DOIS PROBLEMAS PROVANDO P ≠ NP

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

RESUMO. Prova-se que P ≠ 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 x NP, P versus NP, P ≠ NP, P != NP, P <> NP, classe P, classe NP, algoritmo determinístico, algoritmo não determinístico.

INTRODUÇÃO

P ≠ NP, conforme acreditam a maioria dos cientistas da computação.

P x 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], Cobbam[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 x 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 ≠ 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 ≠ NP [5].

O que se fará neste artigo será mostrar dois problemas simples cuja solução é em tempo polinomial num algoritmo não determinístico, mas em tempo não polinomial num algoritmo determinístico, provando que P ≠ 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], [22]).

PROBLEMA 1

Nosso primeiro problema (p_1) a provar P ≠ 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. Nossa função rand() admitirá um parâmetro n, tal que rand(n) retornará um número inteiro positivo randômico entre 0 e n-1.

Existem vários algoritmos conhecidos para a geração de números randômicos (veja, p.ex., [21]), sendo o mais conhecido deles o chamado método congruencial linear, $X_{k+1} = (a\ X_k + b) \pmod n$, mas para nosso

propósito bastará considerá-los como uma “caixa-preta”, capaz de fornecer como saída um número inteiro não negativo entre 0 e n-1 e aparentemente não previsível, uma função não periódica e computada em tempo polinomial em relação ao tamanho do parâmetro n.

Forneceremos um número natural s como parâmetro para a semente aleatória (X_0), obteremos o primeiro número randômico rand(n) gerado com este parâmetro s e calcularemos f, tal que f = rand(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 dada 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 = rand(n) + 1, é igual a y, com 1 ≤ y, s, f ≤ 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 do cômputo do tamanho 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 nosso propósito bastará considerar que p é um polinômio, donde f não é obtido através de um procedimento exponencial no tamanho de L (se utilizássemos o método congruencial linear p seria um polinômio do segundo grau, mas também poderíamos usar funções mais complicadas para o cálculo de f, desde que não resultassem em complexidade exponencial, obviamente).

Então, sendo $n \alpha 2^L$, supondo a entrada w na base 2, temos que a complexidade de p_1D é da ordem de $O(2^L p(L)) = O(2^{|w|/2}\ p(|w|/2)) = O(2^{|w|}\ p(|w|))$, 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 srand(s) tem complexidade no máximo da ordem de $O(L)$ e rand(n) tem complexidade dada por um polinômio p(L), f é da ordem de $O(p(L))$, e a comparação *if (f == y)* tem complexidade $O(L)$, donde p_1ND tem complexidade dada por $O(p(L)) = O(p(|w|/2)) = O(p(|w|))$, i.e., uma

complexidade de ordem polinomial em relação ao tamanho da entrada, donde $L(M(p_1)) \in NP$.

Como $L(M(p_1)) \in NP$, mas $L(M(p_1)) \notin P$, 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, uma classe infinita de equações f(x)=y, onde cada elemento refere-se a uma específica função f:

p_2 = "Com precisão ε, existe $x \in \{-n, -n+\varepsilon, -n+2\varepsilon, ..., +n\}$ tal que x seja solução da equação f(x)=y, *i.e.*, $|f(x) - y| \leq \varepsilon$, com $n \in \mathbb{N}$, $\varepsilon = 10^{-k}$, $1 \leq k \leq n$, $k \in \mathbb{N}$, $y \in [-n, n]$, $(x, y) \in \mathbb{Q}^2$?"

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 com tempo polinomial no tamanho da entrada w=(n,y,f,k), e deve ser tal que para resolvê-lo uma ótima solução não determinística é testar um a um os valores de f(x) e compará-los com y, dentro da precisão requerida. Ou seja, f deve ser tal que $f^{-1}(x)$ não seja resolvida em tempo polinomial por um algoritmo determinístico.

