Universidade Estadual de Campinas — UNICAMP

Centro Superior de Educação Tecnológica — CESET

ANÁLISE DE ALGORITMOS

ST067 — TÓPICOS ESPECIAIS EM INFORMÁTICA

Prof.: **Marco Antonio Garcia de Carvalho**

Fevereiro 2004
Campinas, SP - Brasil

Sumário

1 Introdução aos algoritmos computacionais

1.1 Definição de algoritmos

- Processo ou regras para realização de cálculos (Dicionário Oxford).
- São procedimentos que descrevem passo a passo a resolução de um problema.
- *A procedure for solving a mathematical problem in a finite number of steps that frequently involves a repetition of an operation*[1].

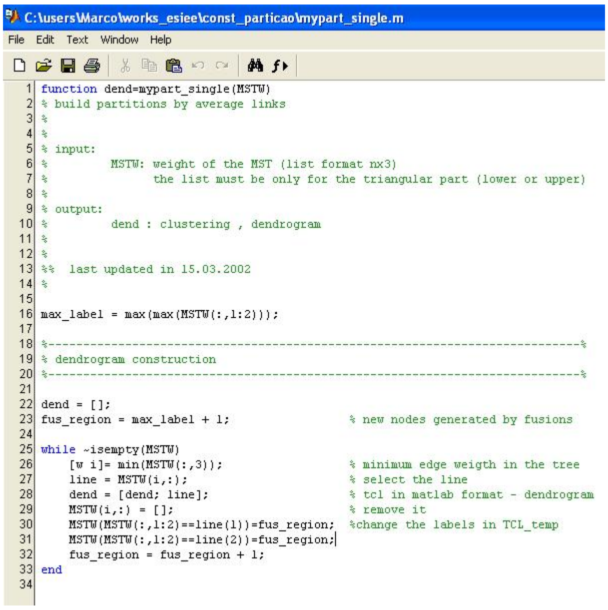
- Algoritmos são o cerne da computação.
- A execução de um algoritmo não deve envolver nenhuma decisão subjetiva.
- Algoritmos sistematizam a solução de um problema, que normalmente pode ser resolvido por diversas maneiras. Desta forma, podemos compará-los a partir de vários critérios (tempo e memória, por exemplo).
- Nem sempre a solução proposta por um algoritmo para resolver um problema é implementável na prática, como pode-se observar pelo exemplo da Tabela 1.

n	**Método de Cramer**	**Método de Gauss**
2	$22\mu s$	$50\mu s$
3	$102\mu s$	$159\mu s$
10	$1.19min$	$4.95ms$
20	*15225 séculos*	$38.63ms$
40	5×10^{33} *séculos*	$0.315s$

Tabela 1: Tabela com valores do cálculo da solução de um sistema de equações 20×20 através dos métodos de Cramer e Gauss (n é o número de variáveis) [6].

1.2 Representação de algoritmos

- Linguagem natural
  Em uma panela, aqueça 1 colher (sopa) de óleo em fogo médio, doure meia cebola média picada e 1 dente de alho amassado ...
- Pseudo-código
  Bastante utilizados nas disciplinas de lógica de programação.
  Usa palavras-chaves da linguagem natural.



C:\users\Marco\works_esiee\const_particao\mypart_single.m

File Edit Text Window Help

```
function dend=mypart_single(MSTW)
% build partitions by average links
%
%
% input:
%         MSTW: weight of the MST (list format nx3)
%               the list must be only for the triangular part (lower or upper)
%
% output:
%         dend : clustering , dendrogram
%
%
%%  last updated in 15.03.2002
%

max_label = max(max(MSTW(:,1:2)));

%--------------------------------------------------------------------------%
% dendrogram construction
%--------------------------------------------------------------------------%

dend = [];
fus_region = max_label + 1;                     % new nodes generated by fusions

while ~isempty(MSTW)
    [w i]= min(MSTW(:,3));                      % minimum edge weigth in the tree
    line = MSTW(i,:);                           % select the line
    dend = [dend; line];                        % tcl in matlab format - dendrogram
    MSTW(i,:) = [];                             % remove it
    MSTW(MSTW(:,1:2)==line(1))=fus_region;      %change the labels in TCL_temp
    MSTW(MSTW(:,1:2)==line(2))=fus_region;
    fus_region = fus_region + 1;
end
```

Figura 1: Código escrito em MatLab.

Identação ajuda na identificação dos blocos do algoritmo (estruturas de laço e decisão).
O Algoritmo 1 ilustra através de um algoritmo para determinação do valor máximo em uma lista.

Algoritmo 1 Determina o valor máximo de uma lista com n valores

---

Entrada: Tabela tab : vetor$[1 \dots n]$
Saída Mx = valor máximo em tab

$Mx \leftarrow tab[1]$
Para i de 2 até n **Faça**
  Se $Mx \leq tab[i]$ **Então**
    $Mx \leftarrow tab[i]$
  Fim Se
Fim Para
Retornar Mx

- Linguagem de programação
  Faz uso das mais diferentes linguagens existentes. A Figura 1 apresenta um exemplo de codificação de um algoritmo para cálculo de partições através do método de Ligação Simples usando a linguagem Matlab.

- Projetos de hardware

1.3 Análise de Algoritmos

- Análise de algoritmo mede a *eficiência* de um algoritmo, ou sua implementação em linguagem de programação, à medida em que o tamanho da entrada torna-se maior.
- A análise de algoritmos é importante, já que podem ser abstraídos aspectos como tipo de máquina e linguagem de programação utilizada.
- *Analysis of algorithms is quite important in computer programming , because there are usually several algorithms available for a particular application and we would like to know which is best* (Knuth, 1973).
- Pode existir mais de uma solução para o mesmo problema. É importante definir qual a melhor solução.
- Dois aspectos importantes na análise de um algoritmo: corretude e eficiência.

