadas de alguma forma natural num determinado alfabeto. Por exemplo, um grafo orientado de n vértices pode representar-se através da sua matriz de adjacência $n \times n$, cuja entrada a_{ij} é 1 se houver um arco de i para j, e é 0 no caso contrário. A matriz de adjacência do grafo do exemplo acima é:

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

Uma matriz de 0s e 1s representa-se muito facilmente no alfabeto $\{0, 1, \#\}$, com o sinal # a desempenhar o papel de separador das linhas da matriz. Claro que há outras maneiras de representar naturalmente um grafo. Isso é indiferente no que concerne aos nossos estudos, desde que as diferentes representações sejam funções computáveis em tempo polinomial umas das outras. Por sua vez, os caminhos Hamiltonianos também se representam naturalmente no alfabeto $\{0, 1, \#\}$. O caminho do exemplo pode representar-se por

$$1\#11\#1111\#111\#11111\#111111.$$

Com esta representação, os caminhos Hamiltonianos dum grafo com n vértices são sempre palavras de comprimento $\sum_{i=1}^{n} i + (n-1) \leqslant n^2$. Logo, $x \in$ HAMILTON se, e somente se,

$$\exists y \, (|y| \leqslant |x|^2 \,\&\, \text{“}y \text{ é um caminho Hamiltoniano do grafo } x\text{”}),$$

pois o comprimento da representação de um grafo majora o número de vértices desse grafo. Tendo em conta que a relação binária “y é um caminho Hamiltoniano do grafo x” é decidível em tempo polinomial, concluímos que HAMILTON $\in$ NP. De facto, HAMILTON é um problema NP-completo. Qual é a razão para isso?

Historicamente, o primeiro exemplo de problema NP-completo foi o problema SATISFAÇÃO. Este problema consiste em decidir se uma fórmula do cálculo proposicional é satisfazível, isto é, se podemos dar valores de verdade às suas variáveis proposicionais de modo a que o valor da fórmula resulte verdadeiro. Stephen Cook mostrou

essencialmente o seguinte. Dado um problema em NP da forma $\exists y\,(|y| \leqslant p(|x|)\,\&\,xRy)$ e dada uma máquina de Turing M (*vide* anexo) que decida a relação de verificação R em tempo polinomial, há uma função computável em tempo polinomial que, dada a entrada x, computa uma fórmula proposicional F_x — a qual "descreve" a computação de M quando são introduzidas as entradas x e y, com (algumas) das variáveis proposicionais a representar a palavra desconhecida y — tal que F_x é satisfazível se, e somente se, para um certo y com $|y| \leqslant p(|x|)$, M decide afirmativamente que xRy. Foi com esta receita *universal* que Cook mostrou que SATISFAÇÃO é um problema NP-completo. Atendendo à transitividade da relação de p-redutibilidade, qualquer problema de NP em relação ao qual SATISFAÇÃO se p-reduza é um problema NP completo. Há, portanto, um método prático para demonstrar que um determinado problema de NP é NP-completo: basta mostrar que SATISFAÇÃO se p-reduz a esse problema.

Mais geralmente, um problema de NP em relação ao qual um determinado problema NP-completo se p-reduza, é NP-completo. É o que acontece com HAMILTON e com centenas de outros problemas que aparecem naturalmente em teoria dos grafos, combinatória, programação matemática, problemas de tabelamento, etc. Alguns destes problemas têm uma importância industrial tremenda. A existência de um algoritmo verdadeiramente eficiente que pudesse decidir SATISFAÇÃO teria consequências revolucionárias na indústria. Em contrapartida, como abordamos na próxima secção, destruiria a segurança das transacções financeiras na *internet* e através de outras vias electrónicas.

4 Criptografia

Entre os matemáticos, são bem conhecidas as opiniões *puristas* do grande matemático inglês Godfrey Hardy (1877-1947). Na sua autobiografia *A Mathematician's Apology* (Cambridge University Press, 1992), escreveu: "Nunca fiz nada de 'útil'. Nenhuma descoberta minha fez, nem é provável que faça, directa ou indirectamente, para o bem ou para ou mal, a mais pequena diferença para a aprazibilidade do mundo". Ironicamente, a área da matemática a que Hardy se dedicava — a teoria dos números — tem tido, desde os anos setenta,

importantíssimas aplicações à segurança das transacções comerciais e, mais geralmente, à confidencialidade das transmissões secretas.

