DOIS PROBLEMAS PROVANDO P ≠ NP

Valdir Monteiro dos Santos Godoi
valdir.msgodoi@gmail.com

RESUMO. Prova-se que P ≠ NP, mostrando-se 2 problemas que são executados em tempo de complexidade constante *O*(1) 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[2], embora Karp[14], Edmonds[6,7,8], Hartmanis e Stearns[10], Cobham[1], von Neumann[16] e outros também tenham contribuído para o estabelecimento desta questão. Para uma história mais exata e completa deste problema pode-se consultar, por exemplo, [9], [11] e [17].

O problema P x NP pretende determinar quando uma linguagem L aceita em tempo polinomial por um algoritmo não determinístico ($L \in NP$) é também aceita em tempo polinomial por algum algoritmo determinístico ($L \in P$) [3].

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.

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 [4].

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 ≠ 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” [12]. É esta noção de paralelismo que adotaremos, de acordo também com outros autores, por exemplo, Diverio e Menezes [5]. Em particular, estaremos desprezando a idéia de que as máquinas de Turing “adivinham” uma solução correta, ou acertam sempre “na primeira”, como parece defender Papadimitriou *et al* em [4].

PROBLEMA 1

Nosso primeiro problema (p_1) é um problema de busca, com a característica especial de que ao invés de buscar um elemento em uma lista de números inputados por um usuário (o que aumentaria o tamanho de nossa entrada) o pesquisará diretamente na memória do computador, no estado em que estiver.

p_1 = “Lidos 2 endereços de memória (ponteiros) n_1 e n_2 e um caracter x verificar se da posição de memória n_1 a n_2 existe o elemento x.”

Supondo que cada ponteiro ocupe no máximo c caracteres o tamanho da entrada será no máximo TAM = 2c + 3 (ponteiros na base hexadecimal) e sua complexidade num algoritmo determinístico será da ordem $O(n_2) = O(16^{TAM/2})$, i.e., uma complexidade de ordem exponencial em relação ao tamanho da entrada. Como não podemos garantir que a memória esteja ordenada, o que nos possibilitaria realizar uma pesquisa binária, nem que ela contenha somente espaços, zeros, etc., não é possível nenhuma redução significativa no algoritmo, caracterizando que o problema não pertence a P.

O algoritmo determinístico é então conforme a seguir:

```
void p1D (char *n1, *n2, x)
{ char *i;
  for (i = n1; i ≤ n2; i++)
      if (*i == x)
         {printf("Sim.\n");
          return;
         }
  printf("Não.\n");
  return;
}
```

O correspondente algoritmo não determinístico, seguindo o estilo de Nivio Ziviani [19], por sua vez baseado em E. Horowitz e S. Sahni [13], é conforme a seguir:

```
void p1ND (char *n1, *n2, x)
{ char *i;
  i = ESCOLHE(n1 .. n2);
  if (*i == x)
     SUCESSO;
  else
     INSUCESSO;
}
```

Este algoritmo é de complexidade $O(1)$, donde $p_1 \in NP$, e obviamente 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. Caso x não se encontre nas posições de memória de n_1 a n_2 a execução não entrará em loop infinito, e parará.

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

É importante mencionar que não computamos como pertencente à entrada (*n_1, *n_2, x) do algoritmo a parte contínua da memória que vai das posições n_1 a n_2.

Os requisitos de tempo de um algoritmo são convenientemente expressos em termos do tamanho da entrada porque se espera que a dificuldade do problema varie rigorosamente com essa medida. Então, para expressar precisamente o requisito de tempo, é preciso ter o cuidado de definir o tamanho da entrada de maneira a levar em conta esses fatores[18].

Uma outra medida de complexidade de algoritmos importante é a "memória" utilizada na computação. No modelo de Máquina de Turing, isto corresponde, naturalmente, ao número de células da fita visitada pela cabeça de leitura/gravação da máquina de Turing durante a computação[15].

No famoso problema P x NP aqui tratado, entretanto, a complexidade do tempo é medida em função do tamanho da entrada (por exemplo, [3], [14]), e não do tamanho da memória, nem da entrada+memória. Se não fosse assim teríamos $p_1 \in P$ (além de não precisarmos mais dos ponteiros). Daqui também se vê que um problema de ordem exponencial em relação ao tamanho da entrada pode ser de ordem polinomial em relação ao tamanho da memória utilizada.

Dito informalmente, os computadores já fornecem à disposição dos programas nele em operação um determinado conteúdo de memória que, ao menos nas linguagens C e Assembly, pode ser acessado rápida e diretamente. A memória do computador é assim uma parte contida implicitamente em todos os programas (e respectivos algoritmos), recebida "de graça", sem precisar de nenhuma entrada extra, ela já está "dentro" o tempo todo, e ao menos em teoria já pode ser utilizada para leitura, sem a necessidade de ser "repassada" como parâmetro de entrada, o que poderia aumentar enormemente o tamanho da entrada. Semelhantemente, variáveis de trabalho, as inúmeras variáveis auxiliares que podem ser utilizadas em um algoritmo, também não são computadas como pertencentes à entrada do algoritmo. Verifica-se assim que p_1 é um problema específico para computadores, capazes de operar ao nível de byte, posição de memória.