1.3.1 Corretude

- O algoritmo deve fornecer uma resposta correta para qualquer entrada.
- Garantir a corretude não é trivial. Usa-se bastante o procedimento empírico.
- Corretude de algoritmos não-recursivos
  - Analisar um laço por vez, começando pelo laço mais interno (se houver aninhamento).
  - Para cada laço, determinar seu *invariante*, que é verdadeiro para qualquer iteração do laço.
  - Usar o invariante para provar que o algoritmo termina e que produz o resultado correto.
- Corretude de algoritmos recursivos
  - Provar que as chamadas recursivas são etapas do problema sem recursão infinita.
  - Provar que a chamada recursiva executam corretamente.
- Uma técnica utilizada para provar a corretude é a indução matemática.

Indução matemática

- Indução matemática é usada como uma técnica de prova primária, mas eficaz.
- Seja T um teorema que se deseja provar e suponha que T possue um parâmetro n, positivo e inteiro. Ao invés de provar que T é válida para qualquer valor de n, pode-se provar as seguintes condições:

1. Provar que é válida para $n = 1$.
2. Provar que $\forall n > 1$, se a propriedade é válida para n, então ela é válida para $n+1$.
Existem variações desta prova. Por exemplo, provar que $\forall n > 1$, se a propriedade é válida para $n - 1$, então ela é válida para n.

- Assumir que T é válido para $n-1$, por exemplo, é uma hipótese chamada de *hipótese da indução*. Portanto, o princípio da indução pode ser assim enunciado:
Se uma proposição P, com um parâmetro n, é verdadeira para n=1, e se para todo $n > 1$, dizer que P é verdadeira para n-1 implica que tambem é verdade para n, então P é verdadeira para todos os números naturais.

- Exemplos de indução
Problema 1: Provar que $\forall n \in \mathbb{N}, 1 + 2 + \cdots + n = n(n + 1)/2$.
Problema 2: Provar que $\forall x, n \in \mathbb{N}, x^n - 1$ é divisível por $x - 1$.
Problema 3: Provar que uma árvore binária completa com k níveis tem exatamente $2^k - 1$ nós.

1.3.2 Eficiência

- Mede o desempenho de um algoritmo de acordo com determinado critério.

- A eficiência de um algoritmo é influenciada pelo tamanho e configuração da entrada.

- As análises, portanto, são realizadas levando-se em consideração o pior caso, o caso médio e o melhor caso (raramente acontece). Veja os exemplos a seguir, com os Algoritmos 2 e 3. Avalie os algoritmos para os vetores $A = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]$, $B = [1, 2, 3, 4, 5, 6]$ e $C = [6, 5, 4, 3, 2, 1]$.

Algoritmo 2 Análise de Eficiência Ex.1

Entrada: Tabela T : vetor$[1 \ldots n]$

Para i de 2 até n **Faça**
  $x \leftarrow T(i)$
  $j \leftarrow i - 1$
  Enquanto $j > 0$ e $x < T(j)$ **Faça**
    $T(j + 1) \leftarrow T(j)$
    $j \leftarrow j - 1$
  Fim Enquanto
  $T(j + 1) \leftarrow x$
Fim Para

- É natural questionar sobre qual unidade utilizar para representar a eficiência de um algoritmo. Infelizmente não existe um computador padrão que possa ser usado como referência.

Algoritmo 3 Análise de Eficiência Ex.2

Entrada: Tabela T : vetor$[1 \dots n]$

Para i de 1 até $n-1$ **Faça**
  $minj \leftarrow i$
  $minx \leftarrow T(i)$
  Para j de $i+1$ até n **Faça**
    Se $T(j) < minx$ **Então**
      $minj \leftarrow j$
      $minx \leftarrow T(j)$
    Fim Se
  Fim Para
  $T(minj) \leftarrow T(i)$
  $T(i) \leftarrow minx$
Fim Para

- *Princípio da invariância* – duas diferentes implementações do mesmo algoritmo irão diferenciar em eficiência de acordo com uma constante positiva.
  Algo 1 $\rightarrow t_1(n)$ *segundos*
  Algo 2 $\rightarrow t_2(n)$ *segundos*, para uma instância de tamanho n.
  $\Rightarrow$ existe uma constante c tal que $t_1(n) \leq c \cdot t_2(n)$

- A eficiência de um algoritmo é expressa pela notação *da ordem de t(n)*, para uma dada função t, se existe uma constante positiva c e uma implementação de um algoritmo capaz de resolver cada instância do problema limitado por $c \cdot t(n)$ segundos (minutos, anos). Esse conceito é conhecido como *notação assintótica*.
  Ex.: Se um algoritmo tem uma eficiência da ordem de n, dado que n é o tamanho da instância a ser resolvida, então esse algoritmo é dito *linear*.

- **Exemplo 3** – Mostre um problema real no qual apenas a melhor solução servirá. Em seguida, apresente um problema em que baste uma solução que seja aproximadamente a melhor.

- **Exemplo 4** – Além da velocidade, que outras medidas de eficiência poderiam ser usadas em uma configuração real?

1.4 Por que estudar algoritmos?

- Evitar *reinventar* algoritmos. Para a maioria dos problemas conhecidos já existem bons algoritmos, tendo sido feitas análises de corretude e eficiência.

- Nem sempre existe uma solução pronta para o seu problema.

- Fonte de idéias na solução de problemas.

- Ajudam a entender ferramentas que utilizamos no cotidiano.

1.5 Exemplos / Exercícios

- **Exemplo 5** – Considere a seqüência de números abaixo. O objetivo é apagar a menor quantidade de números possíveis de tal forma que os números restantes apareçam em ordem crescente (por exemplo, apagando todos os números, com exceção dos dois primeiros, a seqüência ficaria em ordem crescente).
  9 44 32 12 7 42 34 92 35 37 41 8 20 27 83 64 61 28 39 93 29 17 13 14 55 21 66 72 23 73 99 1 2 88 77 3 65 83 84 62 5 11 74 68 76 78 67 75 69 70 22 71 24 25 26

- **Exemplo 6** – Suponha que no Brasil existem 5 tipos de moedas: 15, 23, 29, 41 e 67 centavos. Determine uma combinação dessas moedas para o pagamento de uma conta de R$ 18,08.

