

DOIS PROBLEMAS PROVANDO $P \neq NP$

Valdir Monteiro dos Santos Godoi
Bacharel em Matemática Aplicada e Computacional pela USP
valdir.msgodoi@gmail.com

RESUMO. Prova-se que $P \neq NP$, mostrando-se 2 problemas que são executados em tempo de complexidade polinomial em um algoritmo não determinístico, mas em tempo de complexidade exponencial em relação ao tamanho da entrada num algoritmo determinístico. Os algoritmos são essencialmente simples para que tenham ainda alguma redução significativa em sua complexidade, o que poderia invalidar as provas aqui apresentadas.

Mathematical subject classification: 68Q10, 68Q15, 68Q17.

Palavras-Chaves: $P \times NP$, P versus NP , $P \neq NP$, $P \neq NP$, $P \neq NP$, $P \leftrightarrow NP$, classe P , classe NP , algoritmo determinístico, algoritmo não determinístico.

INTRODUÇÃO

$P \neq NP$, conforme acreditam a maioria dos cientistas da computação.

$P \times NP$ é o mais importante problema em aberto da Ciência da Computação, e foi formulado em 1971, com o trabalho de Cook^[3], embora Karp^[16], Edmonds^[7,8,9], Hartmanis e Stearns^[12], Cobham^[2], von Neumann^[18] e outros também tenham contribuído para o estabelecimento desta questão. Para uma história mais exata e completa deste problema pode-se consultar, por exemplo, [11], [13] e [20].

O problema $P \times NP$ pretende determinar se toda linguagem L aceita em tempo polinomial por um algoritmo não determinístico ($L \in NP$) é também aceita em tempo polinomial por algum algoritmo determinístico ($L \in P$)^[4].

Se $P = NP$ então todo problema resolvido em tempo polinomial (em relação ao tamanho da entrada) por um algoritmo não determinístico também será resolvido em tempo polinomial (em relação ao tamanho da entrada) por um algoritmo determinístico.

Se $P \neq NP$ haverá ao menos um problema pertencente a NP que não terá para sua solução um algoritmo determinístico com complexidade polinomial em relação ao tamanho da entrada.

A classe NP contém uma grande quantidade de problemas importantes cuja solução em tempo polinomial só é possível ou de maneira aproximada ou em

casos particulares (mesmo que para infinitos casos particulares), por exemplo, os problemas da satisfatibilidade (SAT), da soma de subconjuntos, da mochila, do caixeiro-viajante, equação diofantina quadrática, congruência quadrática, etc. (veja, por exemplo, o capítulo 8 de [5]).

Como até hoje não se encontrou nenhum algoritmo “perfeito” e “rápido” (em tempo polinomial em relação ao tamanho da entrada) para se resolver estes problemas em NP, a opinião geral dos especialistas é que de fato $P \neq NP$ [5].

O que se fará neste artigo será mostrar 2 problemas simples cuja solução é em tempo polinomial num algoritmo não determinístico, mas em tempo não polinomial num algoritmo determinístico, provando que $P \neq NP$.

Embora não seja fundamental, adotaremos o ponto de vista de que uma Máquina de Turing não determinística funcionando em tempo polinomial tem a habilidade de pressupor um número exponencial de soluções possíveis e verificar cada uma em tempo polinomial, “em paralelo” [14]. É esta noção de paralelismo que adotaremos, de acordo também com outros autores, por exemplo, Diverio e Menezes [6]. Em particular, não adotaremos a idéia de que as máquinas de Turing “adivinham” uma solução correta, ou acertam sempre na primeira tentativa, como parece defender Papadimitriou *et al* em [5].

Dito de maneira equivalente, uma máquina não determinística é capaz de produzir cópias de si mesma quando diante de duas ou mais alternativas, e continuar a computação independentemente para cada alternativa ([16, p.91], [21]).

PROBLEMA 1

Nosso primeiro problema (p_1) a provar $P \neq NP$ fará uso da geração de números randômicos a partir de uma semente aleatória. Não precisaremos para isso recorrer ao modelo de Máquina de Turing Aleatória, como a descrita na seção 11.4 de [14], que utiliza uma fita de bits aleatórios, 0's e 1's.

Iremos recorrer aos usuais geradores de números pseudo-aleatórios disponíveis nas linguagens de programação, como `rand()` e `srand()` em C, e admitiremos que o tempo de execução destas instruções é de ordem polinomial.