Um sistema criptográfico K é um par (c_K, d_K), onde c_K contém a informação necessária para o emissor codificar mensagens (é a *chave de codificação*) e d_K contém a informação necessária para o receptor ler mensagens codificadas (é a *chave de descodificação*). Nos sistemas criptográficos clássicos, d_K obtém-se facilmente a partir de c_K. Para usar estes sistemas é condição *sine qua non* haver uma comunicação prévia e secreta entre emissor e receptor para concertar o par criptográfico. Sistemas como estes chamam-se sistemas de chave privada. Em 1978, Ronald Rivest, Adi Shamir e Leonard Adleman propuseram o primeiro sistema de troca de mensagens de chave pública, hoje conhecido por RSA. Neste sistema, o receptor fabrica previamente o par criptográfico (c_K, d_K), torna pública a chave de codificação c_K (em particular, torna-a conhecida do emissor) e guarda para si a chave d_K, pelo que não há necessidade de uma comunicação secreta entre emissor e receptor. A segurança deste sistema está na convicção de que é computacionalmente impraticável deduzir d_K a partir de c_K. O sistema RSA continua a ser um dos mais seguros conhecidos, sendo o mais utilizado para codificar mensagens através da *internet*.

A segurança do sistema RSA baseia-se no facto de que é muito fácil multiplicar números, enquanto que há a convicção de que é muito difícil factorizá-los (correntemente, é impraticável factorizar regularmente números com 200 casas decimais). Se $\mathsf{P} = \mathsf{NP}$, então factorizar seria fácil. Com efeito, o problema

$$L_{fac} = \{(k, a) : \exists d\,(1 < d < k \;\&\; a \bmod d = 0)\}$$

está em NP. Dado um inteiro a, com $a > 1$, o seu mais pequeno divisor primo pode encontrar-se por meio duma pesquisa binária na qual se consulta L_{fac} cerca de $\log_2 a$ vezes. Portanto, se $L_{fac} \in \mathsf{P}$ então poderíamos encontrar o mais pequeno divisor de a em tempo polinomial. Repetindo o processo, poderíamos factorizar a em tempo polinomial. Em suma,

$$\mathsf{P} = \mathsf{NP} \;\Rightarrow\; \text{factorizar é fácil.}$$

Tanto quanto se sabe, pode acontecer $\mathsf{P} \neq \mathsf{NP}$ e, ainda assim, ser fácil factorizar. Há mesmo quem encare seriamente esta possibilidade,

baseado na convicção de que ainda não se trabalhou suficientemente no assunto.

Alguns problemas que advêm da teoria dos números têm um estatuto muito interessante e curioso. Por exemplo, o problema COMPOSTO está em NP, mas é muito duvidoso que seja NP-completo. Em 1975, Vaughan Pratt mostrou que o problema de decidir se um dado número inteiro é, ou não, primo está em NP. Dito de outro modo, os números primos têm certificados sucintos! Por conseguinte, COMPOSTO é um problema que está em NP e cujo problema complementar (essencialmente o problema PRIMO há pouco mencionado) também está em NP. Pensa-se que os problemas complementares de problemas NP-completos não estejam em NP pois, caso contrário, *todos* os problemas em NP teriam os seus complementares também em NP.

Um pouco surpreendentemente, em 1976, Gary Miller mostrou que PRIMO $\in$ P, *desde* que se admita uma determinada conjectura matemática: a Hipótese Generalizada de Riemann[1]. Entre os especialistas de teoria dos números há a convicção de que esta hipótese é verdadeira. Note-se, porém, o seguinte: ainda que PRIMO $\in$ P, não se segue necessariamente que factorizar seja fácil, pois poderá não haver nenhuma redução em tempo polinomial do problema de procura (search problem) *factorização* para o problema de decisão PRIMO. A inexistência de tal redução foi recentemente conjecturada por Stephen Cook.

5 Súmula

No dia 24 de Maio de 2000, numa conferência no Collège de France em Paris, o Clay Mathematics Institute de Cambridge (Massachusetts) pôs a concurso sete problemas de matemática. O prémio atribuído à solução de cada um destes "Millennium Prize Problems" é de um milhão de dólares americanos. O primeiro problema da lista é o problema P versus NP. De algum modo, este problema é diferente dos restantes, na medida em que pertence a uma área científica relativamente recente, sem o corpo de conhecimentos e grau de sofisticação matemática de outras áreas mais nobres e antigas da matemática.

