SISTEMAS DA INFORMAÇÃO

MATERIAL INSTRUCIONAL ESPECÍFICO

TOMO 5

CQA/UNIP – Comissão de Qualificação e Avaliação da UNIP

SISTEMAS DA INFORMAÇÃO

MATERIAL INSTRUCIONAL ESPECÍFICO

TOMO 5

Christiane Mazur Doi

Doutora em Engenharia Metalúrgica e de Materiais, Mestra em Ciências - Tecnologia Nuclear, Especialista em Língua Portuguesa e Literatura, Engenheira Química e Licenciada em Matemática, com Aperfeiçoamento em Estatística. Professora titular da Universidade Paulista.

Tiago Guglielmeti Correale

Doutor em Engenharia Elétrica, Mestre em Engenharia Elétrica e Engenheiro Elétrico (ênfase em Telecomunicações). Professor titular da Universidade Paulista.

Questão 1

Questão 1.[1]

O problema P versus NP é um problema ainda não resolvido e um dos mais estudados em Computação. Em linhas gerais, deseja-se saber se todo problema cuja solução pode ser eficientemente verificada por um computador, também pode ser eficientemente obtida por um computador. Por "eficientemente" ou "eficiente" significa "em tempo polinomial". A classe dos problemas cujas soluções podem ser eficientemente obtidas por um computador é chamada de classe P. Os algoritmos que solucionam os problemas dessa classe têm complexidade de pior caso polinomial no tamanho das suas entradas. Para alguns problemas computacionais, não se conhece solução eficiente, isto é, não se conhece algoritmo eficiente para resolvê-los. No entanto, se para uma dada solução de um problema é possível verificá-la eficientemente, então o problema é dito estar em NP. Dessa forma, a classe de problemas para os quais suas soluções podem ser eficientemente verificadas é chamada de classe NP. Um problema é dito ser NP-completo se pertence à classe NP e, além disso, se qualquer outro problema na classe NP pode ser eficientemente transformado nesse problema. Essa transformação eficiente envolve as entradas e saídas dos problemas. Considerando as noções de complexidade computacional apresentadas acima, analise as afirmações que se seguem:

I. Existem problemas na classe P que não estão na classe NP.
II. Se o problema A pode ser eficientemente transformado no problema B e B está na classe P, então A está na classe P.
III. Se P = NP, então um problema NP-completo pode ser solucionado eficientemente.
IV. Se P é diferente de NP, então existem problemas na classe P que são NP-completos.

É correto apenas o que se afirma em

A. I.
B. IV.
C. I e III.
D. II e III.
E. II e IV.

[1]Questão 36 - Enade 2011.

1. Introdução teórica

1.1. Medindo a complexidade de algoritmos

Nos últimos 20 anos, o poder de processamento dos computadores cresceu de forma notável. Máquinas atuais, mesmo as mais baratas, têm capacidades de processamento muito superiores aos computadores disponíveis no início da década de 1990. As pessoas, com base nesse notável desenvolvimento ocorrido nos últimos anos, podem pensar que o projeto de algoritmos não é mais tão importante, dado que as máquinas atuais são muito poderosas. Contudo, esse pensamento não está correto.

Para entender o porquê, iniciemos a observação da tabela 1 abaixo.

n	$n \,.\, ln(n)$	n^2	e^n	$n!$
10	$\approx$ 23,1	100	$\approx$ 22000,5	3628800
100	$\approx$ 460,5	10000	$\approx 2.10^{43}$	$\approx 10^{157}$
1000	$\approx$ 6907,8	1000000	$\approx 2.10^{434}$	$\approx 10^{2567}$

Tabela 1. Crescimento de diversas funções.
Fonte. SKIENA, 1997 (com adaptações).

Se compararmos o crescimento das funções, podemos notar claramente como as duas primeiras colunas apresentam números bem menores do que as demais. Se esses números representarem uma estimativa de tempo de execução de um algoritmo, isso significa que os algoritmos que rodarem com tempos estimados pelas duas primeiras colunas vão rodar em tempo muito menor do que os demais, independentemente da máquina em que estiverem sendo executados (e, portanto, são muito mais rápidos). Logo, algoritmos ruins são indesejados, não importa a rapidez da tecnologia.