Existem vários algoritmos conhecidos para a geração de números randômicos (veja, p.ex., [21]), sendo o mais conhecido deles o chamado método congruencial linear, $X_{k+1} = (a X_k + b) \pmod{m}$, mas para nosso propósito bastará considerá-los como uma “caixa-preta”, capaz de fornecer

como saída um número inteiro não negativo e aparentemente não previsível, em tempo polinomial.

Forneceremos um número natural s como parâmetro para a semente aleatória (X_0), obteremos o primeiro número randômico $\text{rand}()$ gerado com este parâmetro s e calcularemos f , tal que $f = \text{rand}() \pmod{n} + 1$. Ficaremos em loop, variando s de 1 até um valor máximo n , até que f seja igual a um dado valor y . Caso não obtenhamos $f = y$ o processamento terminará sem sucesso, no estado de rejeição da entrada $w = (n, y)$, e imprimirá “Não.”, caso contrário, obtendo $f = y$, terminaremos com sucesso, no estado de aceitação, imprimindo “Sim.”.

Nosso primeiro problema será então definido assim:

$p_1 =$ “Utilizando-se um gerador de números randômicos e dado uma entrada $w=(n,y)$, verificar se há uma semente aleatória s tal que o primeiro número pseudo-aleatório f obtido a partir dele, onde $f = \text{rand}() \pmod{n} + 1$, é igual a y , com $1 \leq y, s, f \leq n$, $(n, y, s, f) \in \mathbb{N}^4$.”.

Para não dar margem a dúvidas, mencionamos que este gerador de números randômicos não faz parte da entrada do nosso problema, e seu algoritmo, assim como também não todas as demais instruções necessárias.

Este gerador faz parte nativa do compilador utilizado, ou pode ser codificado segundo algum critério pré-estabelecido, e no caso das máquinas de Turing é parte intrínseca da codificação da máquina utilizada especificamente para a solução de p_1 , descrito através de sua função de transição. Por isso a entrada de p_1 é $w=(n, y)$.

O algoritmo determinístico (p_1D) é então conforme a seguir:

```
void p1D (int n, y)
{ int s, f;
  for (s = 1; s ≤ n; s++)
    {srand(s);
     f = rand()%n + 1;
     if (f == y)
       {printf(“Sim.\n”);
        return;
       }
    }
  printf(“Não.\n”);
  return;
}
```

O tempo de execução deste algoritmo no pior caso é da ordem $O(n p(L))$, onde $p(L)$ é um polinômio que mede a complexidade do cálculo de f em

função do tamanho L de n , $L = \lfloor \log(n) \rfloor$. O cálculo exato de p depende tanto do gerador quanto da máquina e compilador utilizados, mas para nossos propósitos bastará considerar que p é um polinômio, donde f não é obtido através de um procedimento exponencial no tamanho de L .

Então, sendo $n \propto 2^L$, supondo a entrada w na base 2, temos que a complexidade de p_1D é da ordem de $O(2^L p(L)) = O(2^{\lfloor w/2 \rfloor} p(\lfloor w/2 \rfloor))$, i.e., p_1D tem complexidade exponencial em relação ao tamanho da entrada $|w|$, donde $p_1 \notin P$.

O correspondente algoritmo não determinístico (p_1ND) deve fazer uso do mesmo gerador de números randômicos de p_1D , e deve reconhecer, aceitar e rejeitar as mesmas entradas que p_1D reconhece, aceita e rejeita, respectivamente.

Seguindo o estilo de Nivio Ziviani ^[22], por sua vez baseado em E. Horowitz e S. Sahni ^[15], p_1ND é então conforme a seguir:

```
void p1ND (int n, y)
{ int s, f;
  s = ESCOLHE(1..n);
  srand(s);
  f = rand()%n + 1;
  if (f == y)
    SUCESSO;
  else
    INSUCESSO;
}
```

Obviamente, este algoritmo usa o poder característico de uma Máquina de Turing não determinística, a capacidade de ser chamado simultaneamente por várias vezes, até que o sucesso ocorra.