- **Exemplo 7** – Calcule o valor de 2^{64}. Tente minimizar o número de multiplicações.

1.6 Revisão matemática

- Muitas vezes usaremos conceitos matemáticos vistos em outras disciplinas no cálculo da complexidade de algoritmos. Coloco alguns a seguir.

- Soma dos termos de uma PA:

$$\sum_{i=0}^{n} a_i = \frac{(a_0 + a_n)(n+1)}{2} \tag{1}$$

- Soma dos termos de uma PG:

$$\sum_{i=0}^{n-1} aq^i = \frac{a(q^n - 1)}{q - 1} \tag{2}$$

- Outras somatórias

$$\sum_{i=0}^{n} i^2 = \frac{1}{6} n (n+1)(2n+1) \tag{3}$$

$$\sum_{i=0}^{n} i^3 = \frac{n^2 (n+1)^2}{4} \tag{4}$$

- Propriedades dos logaritmos.
  - Produto: $log_a (n \cdot m) = log_a n + log_a m$
  - Potência: $log_a (n^m) = m \cdot log_a n$
  - Troca de base: $log_a n = log_b n \cdot log_a b$

2 Complexidade de Algoritmos

2.1 Introdução

- A expressão *quantidade de trabalho requerido* também é chamada complexidade do algoritmo.
- Critérios de medidas de complexidade: espaço em memória, tempo de execução. Nesse texto, será dada ênfase à complexidade de tempo.
- Mesmo para entradas do mesmo tamanho, a quantidade de operações efetuadas pelo algoritmo pode depender de uma entrada em particular.

2.2 Análise assintótica

- Ao ver uma expressão como $n+10$ ou n^2+1, a maioria das pessoas pensa automaticamente em valores pequenos de n, tipicamente valores próximos de zero.
- A análise de algoritmos trabalha com grandes valores de n, onde n representa o tamanho da entrada do algoritmo. Para grandes valores de n, as funções
  n^2, $3/2n^2$, $9999n^2$, $n^2/1000$, n^2+100n etc
  crescem todas com a mesma velocidade e, portanto, são *equivalentes*.
- Este tipo de estudo matemático é chamado de *assintótico*.
- **Ordem O**
  - Quando dizemos que $f(n) = O(g(n))$ garantimos que $g(n)$ é um limite superior sobre $f(n)$(ver Figura 2). Isso é escrito na forma:
  $f(n) \leq c \cdot g(n)$
  onde c é uma constante positiva.

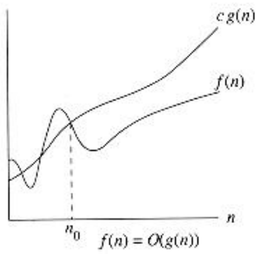


Figura 2: Ordem assintótica superior.

- *Exemplo:* Suponha que $f(n) = (3/2)\,n^2 + (7/2)\,n - 4$ e que $g(n) = n^2$. A Tabela 2 abaixo sugere que $f(n) \leq 2 \cdot g(n)$ para $n \geq 6$ e, portanto, $f(n) = O(g(n))$.

n	$f(n)$	$g(n)$
0	-4	0
1	1	1
2	9	4
3	20	9
4	34	16
5	51	25
6	71	36
7	94	49
8	120	64

Tabela 2: Exemplo funções assintóticas.

- **Ordem Ω**
  - Quando dizemos que $f(n) = \Omega(g(n))$, garantimos que $g(n)$ é um limite inferior sobre $f(n)$ (ver Figura 3). Isso é escrito na forma:
  $f(n) \geq c \cdot g(n)$
  onde c é uma constante positiva.

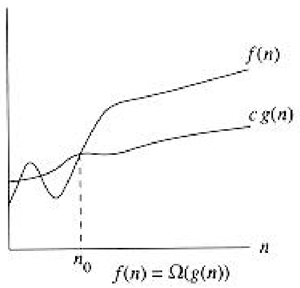


Figura 3: Ordem assintótica inferior.

- A análise assintótica é baseada nessas definições e estabelece uma ordem relativa entre funções. De fato, o que é observado e comparado são as taxas de crescimento das funções. A Tabela 3 apresenta as principais taxas de crescimento de algoritmos presentes na literatura.

- Lembrar que
  $O(1) < O(logn) < O(n) < O(n^2) < O(n^3) < O(2^n)$

2.3 Complexidade de tempo

- Para determinar o tempo de execução de um algoritmo temos que descobrir a forma geral da curva que caracteriza seu tempo de execução em função do tamanho do

Função	Nome
1	*constante*
$logn$	*logarítmico*
n	*linear*
n^2	*quadrático*
n^3	*cúbico*
2^n	*exponencial*

Tabela 3: Taxas de crescimento mais comuns.

problema.