A análise da complexidade de algoritmos permite-nos identificar se um algoritmo é bom ou ruim, estimando o seu consumo de recursos. Dois aspectos podem ser analisados: o tempo de execução e o consumo de memória. Ambos são os recursos mais escassos que temos à disposição: tempo e espaço. Ao projetar ou analisar um algoritmo, devemos pensar qual desses dois recursos é o mais crítico e compararmos as diversas opções, levando em consideração que algumas vezes podemos trocar tempo de processamento por espaço ou vice-versa.

1.2. Máquinas de Turing

Devido à constante evolução tecnológica da computação, apenas executar um programa e medir o seu tempo de execução não são medidas adequadas da sua complexidade. Rapidamente, essa medida poderia tornar-se obsoleta. Além disso, arquiteturas computacionais diferentes podem influenciar fortemente o tempo de execução. Nesse sentido, não estaríamos apenas medindo a eficiência do programa em questão, mas também da arquitetura do computador utilizado.

Para termos uma forma independente de avaliar a complexidade de um algoritmo, podemos contar o número de passos de sua execução em um computador hipotético, sem nos preocuparmos com o *clock* ou com uma arquitetura específica.

Em 1936, um dos fundadores da computação moderna, Alan Turing, propôs uma máquina hipotética chamada de Máquina de Turing (SIPSER, 2012). Essa máquina é capaz de executar qualquer algoritmo que um computador possa executar e, portanto, é um ótimo modelo computacional independente de uma arquitetura específica, como podemos observar na figura 1.

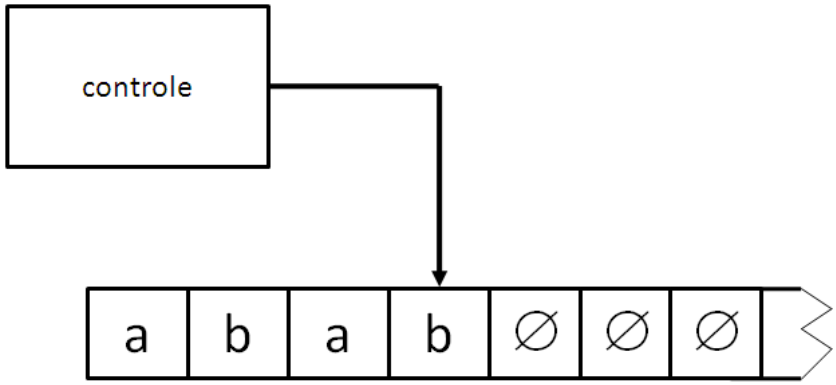


Figura 1. Diagrama hipotético de uma máquina de Turing.
Fonte. SIPSER, 2012 (com adaptações).

De acordo com Sipser (2012), podemos definir formalmente uma máquina de Turing como uma 7-upla $(Q, \Sigma, \Gamma, \delta, q_0, q_{aceita}, q_{rejeita})$, em que Q, Σ, Γ são todos conjuntos finitos e

- Q é o conjunto de todos os estados;
- Σ é o alfabeto de entrada sem o símbolo em branco ($\varnothing$);
- Γ é o alfabeto da fita, sendo $\varnothing \in \Gamma$ e $\Sigma \subseteq \Gamma$;
- δ: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{E,D\}$ é a função de transição;

- $q_0 \in Q$ é o estado inicial;
- $q_{aceita} \in Q$ é o estado de aceitação; e
- $q_{rejeita} \in Q$ é o estado de rejeição, com $q_{rejeita} \neq q_{aceita}$.

Devemos observar que a função de transição δ determina como a função comporta-se a cada interação do algoritmo (SIPSER, 2012). Devemos ter em mente que a cabeça da máquina de Turing pode deslocar-se tanto para a esquerda quanto para a direita da fita (aqui representado por E ou D) e que ela pode ler ou escrever na fita. Além disso, a computação dura até a máquina alcançar um dos estados $q_{rejeita}$ ou q_{aceita}, ou, em alguns casos, a computação pode não parar nunca, indicando que o algoritmo nunca alcança um dos estados de aceitação ou rejeição.

Assim como os autômatos finitos, uma máquina de Turing pode ser determinística ou não determinística. Contudo, toda máquina de Turing não determinística apresenta uma máquina de Turing determinística equivalente (SIPSER, 2012).