Bons candidatos para f(x) são polinômios de grau maior ou igual a 5, já que sabemos que equações algébricas, na forma geral $\sum^{n}_{\kappa=0} a_k x^k = 0$, 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, *Démonstracion 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 cinquème 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 quíntica 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 podem não ser a maneira mais eficiente para se responder ao problema mais simples aqui proposto, de verificar se há um solução de número decimal e com precisão ε, no conjunto $\{-n, -n+\varepsilon, -n+2\varepsilon, ..., +n\}$, 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 (parcialmente[24]) pelos matemáticos russos Vladimir Arnold e Andrey Kolmogorov), $f(x) = \sum^{7}_{\kappa=0} a_k x^k$, $-n \le y \le n$, $-n \le a_i \le n$, $n \in \mathbb{N}$, $a_i \in \mathbb{Z}$, $(x, y) \in \mathbb{Q}^2$, é conforme a seguir:

```
void p2D (int n, double y, int a7, a6, a5, a4, a3, a2, a1, a0, k)
{ double ε, x, f;
  ε = pow(10, -k);
 for (x = -n; x ≤ n; x+=ε)
      {f = ((((((a7*x + a6)*x + a5)*x + a4)*x + a3)*x + a2)*x + a1)*x + a0;
       if (abs(f - y) ≤ ε)
          {printf("Sim.\n");
           return;
          }
      }
  printf("Não.\n");
  return;
}
```

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

Para a versão não determinística (p_2ND) podemos escrever

```
void p2ND (int n, double y, int a7, a6, a5, a4, a3, a2, a1, a0, k)
{ double ε, x, f;
  ε = pow(10, -k);
  x = ESCOLHE(-n, -n+ε, -n+2ε, ..., +n);
  f = ((((((a7*x + a6)*x + a5)*x + a4)*x + a3)*x + a2)*x + a1)*x + a0;
   if (abs(f - y) ≤ ε)
      SUCESSO;
   else
       INSUCESSO;
}
```

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

Como $L(M(p_2)) \in NP$, mas $L(M(p_2)) \notin P$, 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 $\Pi^{k}_{i=1}$ #(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 ainda produzir não só o mesmo resultado de um algoritmo determinístico, mas também possuir uma complexidade de ordem polinomial?

Supondo a implantação destes algoritmos determinísticos e não determinísticos no mundo real, em uma máquina de Turing que fosse efetivamente construída ou senão em um computador moderno, o processamento do algoritmo determinístico imprimiria o esperado “Sim” ou “Não” (ou Sucesso/Insucesso, Aceita/Rejeita, etc.) uma única vez, sem dubiedade, e pararia (ainda que demorasse muito tempo para parar, por exemplo, um tempo de ordem exponencial em relação ao tamanho da entrada e uma entrada de grande tamanho). Já o processamento na máquina não determinística imprimiria os seus *inúmeros* “Não”, ou alguns “Sim” misturados com alguns “Não”, talvez muitos “Não” e um único “Sim”, um “Sim” escondido entre *incontáveis* “Não”, ou “Sim” e “Não” intercalados, etc., e a saída deste processamento poderia ser tão sem controle, desorganizada e confusa no mundo real que tecnicamente, nas piores situações, poderíamos ficar sem saber qual a verdadeira e correta resposta ao problema dado. Mesmo que teoricamente o processamento parasse no *primeiro* “Sim”, poderia haver muitos “Sim” e “Não” simultâneos, portanto poderíamos também não ter como distinguir eficientemente se há algum “Sim”. Claro, podemos ter duas impressoras, ou dois outros dispositivos de entrada e saída, um exclusivo para informar o “Sim”, o outro para o “Não”, de modo a não se misturar os dois tipos distintos de respostas. Ainda assim, nessa situação melhorada de implementação, pensemos: controlar 2^n respostas (o tempo no pior caso) de uma entrada de tamanho da ordem de

O(n) se faz em tempo polinomial? Não... no mundo real, não. E “reescrever” 2^n vezes um programa? Não, com certeza não. Executar 2^n vezes um programa? Testar a parada da máquina (e de qual versão da máquina, se isso fosse importante e possível)?...

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 $S = \sum^{n}_{\kappa=1} a_k = \sum^{n}_{\kappa=1} a_1 + (k-1)r$ 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 $O(|n^2 r|)$, 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 ≠ 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-Hille, 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] Stephen 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, acessed in 06/12/2011.

[24] https://en.wikipedia.org/wiki/Hilbert%27s_problems, accessed in 03/05/2016.