- Para simplificar, não adota-se a existência de uma unidade de tempo em particular.
- Considera-se somente a compexidade de tempo segundo a notação $O()$, ou *Big-Oh*.
- Informalmente, podemos determinar a ordem de complexidade de uma determinada função $f(n)$ através das seguintes etapas:
  1. Separar $f(n)$ em duas partes: termo dominante e termos de ordem inferior.
  2. Ignorar os termos de ordem inferior.
  3. Ignorar as constantes de proporcionalidade.
- Para ilustrar, vejamos o exemplo de um algoritmo cujo tempo de execução é caracterizado pela função $f(n) = an^2 + bn + c$. Qual seria sua complexidade?
  1. O termo an^2 é dominante (maior ordem) sobre os demais. Os termos de ordem inferior podem ser desprezados.
  2. A constante de proporcionalidade no termo an^2 pode ser desprezada.
  3. Conclui-se que $f(n) = O(n^2)$, isto é, a complexidade do algoritmo é de ordem quadrática.
- Analisemos o impacto do aumento de velocidade em algoritmos computacionais.
  - Suponha que uma máquina resolve problemas de tamanho máximo x_1. Em um computador dez vezes mais rápido, o mesmo algoritmo resolverá um problema de tamanho 10 vezes maior, ou seja, $10x_1$.
  - Seja agora um algoritmo com tempo quadrático (tempo proporcional a n^2 para entradas de tamanho n). Suponha que o problema x_2 seja resolvível em um tempo t na máquina mais lenta, isto é, $k \cdot x_2^2 = t$. Para uma máquina 10 vezes mais rápida, um tempo $10t$. O tamanho do problema resolvível será y e dado por:
  $ky^2 = 10t$
  $ky^2 = 10k \cdot x_2^2$

$y^2 = 10x_2^2$
$y = \sqrt{10}x_2$
- Seja uma algoritmo exponencial (tempo 2^n para uma entrada de tamanho n). Se x_4 é o tamanho máximo de um problema resolvível num tempo t na máquina mais lenta e y na máquina mais rápida, tem-se:
$2^{x_4} = t$
$2^y = 10t$
$2^y = 10 \cdot 2^{x_4}$
$y = log_2 10 + x_4$
$y = x_4 + 3.3$
- Outros valores são dados na Tabela 4.
- **Exercício:** Verifique os valores da Tabela 4 para $log_2 n$, n^3 e 3^n.

Complexidade de tempo	**Tamanho máximo máquina lenta**	**Tamanho máximo máquina rápida**
n	x_1	$10x_1$
n^2	x_2	$3.16x_2$
n^3	x_3	$2.15x_3$
2^n	x_4	$x_4 + 3.3$
$log_2 n$	x_5	x_5^{10}

Tabela 4: Tabela complexidade do algoritmo × tamanho máximo do problema resolvível[6].

- **Exemplo 1** — Considere 2 algoritmos A e B com complexidades $8n^2$ e n^3. Qual o maior valor de n para qual o algoritmo B é mais eficiente que o algoritmo A?

- **Exemplo 2** — Um algoritmo tem complexidade $2n^2$. Em um certo computador, num tempo t, o algoritmo resolve um problema de tamanho 25. Imagine agora que você tem disponível um computador 100 vezes mais rápido. Qual o tamanho máximo de problema que o mesmo algoritmo resolve usando o computador mais rápido?

- **Exemplo 3** — Considere o mesmo problema anterior para um algoritmo de complexidade 2^n.

- **Exemplo 4** — Suponha que uma empresa utiliza um algoritmo de complexidade n^2 que resolve um problema de tamanho x em um tempo t na máquina disponível. Suponha agora que o tamanho do problema a ser resolvido aumentou em 20%. A empresa pretende trocar a máquina por uma mais rápida para que o tempo de resposta não se altere. Qual o percentual de melhoria no tempo de execução das operações básicas é necessário para atingir sua meta?

2.4 Análise da complexidade de algoritmos

- Podemos definir qual algoritmo é preferível para resolver determinado problema de duas formas: empírica (implementar o algoritmo e testá-lo para diferentes instâncias) e teórica (determinar matematicamente a quantidade de operações realizadas pelos algoritmos como função do tamanho da instância considerada).

- O tamanho de uma instância n corresponde formalmente ao número de bits para representar a instância [4]. Contudo, para tornar as análises mais claras (e simples de serem efetuadas) a palavra tamanho pode indicar o número de componentes de uma instância ou o seu valor numérico.

- Para analisar algoritmos através da visão assintótica é necessário definir um modelo de computação.

- Consideraremos, portanto, que as instruções são executadas sequencialmente e que o conjunto de instruções simples (adição, comparação etc) são executadas em uma unidade de tempo.

- São definidas a seguir algumas regrinhas básicas.

  - *Laços:* o tempo de execução de um laço é no máximo o tempo de execução das instruções dentro do laço (incluindo os testes) vezes o número de iterações.
  - *Aninhamento de laços:* Analisar primeiro os mais internos. O tempo total de execução de uma instrução dentro de um grupo de laços aninhados é igual ao tempo de execução da instrução multiplicado pelo produto dos tamanhos de todos os laços.
  - *Instruções consecutivas:* Apenas efetuar a soma.
  - *if/else:* o tempo de execução de uma instrução *if/else* nunca é maior do que o tempo de execução do teste mais o maior dos tempos de execução de S_1 e S_2, onde S_1 e S_2 representam as instruções do *then* e *else*, respectivamente.
  - *Chamada de funções:* A análise é feita como no caso dos laços aninhados. Para calcular a complexidade de um programa com várias funções, determina-se primeiro a complexidade de cada uma das funções. Desta forma, cada uma das funções é vista na análise geral como uma instrução com a complexidade que foi calculada.
  - *Recursão:* É a parte mais difícil!!! Em muitos casos, pode-se fazer a linearização através da substituição da chamada recursiva por alguns laços aninhados, por exemplo. Entretanto, existem algoritmos que não admitem este artifício. Neste caso, é necessário usar uma relação de recorrência que tem que ser resolvida.

- Pode-se definir a eficiência de um algoritmo de maneira *exata* ou *aproximada* (a forma adotada!).

- Para medir a quantidade de trabalho é escolhida uma operação importante dentro do algoritmo, denominada de operação fundamental ou elementar[6]. Em seguida, é contado o número de execuções dessa operação na execução do algoritmo. Por exemplo, no caso de uma busca em uma lista, a operação fundamental pode ser a comparação entre dois elementos (a chave e o elemento da lista).