1.3. Medindo a complexidade no tempo

Com base na descrição que fizemos sobre máquinas de Turing, podemos definir o tempo de execução ou a complexidade de tempo como o número máximo de passos que uma máquina de Turing leva para executar um algoritmo sobre uma entrada de tamanho n (SIPSER, 2012).

Outras definições também são possíveis: por exemplo, podemos pensar no tempo médio de execução ou no tempo mínimo.

1.4. Notação O-grande

Frequentemente, não precisamos saber o tempo de execução exato de um algoritmo. Faz pouca diferença dizer que um algoritmo dura 1000,2 ou 1001,0 segundos. De forma geral, estamos interessados apenas na ordem de grandeza do tempo de execução.

Para indicar a ordem de grandeza do crescimento de uma função, podemos utilizar a notação O-grande. Nesse caso, suponha duas funções f e g com $f,g: \mathbb{N} \rightarrow \mathbb{R}^+$. De acordo com SIPSER (2012), dizemos que $f(n)=O(g(n))$ se existirem dois inteiros positivos c e n_0 tais que, para todo inteiro $n \geq n_0$, temos $f(n) \leq c.g(n)$.

Quando $f(n)=O(g(n))$, dizemos que $g(n)$ é um limite superior para $f(n)$ ou, mais precisamente, que $g(n)$ é um limite assintótico superior para $f(n)$, a fim de enfatizar que estamos suprimindo fatores constantes (SIPSER, 2012).

1.5. Classes P, NP e NP-completo

A classe P consiste em problemas que podem ser resolvidos em tempo polinomial (CORMEN, 2001). Especificamente, dizemos que, nesses casos, esses problemas podem ser resolvidos em $O(n^k)$, sendo k uma constante e n o tamanho da entrada do problema (CORMEN, 2001). Isso não significa que esses problemas sejam necessariamente rápidos em um computador convencional, já que um polinômio de grau elevado, com potências maiores do que 3, tendem a consumir tempos extensos. Mas, se comparados com problemas exponenciais, esses algoritmos são muito mais rápidos.

Alguns problemas não podem ser resolvidos em tempo polinomial. Porém, se tivermos uma resposta candidata, podemos verificar se essa resposta está correta em tempo polinomial. Esses problemas são chamados de NP. Devemos prestar atenção ao fato de que NP quer dizer (nondetermistic polynomial time) ou, em português, polinomial não determinista (no tempo). Não estamos afirmando que esses problemas não são polinomiais, porque isso ainda não foi provado matematicamente; estamos afirmando que uma máquina de Turing não-determinista é capaz de resolver esses problemas em tempo polinomial ou que as respostas podem ser verificadas em tempo polinomial, o que é equivalente (SIPSER, 2012).

Não se conhece, até o presente momento, um algoritmo polinomial que resolva o problema em uma máquina de Turing determinista, porém ninguém provou que o algoritmo não existe. Dessa forma, o problema continua aberto.

Quando um problema está na classe NP e pode ser eficientemente transformado em outro problema da classe NP, dizemos que o problema é do tipo NP-completo.

2. Análise das afirmativas

I – Afirmativa incorreta.

JUSTIFICATIVA. Problemas na classe P também podem ser verificados em tempo polinomial, e, portanto, também estão na classe NP.

II – Afirmativa correta.

JUSTIFICATIVA. Se soubermos resolver o problema B de forma eficiente, e tivermos um algoritmo eficiente capaz de transformar o problema A no problema B, então temos uma forma eficiente de resolver o problema A. Basta aplicarmos o algoritmo de conversão de A para B. Dessa forma, podemos dizer que ambos os problemas pertencem à mesma classe.

III – Afirmativa correta.

JUSTIFICATIVA. Dizer que P=NP significa dizer que todos os problemas na classe NP podem ser resolvidos por um algoritmo polinomial, que é muito mais rápido do que um algoritmo exponencial. Teoricamente, isso significa que esses problemas podem ser resolvidos de forma eficiente. Contudo, devemos ter em mente que, se trabalharmos com polinômios de ordem elevada, podemos ter um tempo de execução potencialmente elevado. De qualquer forma, o crescimento de uma função polinomial é bem menor do que de uma função exponencial.