[1] A Hipótese de Riemann aparece referida no artigo de Nuno Costa Pereira, no presente volume

A maior parte dos especialistas acredita que $\mathsf{P} \neq \mathsf{NP}$, ainda que tenham havido — e, certamente, continuarão a haver — tentativas sérias de decidir em tempo polinomial determinados problemas NP-completos. O método *par excellence* de separar classes de complexidade computacional é o método da diagonalização. Este método perpassa a lógica matemática e teve a sua origem há mais de um século num célebre argumento de Georg Cantor no âmbito da teoria dos conjuntos. Infelizmente, tem sido impotente para separar as classes P e NP. Métodos alternativos, como argumentos combinatoriais complicados na teoria dos circuitos Booleanos, parecem estar longe de conseguir abordar a plenitude do problema. Sendo verdade que nos últimos trinta anos se aprendeu muito acerca da natureza do problema P versus NP, também não deixa de ser verdade que ninguém hoje sabe realmente como abordar o problema. Não obstante, não se pode excluir que alguém, com um golpe de asa, consiga separar P de NP através dum argumento relativamente elementar.

Q:	$\mathsf{P} = \mathsf{NP}$?
R:	Não.
Q:	Factorizar é difícil?
R:	A respeito desta questão, sou agnóstico.
Q:	Quando será resolvido o problema P versus NP?
R:	*Circa* 2015.
Q:	Quem o resolverá?
R:	Talvez o meu caro leitor!

6 Anexo sobre máquinas de Turing

Dado um alfabeto Σ, uma máquina de Turing é, informalmente, uma sequência finita de instruções que rege uma cabeça móvel de leitura e escrita a qual, em cada momento, visiona uma célula duma fita de trabalho. Esta fita é infinitamente prolongável em ambas as direcções e está dividida em células contíguas. Em cada célula está inscrito um símbolo de Σ ou, então, não está nada inscrito (neste caso, também se diz que a célula está em branco, ou que nela está o símbolo especial b). Em cada momento da computação, a máquina de Turing está num determinado estado q de entre um conjunto (finito), previamente especificado, de estados Q. O que a máquina faz no

momento seguinte depende do estado em que está, do símbolo que a cabeça está a ler e daquilo que consta na lista de instruções (função de transição).

Definição. Seja Σ um alfabeto e $b \notin \Sigma$. Uma *máquina de Turing (determinista)* M no alfabeto Σ é um par (Q, Θ), onde Q é um conjunto finito não vazio com três elementos especiais q_0, q_s e q_n, e onde Θ é uma função de $(Q \backslash \{q_s, q_n\}) \times (\Sigma \cup \{b\})$ para $Q \times (\Sigma \cup \{b\}) \times \{-1, 1\}$.

Suponhamos que $\Theta(q, \sigma) = (q', \sigma', \delta)$, que M está no estado q e que a sua cabeça está sobre uma célula cujo símbolo inscrito é σ. No passo subsequente, M passa para o estado q', substitui na célula em questão o símbolo σ por σ' e move a cabeça para a esquerda ou para a direita conforme δ é -1 ou 1, respectivamente. As próximas definições formalizam estas ideias.

Definição. Seja M uma máquina de Turing no alfabeto Σ. Uma *configuração* C de M é um triplo (x, q, y) com x e y palavras de $\Sigma \cup \{b\}$, $y \neq \epsilon$ (i.e., y não é a palavra vazia) e $q \in Q$.

Sendo $y \neq \epsilon$, então y é da forma sw, com $s \in \Sigma \cup \{b\}$ e $w \in (\Sigma \cup \{b\})^*$. Dito de outro modo, y é a concatenação dum símbolo de $\Sigma \cup \{b\}$ com uma palavra (possivelmente vazia) de $\Sigma \cup \{b\}$. Dizer que a máquina M está na configuração (x, q, sw) significa dizer que M está no estado q, que a sua cabeça está sobre uma célula cujo símbolo inscrito é s, que à esquerda dessa célula consta a palavra x e que à sua direita consta a palavra w. Com esta interpretação, seguem-se duas definições naturais:

Definição. Sejam C e C' configurações duma máquina de Turing M. Diz-se que $C \to_M C'$ se $C = (x, q, sy)$, $\Theta(q, s) = (q', s', h)$ e se se der um dos seguintes casos:

1. $h = 1$, $y \neq \epsilon$ e $C' = (xs', q', y)$.
2. $h = 1$, $y = \epsilon$ e $C' = (xs', q', b)$.
3. $h = -1$, x é da forma $x't$, onde $t \in \Sigma \cup \{b\}$, e $C' = (x', q', ts'y)$.
4. $h = -1$, $x = \epsilon$ e $C' = (\epsilon, q', bs'y)$.