As funções ESCOLHE, SUCESSO e INSUCESSO têm complexidade $O(1)$, assumiremos que $\text{rand}()$ tem complexidade dada por uma constante (não depende de parâmetros de entrada) e $\text{srand}(s)$ tem complexidade no máximo da ordem de $O(L)$, f é da ordem de $O(L^2)$, e a comparação *if* ($f == y$) tem complexidade $O(L)$, donde p_1ND tem complexidade dada por $O(L^2) = O(|w|^2/4) = O(|w|^2)$, i.e., uma complexidade de ordem polinomial em relação ao tamanho da entrada, donde $p_1 \in NP$.

Como $p_1 \in NP$, mas $p_1 \notin P$, então $P \neq NP$.

PROBLEMA 2

Nosso segundo problema (p_2) a provar $P \neq NP$ será bastante geral, de fato pode ser considerado como uma classe infinita de problemas, onde cada elemento refere-se a uma específica função f :

$p_2 = \text{“Existe } x \in [-n, n] \text{ tal que } f(x) = y, y \in [-n, n], n \in \mathbb{N}, (x, y) \in \mathbb{Z}^2\text{?”}$

$f(x)$ deve ser uma função computável em tempo polinomial e não pode ser uma função com alguma propriedade que possibilite resolver p_2 de maneira óbvia, como $f(x) = c$ ou $f(x) = ax^2 + bx + c$, ou mesmo através de um algoritmo polinomial no tamanho da entrada $w=(n,y)$, e deve ser tal que para resolvê-lo uma ótima solução é testar um a um os valores de $f(x)$ e compará-los com y .

Bons candidatos para $f(x)$ são polinômios de grau maior ou igual a 5, já que sabemos que equações algébricas, na forma geral $\sum_{k=0}^n a_k x^k = 0$, só tem soluções através de radicais e operações elementares até o grau $n=4$, conforme o Teorema de Abel-Ruffini (P. Ruffini, *Teoria generale delle equazioni, in cui si dimostra impossibile la soluzione algebraica delle equazioni generali di grado superiore al 4°*, 1799; e N.H.Abel, *Démonstration de l'impossibilité de la résolution algébrique des équations générales que passent le quatrième degré*, 1824).

É evidente que isto não significa que equações de grau 5 ou maior não tenham soluções, ou que estas não podem ser encontradas, mas apenas que não há um método único que forneça para elas, sempre, os valores de suas raízes em termos de um número finito de radicais e as operações elementares de adição, subtração, multiplicação, divisão, resto e exponenciação.

A solução geral para a equação de grau 5 é dada através de funções elípticas, conforme descoberto por Hermite (*Sur la résolution de l'équation du cinquième degré*, 1858), e as soluções das equações de grau n geral são dadas através das funções fuchsianas^[10] (ou automórficas), criadas por Poincaré, advindas das séries hipergeométricas e que geram tanto funções trigonométricas quanto elípticas^[19]. Em ambos os casos o cálculo não é simples e não tem complexidade de ordem polinomial em relação ao tamanho da entrada. A título de curiosidade, segundo a página da Wolfram (software *Mathematica*) [23], são necessários cerca de um trilhão de bytes para expressar a solução da quártica em termos de coeficientes simbólicos.

Outra possibilidade para se calcular as raízes das equações de grau maior ou igual a 5 (e equações em geral) é através de métodos numéricos (método da bisseção, de Newton, de Muller, de Monte Carlo, etc.), mas estes métodos podem não convergir para uma solução correta ou podem demorar

para convergir (veja, por exemplo, [1]), donde também não são a maneira mais eficiente para se responder ao problema mais simples aqui proposto, de verificar se há um solução de número inteiro no intervalo $[-n, n]$ para a equação $f(x) = y$.

Um algoritmo determinístico para p_2 (p_2D) para o caso particular de $f(x)$ um polinômio do sétimo grau (em lembrança ao décimo terceiro problema de Hilbert, resolvido pelos matemáticos russos Andrey Kolmogorov e seu aluno Vladimir Arnold), $f(x) = \sum_{k=0}^7 a_k x^k$, $-n \leq y \leq n$, $-n \leq a_i \leq n$, $n \in \mathbb{N}$, $y, a_i \in \mathbb{Z}$, é conforme a seguir:

```
void p2D (int n, y, a7, a6, a5, a4, a3, a2, a1, a0)
{ int x, f;
  for (x = -n; x ≤ n; x++)
    {f = ((((((a7*x + a6)*x + a5)*x + a4)*x + a3)*x + a2)*x + a1) + a0;
      if (f == y)
        {printf("Sim.\n");
          return;
        }
    }
  printf("Não.\n");
  return;
}
```

A complexidade deste algoritmo é de ordem exponencial em relação ao tamanho da entrada $w=(n, y, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$, donde $p_2 \notin P$.