IV – Afirmativa incorreta.

JUSTIFICATIVA. Se considerarmos que P≠NP (o que ainda não foi provado, mas pode ser adotado como uma suposição), um problema NP-completo certamente não poderia pertencer à classe P.

Alternativa correta: D.

3. Indicações bibliográficas

- CORMEN, T. H. et al. *Introduction to algorithms*. Cambridge: MIT Press, 2001.
- LEWIS, H. R.; PAPADIMITRIOU, C. H.; NETO, J. J. *Elementos de teoria da computação*. Porto Alegre: Bookman, 2004.
- SIPSER, M. *Introdução à teoria da computação*. São Paulo: Cengage Learning, 2012.
- SKIENA, S. S. *The algorithm design manual*. New York: Springer-Verlag, 1998.

Questão 2

Questão 2.[2]

Uma antiga empresa de desenvolvimento de software resolveu atualizar toda sua infraestrutura computacional adquirindo um sistema operacional multitarefa, processadores multi-core (múltiplos núcleos) e o uso de uma linguagem de programação com suporte a threads. O sistema operacional multitarefa de um computador é capaz de executar vários processos (programas) em paralelo. Considerando esses processos implementados com mais de uma thread (multi-threads), analise as afirmações abaixo.

I. Os ciclos de vida de processos e threads são idênticos.
II. Threads de diferentes processos compartilham memória.
III. Somente processadores multi-core são capazes de executar programas multi-threads.
IV. Em sistemas operacionais multitarefa, threads podem migrar de um processo para outro.

É correto apenas o que se afirma em

A. I.
B. II.
C. I e III.
D. I e IV.
E. II e IV.

1. Introdução teórica

Processos e threads

Quando escrevemos um programa de computador em alguma linguagem não interpretada (por exemplo, em linguagem C) e geramos um executável, estamos criando uma entidade passiva, que ainda não é um processo (DHAMDHERE, 2006). Essencialmente, trabalhamos com um texto, o nosso código fonte, que, posteriormente, será transformado em binário executável.

Porém, quando executamos um programa, especialmente quando executamos o binário gerado pelo compilador, o sistema operacional gera um processo, que será uma execução do nosso programa. Devemos salientar que essa é uma das várias possíveis execuções, já que poderíamos, na maioria dos casos, fazer várias execuções de um mesmo

[2]Questão 29 - Enade 2011.

programa. Assim, um único programa poderia gerar vários processos, caso fosse executado várias vezes (DHAMDHERE, 2006).

Essa não é a única forma de gerar um processo. Por exemplo, na linguagem C e no ambiente Unix, podemos utilizar a chamada de sistema *fork()*, que cria um processo filho. Nesse caso, o processo pai faz uma cópia de si mesmo e gerar um processo filho. Existem formas de agilizar esse processo, mas ainda assim o programa filho vai ser executado em um espaço de endereçamento diferente do processo pai. Dessa maneira, mesmo sendo executado uma única vez, programas que utilizem a chamada *fork()* podem gerar vários processos (especialmente se o processo filho também executar a chamada *fork()* em outros momentos).

Ainda que esse modo de trabalho funcione bem, a geração de um processo é cara. Além disso, há momentos em que queremos compartilhar memória, bem como recursos entre as execuções paralelas. Mesmo que isso seja possível utilizando-se apenas processos, outra possível solução, potencialmente mais simples, é a utilização de threads.

Devemos ter em mente que tanto processos quanto threads têm como finalidade permitir a execução concorrente de um programa (ou de partes de um programa) (DHAMDHERE, 2006). Isso depende do sistema operacional utilizado, bem como do hardware.

É importante notarmos que máquinas com mais de um processador ou com processadores com mais de um núcleo representam grandes avanços e resultam em ganhos de desempenho. Contudo, tanto a execução de várias threads quanto a execução de vários processos não requerem a presença de mais de uma unidade de execução. Ou seja, um computador com apenas um processador que apresente um núcleo pode executar vários processos ou threads em paralelo. No passado, muitos computadores rodavam sistemas operacionais que tinham recursos de execução concorrente, mesmo não tendo mais de uma unidade central de processamento.