- O uso da operação fundamental pode evitar a análise linha-a-linha de todo o algoritmo.

- Às vezes, é necessário o uso de mais de uma operação fundamental. Pode-se considerar que uma operação fundamental é executada em uma unidade de tempo.

- A complexidade de tempo de um algoritmo é apenas um fato sobre ele. Um algoritmo assintoticamente mais barato pode ser mais difícil de ser implementado do que um outro algoritmo com complexidade de tempo maior.

- Lembre-se: a superioridade assintótica de um algoritmo só é evidente quando os problemas são suficientemente grandes.

Exemplos de análises de algoritmos simples

- **Exemplo 1** — Analise os 3 algoritmos da Questão 5 da primeira lista de exercícios.

- **Exemplo 2** — Calcule a complexidade do algoritmo para calcular x^n (Algoritmo 4).

Algoritmo 4 Determina $y = x^n$

$i \leftarrow n$
$y \leftarrow 1$
Enquanto $i > 0$ **Faça**
  $y \leftarrow y * x$
  $i \leftarrow i - 1$
Fim Enquanto
Retornar y

- **Exemplo 3** — Analise o algoritmo de determinação do máximo em uma lista (Algoritmo 5), mostrado a seguir.

- **Exemplo 4** — Analise o algoritmo de busca sequencial de um número x em uma lista de n elementos (Algoritmo 6).

Algoritmo 5 Determina o valor máximo de uma lista com n valores

Entrada: Tabela tab : vetor$[1 \dots n]$
Saída Mx = valor máximo em tab

$Mx \leftarrow tab[1]$
Para i de 2 até n **Faça**
  Se $Mx \leq tab[i]$ **Então**
    $Mx \leftarrow tab[i]$
  Fim Se
Fim Para
Retornar Mx

Algoritmo 6 Busca seqüêncial em uma lista com n valores

Entrada: x: chave; lista tab : vetor$[1 \dots n]$;
Saída: pos: posição do elemento na lista tab

$i \leftarrow 0$
$achou \leftarrow F$
Repita
  $i = i + 1$
  Se $tab[i] = x$ **Então**
    $achou = V$
  Fim Se
Até $achou = V$ ou $i = n$
Se $achou$ **Então**
  $pos \leftarrow i$
Fim Se
Retornar pos

Exemplos de análises de algoritmos recursivos

- **Exemplo 1** — Algoritmo do cálculo do fatorial de um número n. Comparar as versões recursiva e não-recursiva.

- **Exemplo 2** — Algoritmo da Torre de Hanoi.

3 Algoritmos de ordenação

3.1 Introdução

- Ordenar é um problema importante em diversas tarefas na computação.
- Algoritmos de ordenação por seleção e inserção são excelentes quando n é pequeno, mas para grandes valores de n outros algoritmos de ordenação são mais eficientes(mergesort, quicksort).
- Uma comparação simples entre tempo de execução de algoritmos de ordenação é dada a seguir (usa-se implementações em Pascal [4]).
  . Quando n é pequeno, a diferença entre eficiência quase não é perceptível.
  . *Quicksort* é aproximadamente 2 vezes mais rápido que o *insertion sort* na ordenação de 50 elementos e três vezes mais rápido na ordenação de 100 elementos.
  . Para ordenar 1000 elementos, *insertion sort* leva cerca de 3 segundos, enquanto que o *quicksort* requer menos de 1/15 segundo. A ineficiência de usar o *insertion sort* torna-se clara para n da ordem de 5000 elementos: cerca de 1 minuto e meio; enquanto *quicksort* leva um pouco mais de 1 segundo.
- Avalie os algoritmos de ordenação por seleção, inserção e quicksort para o vetor $A = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]$ (faça um acompanhamento das variáveis e do vetor A).

3.2 Ordenação por seleção

- A maior parte do tempo de execução é gasto na realização do laço interno.
- O algoritmo é da ordem de $O(n^2)$.

3.3 Ordenação por inserção

- O tempo que o algoritmo gasta para ordenar n elementos depende da configuração inicial dos elementos.
- O número de comparações realizadas entre elementos é uma boa medida da complexidade (como para a maioria dos algoritmos de ordenação).
- Uma ordem de $O(n)$ é suficiente para a maioria das instâncias, embora para o pior caso seja de $\Omega(n^2)$.

Algoritmo 7 Algoritmo *Selection sort*

Entrada: Tabela T : vetor$[1 \ldots n]$

Para i de 1 até $n-1$ **Faça**
  $minj \leftarrow i$
  $minx \leftarrow T(i)$
  Para j de $i+1$ até n **Faça**
    Se $T(j) < minx$ **Então**
      $minj \leftarrow j$
      $minx \leftarrow T(j)$
    Fim Se
  Fim Para
  $T(minj) \leftarrow T(i)$
  $T(i) \leftarrow minx$
Fim Para

Algoritmo 8 Algoritmo *Insertion sort*

Entrada: Tabela T : vetor$[1 \ldots n]$

Para i de 2 até n **Faça**
  $x \leftarrow T(i)$
  $j \leftarrow i-1$
  Enquanto $j > 0$ e $x < T(j)$ **Faça**
    $T(j+1) \leftarrow T(j)$
    $j \leftarrow j-1$
  Fim Enquanto
  $T(j+1) \leftarrow x$
Fim Para

3.4 Quicksort

- É um dos algoritmos de ordenação mais usados na prática devido à sua eficiência.
- Faz parte das técnicas denominadas de dividir e conquistar, que consistem em particionar o problema em instâncias menores, resolvendo-os sucessiva e independentemente, e ao final combinando-as para obter a solução do problema original.
- O algoritmo (ver Algoritmo 9) inicia com a escolha de um dos elementos do vetor a ser ordenado como pivô. Em seguida, o vetor é dividido em duas partes: elementos maiores que o pivô são posicionados à direita e elementos menores que o pivô, à esquerda.
- Em seguida, as duas partes do vetor são ordenadas independentemente através de chamadas recursivas ao algoritmo. O resultado é um vetor completamente ordenado,