Para a versão não determinística (p_2ND) podemos escrever

```
void p2ND (int n, y, a7, a6, a5, a4, a3, a2, a1, a0)
{ int x, f;
  x = ESCOLHE(-n .. n);
  f = ((((((a7*x + a6)*x + a5)*x + a4)*x + a3)*x + a2)*x + a1) + a0;
  if (f == y)
    SUCESSO;
  else
    INSUCESSO;
}
```

Tal qual a versão não determinística anterior, p_2ND tem complexidade de ordem polinomial em relação ao tamanho da entrada, donde $p_2 \in NP$.

Como $p_2 \in NP$, mas $p_2 \notin P$, então $P \neq NP$.

CONCLUSÃO

Mostramos dois problemas simples e seus respectivos algoritmos de solução. Ambos são resolvidos com complexidade polinomial em relação ao tamanho da entrada por um algoritmo não determinístico, caracterizando que ambos pertencem a NP.

As soluções pelos respectivos algoritmos determinísticos requerem ligeiras modificações, como era de se esperar, mas o “tempo” de execução aumenta de forma exponencial, caracterizando que estes problemas não pertencem a P, e que $P \neq NP$.

Os algoritmos não determinísticos se valem da vantagem de que a cada chamada escolhem uma alternativa e a testam, e quando a solução tentativa é a correta o processamento termina com sucesso. Por outro lado, se não há solução correta alguma dentre o conjunto de possibilidades escolhido o processamento termina sem sucesso, e teoricamente em tempo finito, sem loop infinito, demandando o mesmo tempo de que se houvesse uma solução válida.

Se um problema admite, por exemplo, 2^n alternativas para uma solução correta, a exemplo de SAT, um algoritmo não determinístico teria a capacidade de testar “simultaneamente” as 2^n alternativas, enquanto um algoritmo determinístico, seguindo a mesma estratégia, deveria testar alternativa por alternativa, sucessivamente (e não simultaneamente), e ainda controlar o momento de parada. Esta é uma diferença fundamental entre os dois tipos de algoritmos, e por isso seria extremamente improvável que $P = NP$.

Se $P = NP$ então não precisaríamos de nenhum recurso específico dos algoritmos não determinísticos para se resolver em tempo polinomial um problema computacional, em especial, da função ESCOLHE e sua propriedade de paralelismo.

Lembremos que usar a função ESCOLHE(C) não significa apenas escolher um dos elementos c_i de C e continuar o processamento para este elemento específico escolhido, e eventualmente se a escolha não for bem sucedida escolher outro elemento, etc. Significa escolher todos os elementos de C, processar uma versão do algoritmo para cada elemento de C, com todas estas versões processando simultaneamente, e parar se alguma destas versões levar ao Sucesso (resposta Sim).

Se um algoritmo tiver mais de uma função ESCOLHE, por exemplo, k, e cuja aceitação depender de um conjunto de escolhas acertadas, podemos

concluir que são produzidas até $\prod_{i=1}^k \#(C_i)$ cópias (ou versões) do algoritmo, onde $\#(C_i)$ é o número de elementos do conjunto C_i em ESCOLHE(C_i).

Como entender que uma quantidade potencialmente ilimitada de versões de um algoritmo não determinístico pode ser desprezada e produzir não só o mesmo resultado, mas no mesmo tempo de um algoritmo determinístico?

É claro que um algoritmo de complexidade de ordem exponencial pode eventualmente ser otimizado para um algoritmo de complexidade de ordem polinomial, por exemplo, a soma de uma P.A. A soma $S = \sum_{k=1}^n a_k = \sum_{k=1}^n a_1 + (k-1)r$ é de complexidade exponencial em relação ao tamanho da entrada (a_1, r, n), enquanto $S = (a_1 + a_n) n / 2 = (2a_1 + (n-1)r) n/2$ é de complexidade polinomial $O(|n^2r|)$, embora ambos conduzam à mesma solução.