2. Análise das afirmativas

I – Afirmativa correta.

JUSTIFICATIVA. Segundo DHAMDHERE (2006), "exceto pelo fato de que threads não apresentem recursos diretamente alocados para elas mesmas, threads e processos são análogos. Dessa forma, os estados das threads e de suas transições de estado são análogos aos estados de processos e de suas transições de estado".

II – Afirmativa incorreta.
JUSTIFICATIVA. Threads de diferentes processos não podem compartilhar memória.

III – Afirmativa incorreta.
JUSTIFICATIVA. Um único processador com um único núcleo pode executar um programa multithread.

IV – Afirmativa incorreta.
JUSTIFICATIVA. Threads não podem migrar de um processo para outro.

Alternativa correta: A.

3. Indicações bibliográficas

- DHAMDHERE, D. M. *Operating systems: a concept-based approach*. 2. ed. Noida: Tata McGraw-Hill Education, 2006.
- TANENBAUM, A. S.; BOS, H. *Sistemas operacionais modernos*. 4. ed. São Paulo: Pearson Education, 2014.

Questão 3

Questão 3.[3]

Os números de Fibonacci correspondem a uma sequência infinita na qual os dois primeiros termos são 0 e 1. Cada termo da sequência, à exceção dos dois primeiros, é igual à soma dos dois anteriores, conforme a relação de recorrência abaixo:

$$f_n = f_{n-1} + f_{n-2}$$

Desenvolva dois algoritmos, um iterativo e outro recursivo, que, dado um número natural n>0, retorna o n-ésimo termo da sequência de Fibonacci. Apresente as vantagens e desvantagens de cada algoritmo.

1. Introdução teórica

Recursão

Na matemática e na computação, podemos pensar na definição de uma sequência de várias formas. Uma delas é a definição de cada um dos termos da sequência por meio de uma fórmula (EPP, 2010). Como exemplo, considere, para todos os inteiros n>0, a sequência a_0, a_1, a_2:

$$a_n = \frac{(-1)^n}{(n+1)}$$

Podemos facilmente calcular essa fórmula para qualquer valor de n (inteiro), mesmo sem saber os valores anteriores da sequência. Uma das grandes vantagens dessa forma "genérica" de especificação é que, nesses casos, podemos afirmar que cada termo da sequência pode ser calculado em um número finito e fixo de passos.

Existem outras maneiras de se definir uma sequência, como, por exemplo, utilizar recursão (EPP, 2010). Nesse caso, definimos um termo da sequência em relação aos demais.

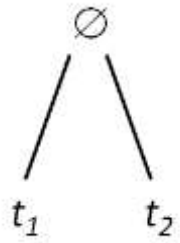
Por exemplo, podemos definir a sequência dos números Naturais conforme segue.

- 1 é um número natural.
- O sucessor de um número natural também é um número natural.

[3]Questão 3 – discursiva – Enade 2011.

A recursão também pode ser muito útil para definir estruturas de dados. Por exemplo, podemos definir uma árvore conforme segue (WIRTH, 1976).

- $\varnothing$ é uma árvore (chamada de árvore vazia)
- Se t_1 e t_2 são árvores, então a representação a seguir também é uma árvore.



A recursão pode ser utilizada para definir funções, como a função fatorial. Para qualquer n inteiro não negativo, temos o que segue.

- $0! = 1$
- Se $n > 0$, então $n! = n.(n-1)!$

Segundo Wirth (1976), em seu famoso livro sobre algoritmos: "o poder da recursão está na possibilidade de definir um número infinito de objetos por meio de uma declaração finita".

Para trabalharmos com recursão, devemos inicialmente ter uma equação, chamada de relação de recorrência, que define como um termo da sequência relaciona-se com os demais e as condições iniciais, com os primeiros termos da sequência (EPP, 2010).