Definição. A *computação finita* da máquina de Turing M com entrada $x \in \Sigma^* \setminus \{\epsilon\}$ é, quando existe, a única sequência finita $C_0, C_1, \ldots, C_k$ de configurações tal que $C_0 = (\epsilon, q_0, x)$, $C_i \to_M C_{i+1}$ para todos os $i < k$, e tal que o estado de C_k é q_s ou q_n.

Observe-se que, dada uma entrada, pode não haver uma computação finita, devido à circunstância de nunca se atingir um estado de paragem q_s ou q_n (chamados estados de aceitação e de rejeição, respectivamente).

Definição. Seja L um problema no alfabeto Σ. Diz-se que L é *decidível em tempo polinomial* se existir uma máquina de Turing M no alfabeto Σ e se houver um polinómio $p(X) \in \mathbb{N}[X]$ tais que, para todas as entradas $x \in \Sigma^*$,

1. há uma computação finita $C_0, C_1, \ldots, C_k$ com $k \leqslant p(|x|)$,
2. se o estado de C_k é q_s então $x \in L$,
3. se o estado de C_k é q_n então $x \notin L$.

7 Brevíssima nota bibliográfica

Uma concisa e excepcional exposição sobre o problema P versus NP — que inclui um pouco da sua história e comentários sobre os seus desenvolvimentos mais recentes — deve-se ao próprio Stephen Cook no artigo de apresentação do problema para os "Millennium Prize Problems". Neste artigo, o leitor também pode encontrar referências bibliográficas pertinentes. O artigo intitula-se "The P versus NP Problem" e encontra-se disponível no endereço URL:

`http://www.claymath.org/millennium-problems/p-vs-np-problem`

O livro "Algorithmics: the Spirit of Computing" de David Harel (Addison-Wesley, 2ª edição, 1992) é uma excelente introdução à algoritmia e à complexidade computacional para leitores com uma formação universitária científica geral. Uma introdução mais tradicional a estes assuntos — de facto, um excelente manual universitário — é o (longo) texto "Introduction to Algorithmics" de Thomas Cormen, Charles Leiserson, Ronald Rivest e Clifford Stein (The MIT Press, 2ª edição, 2001).

Sobre o assunto da complexidade computacional propriamente dita, recomendo o livro de Christos Papadimitriou intitulado, precisamente, "Computational Complexity" (Addison-Wesley, 1994). A meu ver, este livro tem a clarividência de colocar a lógica matemática como firme peça central nestas coisas da ciência da computação. Trata-se

dum livro avançado, ao nível dum quarto ano de uma licenciatura em Informática ou Matemática. Em português, Jorge Orestes Cerdeira tem umas notas na *internet* intituladas "Umas Coisitas de Grafos e uma Introdução Informal à Complexidade Computacional". O endereço URL é:

`http://math.isa.utl.pt/~mlc/AlgProg/AlgProg0304/grafos_complex.pdf`

Com excepção desta última referência, todas as outras abordam, com maior ou menor detalhe, questões criptográficas. Carlos Sarrico tem um pequeno artigo em português intitulado "Os Números Primos e o Sistema de Codificação R.S.A." (Boletim da Sociedade Portuguesa de Matemática nº 33, 1995, pp. 81-89) que descreve e analisa o sistema criptográfico RSA. Para um estudo mais sistemático, recomendo "A Course in Number Theory and Cryptography" de Neal Koblitz (2ª edição, Graduate Texts in Mathematics 114, Springer-Verlag, 1994).

Finalmente, no apêndice do livro "Computers and Intractability: a Guide to the Theory of NP-completeness" de Michael Garey e David Johnson (W. H. Freeman and Co., 1979), o leitor pode encontrar uma lista com cerca de três centenas de problemas NP-completos.

8 Agradecimentos

Gostaria de agradecer a Miguel Ramos pelo amável convite para participar no ciclo de conferências "2000 matemática $\sqrt{\text{radical}}$" e pelo cuidado e interesse que, tanto ele como Jorge Nuno Silva, tiveram com a edição deste artigo.

Isabel Oitavem e Jorge Orestes Cerdeira leram uma versão preliminar deste artigo, tendo sugerido algumas correcções e melhoramentos. Agradeço-lhes o interesse e a disponibilidade manifestados. É claro que a responsabilidade última pelo que aqui está escrito cabe apenas a mim próprio.

Classificação AMS 2000: 03, 68

Fernando Ferreira
Departamento de Matemática, Faculdade de Ciências
Campo Grande, Edifício C1, 3º piso, 1749-016 Lisboa
fjferreira@fc.ul.pt
`http://webpages.fc.ul.pt/~fjferreira/`