Os algoritmos p_1D e p_2D apresentados, entretanto, já são suficientemente “simples” para que possam ter ainda uma redução exponencial em sua complexidade, e que invalidasse nossa prova de $P \neq NP$.

AGRADECIMENTOS

À professora Jordana Motta e aos professores Paulo Feofiloff, Janos Simon e Arnaldo Mandel.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] F.F.Campos, filho, *Algoritmos Numéricos*, LTC, Rio de Janeiro (2009), 271-320.
- [2] A. Cobham, *The intrinsic computational difficulty of functions*, in Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science, Y. Bar-Hillel, ed., Elsevier/North-Holland, Amsterdam (1964), 24–30.
- [3] S. Cook, *The complexity of theorem-proving procedures*, in Conference Record of Third Annual ACM Symposium on Theory of Computing, ACM, New York (1971), 151–158.
- [4] S. Cook, *The P versus NP Problem*, available in http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf, accessed in 02/22/2011.
- [5] S. Dasgupta, C. Papadimitriou and U. Vazirani, *Algoritmos*, McGraw-Hill Interamericana do Brasil Ltda., São Paulo (2009), 244.
- [6] T.A. Diverio and P.M Menezes, *Teoria da Computação – Máquinas Universais e Computabilidade*, Sagra Luzzatto, Porto Alegre (2004), 122-124.

- [7] J. Edmonds, *Maximum matchings and a polyhedron with 0,1-vertices*, Journal of Research at the National Bureau of Standards, Section B, 69 (1965), 125-130.
- [9] J. Edmonds, *Minimum partition of a matroid into independent subsets*, J. Res. Nat. Bur. Standards, Section B, 69 (1965), 67–72.
- [9] J. Edmonds, *Paths, trees and flowers*, Canadian Journal of Mathematics, 17 (1965), 449-467.
- [10] H. Eves, *Introdução à História da Matemática*, Editora da Unicamp, Campinas (1995), 306, 563.
- [11] L. Fortnow and S. Homer, *A Short History of Computational Complexity*, available in <http://people.cs.uchicago.edu/~fortnow/beatcs/column80.pdf>, accessed in 02/22/2011.
- [12] J. Hartmanis and R.E. Stearns, *On the computational complexity of algorithms*, Transactions of the AMS 117 (1965), 285-306.
- [13] R. Herken, ed., *The Universal Turing Machine – A Half-Century Survey*, Verlag Kammerer & Unverzagt, Hamburg-Berlin (1988).
- [14] J.E. Hopcraft, J.D. Ullman and R. Motwani, *Introdução à Teoria dos Autômatos, Linguagens e Computação*, Elsevier e Campus, Rio de Janeiro (2003), 452.
- [15] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press (1984), 501-510.
- [16] R.M. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R.E. Miller and J.W. Thatcher, eds., Plenum Press, New York (1972), 85–103.
- [17] C.I. Lucchesi, I.S.I. Simon, J. Simon and T. Kowaltowski, *Aspectos teóricos da computação*, IMPA, Rio de Janeiro (1979), 54, 110, available in <http://www.ime.usp.br/~is/atc/index.html>.
- [18] J. von Neumann, *A certain zero-sum two-person game equivalent to the optimal assignment problem*, in Contributions to the Theory of Games II, H.W. Kahn and A.W. Tucker, eds., Princeton Univ. Press, Princeton, NJ (1953), 5–12.
- [19] M. Paty, *A Criação Científica Segundo Poincaré e Einstein*, Estudos Avançados, 15, 41 (2001), available in http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0103-0142001000100013, accessed in 06/12/2011.
- [20] M. Sipser, *The History and Status of the P versus NP question*, Proceedings of the 24th Annual ACM Symposium on the Theory of Computing (1992), 603-619, available in <http://www.win.tue.nl/~gwoegi/P-versus-NP/sipser.pdf>, accessed in 02/22/2011.

[21] S. Tezuka, *Uniform Random Numbers; Theory and Practice*, Kluwer International Series in Engineering and Computer Science, Kluwer Academic, Norwell, Massachusetts (1995).

[22] N. Ziviani, *Projeto de Algoritmos com Implementações em Java e C++*, Thomson Learning, São Paulo, Brasil (2007), 381-383, 388, 551.

LINKS DA INTERNET

[23] <http://library.wolfram.com/examples/quintic/steps.html#quintic>, accessed in 06/12/2011.