2. Resolução da questão

No caso da sequência de Fibonacci, sabemos que:

(1) $f_n = f_{n-1} + f_{n-2}$

(2) $f_0 = 0$ e $f_1 = 1$

Então,

$f(2) = f(1) + f(0) = 0 + 1 = 1$
$f(3) = f(2) + f(1) = 1 + 1 = 2$
$f(4) = f(3) + f(2) = 2 + 1 = 3$
$f(5) = f(4) + f(3) = 3 + 2 = 5$
…

Observe que:

$f(4) = f(3) + f(2) = (f(2) + f(1)) + f(2) = ((f(1) + f(0)) + f(1)) + f(1) + f(0) = 1 + 0 + 1 + 1 + 0 = 3$

Podemos obter os valores de *f(n)* de forma recursiva, calculando os valores de *f(n-1)*, *f(n-2)*... até chegarmos aos valores inicias de *f(1)* e *f(0)*.

Segue abaixo uma implementação dos dois algoritmos em C, imprimindo-se na saída os primeiros 20 elementos de casa sequência:

```
#include<stdio.h>
/* Formato recursivo */
int fibrec(int n)
{
     if( n == 0 )
     {
          return 0;
     }
     if( n == 1 )
     {
          return 1;
     }
     return fibrec(n-1) + fibrec(n-2);
}
/* Formato iterativo */
int fibit(int n)
{
     int soma = 0;
     int antsoma = 0;
     int temp;
     int it = 0;
     for( it = 0; it<=n; it++)
     {
          if( it == 0 )
          {
               soma = 0;
          }
          if( it == 1 )
          {
               soma = 1;
          }
          if( it > 1 )
          {
               temp = soma;
               soma = soma + antsoma;
               antsoma = temp;
          }
     }
     return soma;
}
int main()
{
     int max = 20, it= 0;
     printf("Formato iterativo: \n");
     for( it = 0; it<max; it++)
```

```
    {
        printf("fibit %d: %d \n", it, fibit(it));
    }
    printf("Formato recursivo: \n");
    for( it = 0; it<max; it++)
    {
        printf("fibrec %d: %d \n", it, fibrec(it));
    }
    return 0;
}
```

A solução recursiva apresenta a vantagem de ser implementada diretamente a partir da definição do problema dado no enunciado. Mas a complexidade dessa solução é maior do que a da solução interativa.

Quanto ao tempo de cálculo, a solução recursiva apresenta tempo de cálculo exponencial. Já a implementação iterativa convencional demanda apenas tempo linear, $O(n)$ (SKIENA, 1998).

3. Indicações bibliográficas

- EPP, S. *Discrete mathematics with applications*. São Paulo: Cengage Learning, 2010.
- SKIENA, S. S. *The algorithm design manual*. New York: Springer-Verlag, 1998.
- WIRTH, N. *Algorithms + Data Structures = Programs*. Ontario: Pearson Education Canada, 1976.

Questão 4

Questão 4.[4]

Um dos problemas clássicos da computação científica é a multiplicação de matrizes. Assuma que foram declaradas e inicializadas três matrizes quadradas de ponto flutuante, a, b e c, cujos índices variam entre 0 e n - 1. O seguinte trecho de código pode ser usado para multiplicar matrizes de forma sequencial:

```
1. for [i = 0 to n - 1] {
2.       for [j = 0 to n - 1] {
3.                c[i, j] = 0.0;
4.                for [k = 0 to n - 1]
5.                         c[i, j] = c[i, j] + a[i, k] * b[k, j];
6.       }
7. }
```

O objetivo é paralelizar esse código para que o tempo de execução seja reduzido em uma máquina com múltiplos processadores e memória compartilhada. Suponha que o comando "co" seja usado para definição de comandos concorrentes, da seguinte forma: "co [i = 0 to n - 1] { x; y; z;}" cria n processos concorrentes, cada um executando sequencialmente uma instância dos comandos x, y, z contidos no bloco.

Avalie as seguintes afirmações sobre o problema:

I. Esse problema é exemplo do que se chama "embaraçosamente paralelo", porque pode ser decomposto em um conjunto de várias operações menores que podem ser executadas independentemente.

II. O programa produziria resultados corretos e em tempo menor do que o sequencial, trocando-se o "for" na linha 1 por um "co".

III. O programa produziria resultados corretos e em tempo menor do que o sequencial, trocando-se o "for" na linha 2 por um "co".

IV. O programa produziria resultados corretos e em tempo menor do que o sequencial, trocando-se ambos "for", nas linhas 1 e 2, por "co".