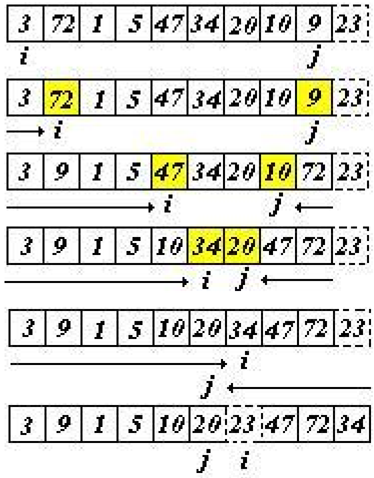


Figura 4: Exemplo para a função *Particiona*. O último elemento foi o escolhido como pivô.

obtido sem a necessidade de outras etapas, como intercalações e fusões.

Algoritmo 9 Algoritmo $Quicksort(T\,[i\ldots j])$

Se $j - i$ *pequeno* **Então**
  $insertionsort(T\,[i\ldots j])$
Se Não
  $Particiona(T\,[i\ldots j]\,,l)$
  $Quicksort(T\,[i\ldots l-1])$
  $Quicksort(T\,[l+1\ldots j])$
Fim Se

- A função *Particiona* permuta os elementos do vetor $(T\,[i\ldots j])$ de modo que ao fim de sua execução $i \leq l \leq j$, os elementos de $(T\,[i\ldots l-1])$ são menores que um pivô p e os elementos $(T\,[l+1\ldots j])$ são maiores que p. O pivô p é normalmente escolhido como o primeiro ou último elemento no vetor, como mostrado no exemplo a seguir (Figura 4).

- Na Figura 4:

  1. Percorre-se da esquerda para direita até achar um elemento *T(i)* maior do que p.
  2. Percorre-se também da direita para esquerda até achar um elemento *T(j)* menor do que p.
  3. Troca-se esses dois elementos.

4. Continua-se percorrendo e trocando os elementos até $j \leq i$.
5. Por fim, troca-se p com *T(i)*.

- O algoritmo é da ordem de $O(n^2)$, para o pior caso.

3.5 Exemplos / Exercícios

- **Exemplo 1** – Efetue uma análise do algoritmo de particionamento do *quicksort* para o vetor $A = \{105, 5, 90, 3, 40, 13, 30, 10, 21, 25\}$.
  Escreva TODOS os passos do desenvolvimento para uma primeira chamada à essa função. Indique quais os dois vetores que serão gerados pela função de particionamento.

- **Exemplo 2** – A ordenação BOLHA é uma das formas mais conhecidas e simples de se ordenar elementos. Porém, é uma das piores ordenações já concebidas [Schildt, *C Completo e total*, 1996]. Dado o Algoritmo 10 abaixo, faça:
  (a) Acompahe as variáveis *i, j* e t, além do vetor V. Assuma que inicialmente $V = \{20, 30, 15, 8, 3\}$. Explique em poucas palavras como funciona este famoso algoritmo;
  (b) Faça uma análise aproximada da complexidade deste algoritmo. Qual sua ordem de complexidade?

Algoritmo 10 Algoritmo Bolha

Entrada: Vetor V, número de elementos n de V
Saída: Vetor V ordenado
Para i de 2 até n **Faça**
  Para j de n até i **Faça**
    Se $V(j-1) > V(j)$ **Então**
      $t \leftarrow V(j-1)$
      $V(j-1) \leftarrow V(j)$
      $V(j) \leftarrow t$
    Fim Se
  Fim Para
Fim Para
Retornar V

- **Exemplo 3** – Utilizando-se do mesmo vetor V do exemplo anterior, apresente o acompanhamento das variáveis (teste de mesa) dos algoritmos de ordenação por seleção e inserção.

4 Algoritmos em grafos

4.1 Introdução à teoria dos grafos

- Bons livros em teoria dos grafos são as referências [10] e [9].
- **Grafo** – Um grafo G é definido por $G = (V, E)$, sendo que V representa o conjunto de nós e E, o conjunto de arestas (i, j), onde $i, j \in V$. Dois nós i, j são vizinhos, denotado por $i \sim j$, se eles estão conectados por uma aresta. A Figura 5 mostra um exemplo de grafo, consistindo dos conjuntos $V = \{a, b, c, d, e\}$ e $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$.

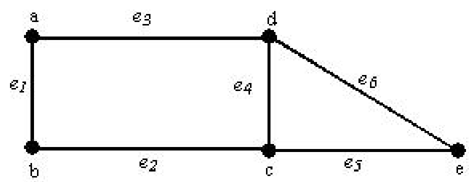


Figura 5: Exemplo de um Grafo.

- **Caminho** – Um caminho entre i_1 e i_n é a lista $(i_1, i_2, \ldots, i_{n-1}, i_n)$, onde $i_k \sim i_{k+1}$, $k = 1, 2, \ldots, n-1$. Dois nós i, j são conectados se existe ao menos um caminho entre i e j. Um caminho onde $i_1 = i_n$ é chamado de *ciclo*. Um exemplo de caminho entre os nós a e e do grafo na Figura 5 é a lista $(e3, e6)$
- **Grafo conexo** – Um grafo G é dito conexo se existe um caminho para qualquer par de nós (i, j) pertencente à G.
- **Grafo direcional** – Um grafo é dito direcional, ou dígrafro, quando o sentido das arestas é importante e indicado através de uma seta, como ilustrado na Figura 6.

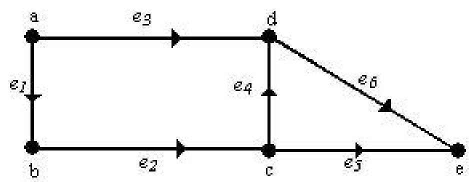


Figura 6: Exemplo de um dígrafo.