Convém destacar ainda que nosso raciocínio está de acordo também com o modelo de Máquina de Turing de Acesso Aleatório. Uma fita pode ser usada simultaneamente como dispositivo de entrada, de saída e de memória de trabalho, ou ser usado uma fita para cada tipo de operação, e portanto o que entendemos por posição de memória pode referir-se a uma determinada posição (finita) de uma fita de comprimento (teoricamente) infinito, com qualquer valor x desconhecido: "sujeira" ou "lixo" de algum processamento anterior (na hipótese realística de que os algoritmos não tem a necessidade de limpar (inicializar) ao final do processamento todas as variáveis de trabalho e de I/O que utilizaram), ou pertencente ao processamento atual (já que nossos algoritmos podem ser apenas uma pequena parte de um algoritmo maior), ou de algum processamento paralelo. Portanto, fica claro que num modelo mais avançado de máquina (seja de Turing ou não), o conteúdo de memória não precisa vir de um dispositivo de entrada e saída (modelos RAM).

PROBLEMA 2

Nosso segundo problema (p_2), embora à primeira vista pareça não ter utilidade prática alguma, demonstra claramente que $P \neq NP$.

Trata-se apenas da geração de números aleatórios y até que o número gerado seja igual a um parâmetro dado x. Lê-se como entrada dois números inteiros n e x, com $1 \leq x \leq n$ e $1 \leq y \leq n$. Para deixá-lo no formato de um problema de decisão, com resposta entre Sim e Não (embora P e NP não contenham apenas problemas de decisão, por exemplo, [14], [15], [19]), nossos algoritmos responderão se x é obtido na primeira iteração em execução ou não.

Na versão determinística (p_2D) a probabilidade de se obter o valor de x é igual a 1/n, admitindo-se distribuição uniforme na geração dos números, e em média deve-se esperar n repetições até que o número y gerado seja igual a x, um número de repetições que é de ordem exponencial em relação ao tamanho da entrada. Então $p_2 \notin P$.

```
void p2D (int n, x)
{ int y, cont;
  srand(time(NULL));
  cont = 0;
  do
     {cont += 1;
      y = rand()%n + 1;
      }
  while (y != x);
  if (cont <= 1)
      printf("Sim.\n");
  else
      printf("Não.\n");
  return;
}
```

Na versão não determinística (p_2ND) o tempo de execução é da ordem $O(1)$. Então $p_2 \in NP$.

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

```
void p2ND (int n, x)
{ int y;
  cont = 0;
  do
     {cont += 1;
      y = ESCOLHE(1 .. n);
      }
  while (y != x);
  if (cont <= 1)
     SUCESSO;
  else
     INSUCESSO;
}
```

Neste segundo algoritmo a função *ESCOLHE* toma o lugar da função geradora de números aleatórios, o que é perfeitamente compatível com a característica de um algoritmo não determinístico. Esta mesma função também poderia tomar o lugar da expressão que gera os números aleatórios na versão determinística, e ser definida, por exemplo, assim:

```
int ESCOLHE(int y1, y2)
{ int y;
  y = y1 + rand()%(y2 – y1 + 1);
  return y;
}
```

CONCLUSÃO

Mostramos dois problemas simples e seus respectivos algoritmos de solução. Ambos são resolvidos com complexidade polinomial em relação ao tamanho da entrada, mais exatamente, com complexidade constante $O(1)$, 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.

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 $S = (a_1 + a_n)\ n / 2 = n\ (a_1 + (n - 1)\ r / 2)$ é de complexidade constante $O(1)$, 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.

p_1D, por exemplo, se não usasse a instrução *for*, uma estrutura similar do tipo *while*, *repeat* ou *until*, ou ainda o péssimo algoritmo que conteria n_2 ou mais condições *if* sucessivas, todas resultando em complexidade exponencial, deveria usar uma instrução "supostamente" $O(1)$, do tipo $TESTA(*n_1, *n_2, x)$, mas que esconderia em si a verdadeira complexidade exponencial do problema.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] 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.

[2] 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.

[3] Stephen Cook, *The P versus NP Problem*, available in http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf, accessed in 22/02/2011.

[4] S. Dasgupta, C. Papadimitriou and U. Vazirani, *Algoritmos*, McGraw-Hill Interamericana do Brasil Ltda., São Paulo (2009), 244.

[5] T.A. Diverio and P.M Menezes, *Teoria da Computação – Máquinas Universais e Computabilidade*, Sagra Luzzatto, Porto Alegre (2004), 122-124.

[6] 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.

[7] J. Edmonds, *Minimum partition of a matroid into independent subsets*, J. Res. Nat. Bur. Standards, Section B, 69 (1965), 67–72.

[8] J. Edmonds, *Paths, trees and flowers*, Canadian Journal of Mathematics, 17 (1965), 449-467.

[9] L. Fortnow and S. Homer, *A Short History of Computational Complexity*, available in http://people.cs.uchicago.edu/~fortnow/beatcs/column80.pdf, accessed in 22/02/2011.

[10] J. Hartmanis and R.E. Stearns, *On the computational complexity of algorithms*, Transactions of the AMS 117 (1965), 285-306.

[11] R. Herken, ed., *The Universal Turing Machine – A Half-Century Survey*, Verlag Kammerer & Unverzagt, Hamburg-Berlin (1988).

[12] 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.

[13] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press (1984), 501-510.

[14] 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

[15] C.I. Lucchesi, I.S.I. Simon, J. Simon and T. Kowaltowski, *Aspectos teóricos da computação*, IMPA, Rio de Janeiro (1979), 54, 110.

[16] 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.

[17] 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 22/02/2011.

[18] L.V. Toscani and P.A.S. Veloso, *Complexidade de Algoritmos*, Sagra Luzzatto, Porto Alegre (2005), 225.

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