É correto o que se afirma em

A. I, II e III, apenas.

B. I, II e IV, apenas.

C. I, III e IV, apenas.

D. II, III e IV, apenas.

E. I, II, III, IV.

[4]Questão 27 – Enade 2011.

1. Introdução teórica

Computação paralela

Devido aos recentes avanços nas tecnologias de microprocessadores, muitos dos dispositivos computacionais utilizados na atualidade apresentam mais de um núcleo de processamento (dispositivos multi-core). Equipamentos como telefones celulares já utilizam esse tipo de tecnologia. A maioria dos servidores na atualidade já conta com múltiplos núcleos, bem como notebooks e desktops. A grande vantagem de se ter mais de um núcleo de processamento é a capacidade em se fazer tarefas de forma paralela, melhorando a experiência de uso (por exemplo, no caso de equipamentos pessoais) ou o desempenho (por exemplo, no caso de servidores).

Contudo, não basta apenas termos uma tecnologia poderosa à nossa disposição; é preciso que aproveitemos adequadamente essa tecnologia. Problemas computacionais diferentes têm potenciais diferentes para serem paralelizados, dependendo também da abordagem para a solução.

Quando um problema pode ser facilmente paralelizável, sendo possível dividi-lo em tarefas completamente independentes entre si, sem nenhuma necessidade de comunicação e ainda sem a necessidade de se impor uma ordem específica no seu cálculo, chamamos esse tipo de problema (ou implementação) de embaraçosamente paralela (MCCOOL et al. 2012). Nesse caso, o termo embaraçoso não tem uma conotação negativa: ele apenas quer dizer que o paralelismo, nessa situação, é absolutamente trivial, e não que o paralelismo seja ruim.

Essa abordagem também é chamada de mapa (no Inglês map), porque essencialmente uma função elementar é aplicada (mapeada) em um conjunto de dados gerando outro conjunto. Essa função não pode ter nenhum tipo de efeito colateral visto pelas outras execuções em paralelo, nem pode se comunicar com as outras execuções (MCCOOL et al. 2012).

2. Análise das afirmativas

I – Afirmativa correta.

JUSTIFICATIVA. Esse problema é tipo embaraçosamente paralelo, uma vez que cada elemento da matriz pode ser calculado independentemente dos demais.

II – Afirmativa correta.
JUSTIFICATIVA. O "for" pode ser calculado de forma paralela, já que cada elemento da matriz c não depende de outros elementos da mesma matriz c para ser calculado (apenas das matrizes a e b).

III – Afirmativa correta.
JUSTIFICATIVA. O "for" pode ser calculado de forma paralela, já que cada elemento da matriz c não depende de outros elementos da mesma matriz c para ser calculado (apenas das matrizes a e b).

IV – Afirmativa correta.
JUSTIFICATIVA. O "for" pode ser calculado de forma paralela, já que cada elemento da matriz c não depende de outros elementos da mesma matriz c para ser calculado (apenas das matrizes a e b).

Alternativa correta: E.

3. Indicação bibliográfica

- MCCOOL, M.; REINDERS, J.; ROBISON, A. *Structured parallel programming: Patterns for efficient computation*. São Paulo: Elsevier, 2012.

Questão 5

Questão 5.[5]

Considere que G é um grafo qualquer e que V e E são os conjuntos de vértices e de arestas de G, respectivamente. Considere também que $grau(v)$ é o grau de um vértice v pertencente ao conjunto V. Nesse contexto, analise as seguintes asserções.

I. Em G, a quantidade de vértices com grau ímpar é ímpar.

PORQUE

II. Para G, vale a identidade dada pela expressão

$$\sum_{v \in V} grau(v) = 2|E|$$

Acerca dessas asserções, assinale a opção correta.

A. As duas asserções são proposições verdadeiras, e a segunda justifica a primeira.
B. As duas asserções são proposições verdadeiras, e a segunda não justifica a primeira.
C. A primeira asserção é uma proposição verdadeira, e a segunda uma proposição falsa.
D. A primeira asserção é uma proposição falsa, e a segunda uma proposição verdadeira.
E. Tanto a primeira quanto a segundas asserções são proposições falsas.

1. Introdução teórica

Grafos simples ou não direcionados

Segundo Chartrand (2012), "um grafo é formado por conjunto finito não vazio V junto com uma relação irreflexiva, simétrica R em V". Isso significa que temos um conjunto de vértices V e cada vértice é relacionado aos demais por um conjunto de arestas.