- **Grafo Ponderado**– Em um grafo ponderado, um peso ou conjunto de pesos é associado a cada aresta, representado da forma $w(i, j)$. Já em um grafo com atributos A, definido por $G = (V, E, A)$, os valores são associados aos nós de G. O menor custo entre dois nós i, j, $C(i, j)$, é definido como o menor custo entre os custos de todos os caminhos existentes entre i e j.

- **Subgrafo** – Um subgrafo de um grafo G é o grafo H tal que $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Podemos representar na forma $H \subseteq G$ e dizer que G contém H. Os grafos da Figura 7(b),(c) são subgrafos do grafo mostrado na Figura 7(a).

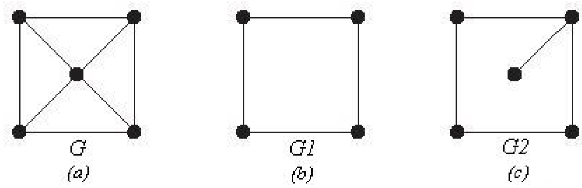


Figura 7: Um grafo G em (a) com dois subgrafos $G1$ (b) e $G2$ (c).

4.2 Representação de grafos

- **Matriz de adjacência** – Uma das formas mais utilizadas para representar grafos é via a matriz de adjacência. Seja $A = [a_{ij}]$ uma matriz $n \times n$, onde n é o numero de nós de um grafo $G = (V, E)$ qualquer. A matriz de adjacência A é construída da seguinte forma:

$$A(i,j) = \begin{cases} 1 & \text{se } i \sim j \\ 0 & \text{caso contrário} \end{cases}$$

A Figura 8 ilustra o conceito de matriz de adjacência para um grafo simples.

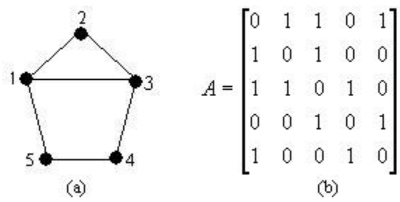


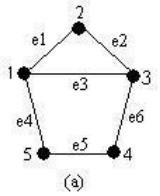
Figura 8: (a) Um grafo G e (b) sua matriz de adjacência.

Quando o grafo possui peso associado aos arcos, a representação só fica completa quando também se indica a sua matriz de pesos, construída de maneira semelhante à matriz adjacência (troca-se o valor do peso pelos 1´s). Para dígrafos, é preciso observar o sentido do caminho entre os nós.

- **Matriz de incidência** – A matriz de incidência $B = [b_{ij}]$ de um grafo $G = (V, E)$, com $V = (v_1, v_2, \ldots, v_n)$ e $E = (e_1, e_2, \ldots, e_m)$, é definida da seguinte forma:

$$B(i,j) = \begin{cases} 1 & \text{se } v_i \in e_j \\ 0 & \text{caso contrário} \end{cases}$$

A matriz de incidência do grafo na Figura 8(a) é dada pela Figura 9.



$$
B = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}
$$

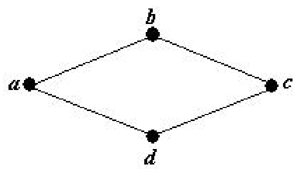
(b)

Figura 9: Matriz de incidência do grafo na Figura 8(a). Observe que as arestas foram rotuladas. As linhas da matriz correspondem aos nós, e as colunas correspondem às arestas.

Se G é um dígrafo, então $b_{ij} = +1$ se v_i está no início da seta e $b_{ij} = -1$, caso v_i esteja na cabeça da seta.

4.3 Matriz distância em um grafo

- Em diversas situações, deseja-se calcular a distância, ou o comprimento do caminho entre dois nós. A Figura 10 apresenta um grafo e sua respectiva matriz distância. A cada arco é associado um peso de valor 1.



$$
dist = \begin{array}{c} a \\ b \\ c \\ d \end{array} \begin{bmatrix} 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{bmatrix}
$$

Figura 10: Um grafo e sua matriz distância.

4.3.1 Algoritmo Dijkstra

- Como se observa na Figura 10, a matriz distância é construída com base na menor distância entre dois vértices. O Algoritmo 11 determina o comprimento do menor caminho entre dois nós a e z, em um grafo conectado. Esse algoritmo foi desenvolvido por Edsger W. Dijkstra, na metade do século passado.

- Um exemplo da implementação desse algoritmo é dado na Figura 11. A Figura 11(a) apresenta um grafo G com pesos nos arcos. Deseja-se encontrar a menor distância entre os nós a e z. As Figuras 11(b)-11(f) mostram os passos realizados na implementação do Algoritmo 11 até se alcançar o resultado desejado.

- Seja o algoritmo de Dijkstra aplicado a um grafo tendo n nós e a arestas. Para uma implementação simples, a complexidade é $O\left(n^2\right)$[4]. Outras implementações

Algoritmo 11 Algoritmo Dijkstra — Determina o menor caminho entre dois nós

Entrada: Grafo conectado com pesos nos arcos (matriz w), nós a e z.
Saída: L(z) - comprimento do menor caminho entre a e z.

$L(a) \leftarrow 0$
Para todo nó $x \neq a$ **Faça**
  $L(x) \leftarrow \infty$
Fim Para
$T \leftarrow$ conjunto de todos os nós cuja menor distância até a ainda não foi calculada.
Enquanto $z \in T$ **Faça**
  Escolha $v \in T$ com menor $L(v)$
  $T = T - \{v\}$
  Para $x \in T$ vizinho a v **Faça**
    $L(x) \leftarrow min\left\{L\left(x\right), L\left(v\right) + w\left(v, x\right)\right\}$
  Fim Para
Fim Enquanto

podem ter complexidade da ordem $O\left(\left(a+n\right)\log n\right)$. Uma implementação simples é preferível quando o grafo é denso.