Por exemplo, se u e v são vértices e estão conectados por uma aresta, podemos dizer que o par (u,v) representa essa aresta e que, portanto, (u,v) $\in$ R. Além disso, por R ser uma relação simétrica, se o par (u,v) pertence a R, o par (v,u) também pertence a R (CHARTRAND,2012). Adicionalmente, podemos chamar esse tipo de grafo de grafo não direcionado.

Um dos primeiros trabalhos da área de teoria dos grafos foi o famoso artigo do matemático Leonhard Euler sobre as sete pontes de Königsberg. Nesse texto, foi utilizada como motivação um pequeno jogo, que foi abstraído de forma a dar origem a um problema matemático. No artigo, foi provada a seguinte fórmula:

[5]Questão 20 – Enade 2011.

$$\sum_{v \in V} grau(v) = 2|E|$$

Na fórmula, também chamada de "lema do aperto de mão", V corresponde ao conjunto de vértices, $grau(v)$ corresponde ao grau de um determinado vértice v, E corresponde ao conjunto de arestas e $|E|$ corresponde ao número total de arestas. Observe que o somatório é feito sobre todos os vértices v pertencentes ao conjunto V, ou seja, todos os vértices do grafo. A fórmula pode ser lida como: "o somatório do grau de todos os vértices é igual ao dobro do número de arestas".

Além disso, e como decorrência dessa fórmula, foi provado que "a quantidade de vértices com grau ímpar é par". Uma forma de visualizar essa afirmação é lembrando que a somatória do grau de todos os vértices é par, como mostra a fórmula. Em uma soma de inteiros, e temos claramente uma soma de inteiros, a soma de números pares não altera se o resultado final for par ou ímpar.

Sabemos que a somatória total dos graus dos vértices corresponde à soma do grau de todos os vértices de grau ímpar mais a soma do grau de todos os vértices de grau par. Se tivermos uma quantidade par de vértices de grau par ou uma quantidade ímpar de vértices com grau par, a soma de ambos também é par (porque sendo o grau dos vértices par, a soma também será par) e, portanto, isso não afeta o resultado final da soma total.

Sabemos que a soma dos graus de todos os vértices (pares e ímpares) deve ser par e, se somarmos um número ímpar de vértices de grau ímpar, temos como resultado uma soma ímpar, o que é incorreto, uma vez que a soma total deve ser par. Logo, decorre que a quantidade de vértices com grau ímpar deve ser par para que a soma total também seja par. Na figura 1, temos um exemplo de grafo.

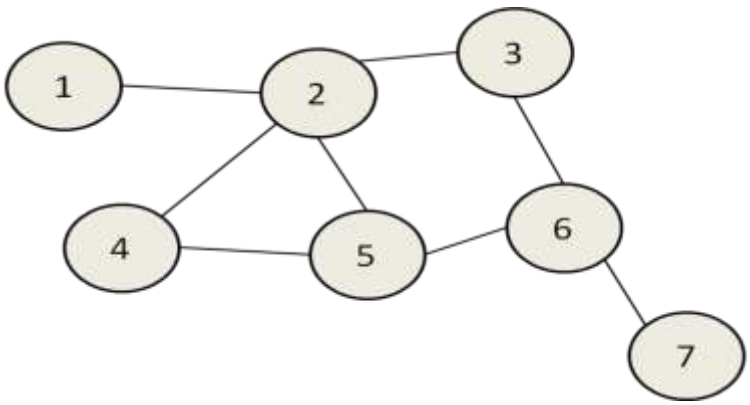


Figura 1. Exemplo de um grafo (simples ou não direcionado).

2. Resolução da questão

A primeira asserção é falsa, pois o correto seria o seguinte: "a quantidade de vértices com grau ímpar é par". A segunda asserção está correta e corresponde à famosa fórmula encontrada por Euler $\sum_{v \in V} grau(v) = 2|E|$.

Alternativa correta: D.

3. Indicação bibliográfica

- CHARTRAND, G. *Introductory graph theory*. Nova Iorque: Courier Dover Publications, 2012.

ÍNDICE REMISSIVO