4.4 Árvore geradora mínima

- **Árvore** – Um grafo conexo sem ciclos é chamado de árvore (exemplo na Figura 12).

  Pode-se designar um nó para ser a raiz da árvore, o que demonstra uma relação lógica entre os nós. Essas árvores são ditas hierárquicas e a distância entre cada nó e a raiz é denominada de *nível*. Em uma árvore hierárquica os nós podem ser rotulados de acordo com a denominação de uma árvore genealógica: filhos, pais e ancestrais, no sentido literal das palavras. Uma árvore hierárquica onde cada nó dá origem a dois outros nós de nível inferior é chamada de *árvore binária*.

- **Árvore geradora** – É uma árvore T, subgrafo de G, que contém todos os nós de G. Uma árvore geradora cuja a soma dos pesos de seus arcos seja menor do que em qualquer outra situação é chamada de *árvore geradora mínima*.

4.4.1 Algoritmo de Kruskal

- Determina uma árvore geradora mínima de um grafo $G = (V, E)$.

- Descreveremos de maneira informal o algoritmo de Kruskal:

  1. O conjunto T de arestas está inicialmente vazio. Conforme o andamento do algoritmo, arestas vão sendo adicionadas.

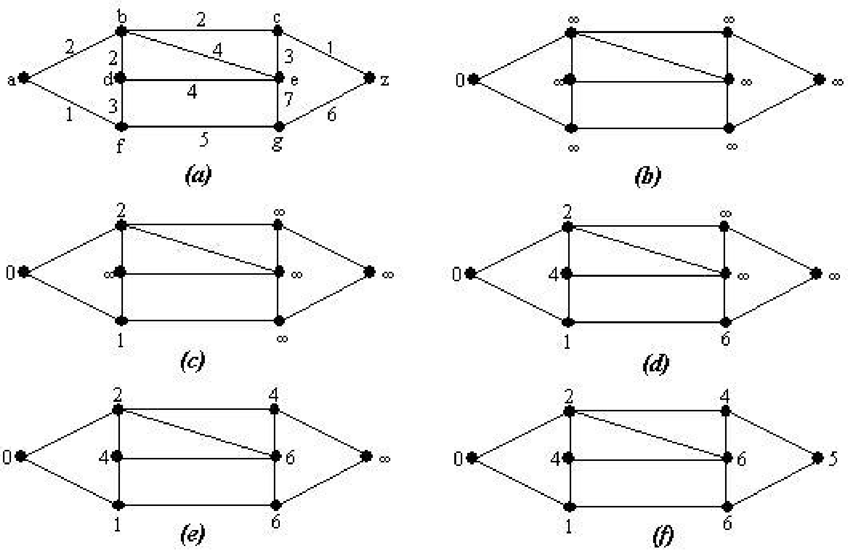


Figura 11: (a) Um grafo G e (b)-(f) os passos necessários para se obter a menor distância entre os nós a e z segundo o Algoritmo 11.

2. A cada instante, o grafo parcial formado pelos nós de G e as arestas em T consistem de várias componentes conexas (inicialmente, quando T está vazio, cada nó de G forma uma componente conexa distinta).
3. Para construir componentes conexas cada vez maiores, examinam-se as arestas de G em ordem crescente de comprimento. Se uma aresta junta dois nós em uma diferente componente conexa, adiciona-se ela a T e, conseqüentemente, as duas componente conexas transformam-se me uma. Caso contrário, a aresta é rejeitada: ela une dois nós na mesma componente conexa e não pode ser adicionada a T sem formar um ciclo.
4. O algoritmo pára somente quando uma componente conexa é determinada.

- Vejamos um exemplo de determinação de AGM usando o algoritmo Kruskal para o grafo da Figura 14 e Tabela 5.

- A complexidade do algoritmo de Kruskal é dada da seguinte forma [4]. Em um grafo com n nós e a arestas, o número de operações é:
(i) $O\left(a \log a\right)$, para ordenar as arestas, que é equivalente a $O\left(a \log n\right)$;
(ii) $O\left(n\right)$ para inicializar os conjuntos distintos de cada componente conexa;
(iii) No pior caso, $O\left(\left(2a + n - 1\right) \lg^{*} n\right)$ para determinar e misturar as componentes conexas;
(iv) $O\left(a\right)$ para o restante das operações.

- Conclui-se que o tempo total para o algoritmo de Kruskal é $O\left(a \log n\right)$.

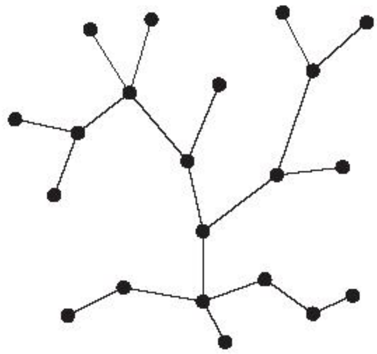


Figura 12: Exemplo de árvore.

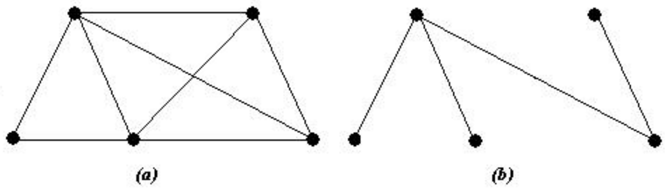


Figura 13: (b) é um exemplo de árvore geradora de (a).

4.4.2 Algoritmo Prim

- Determina uma árvore geradora mínima T de um grafo $G = (V, E)$.
- No algoritmo de Prim (Algoritmo 12), a AGM cresce naturalmente a partir de um determinado nó denominado de *raiz*. Em cada estágio, adiciona-se um novo ramo à árvore e o algoritmo pára quando todos os nós tenham sido visitados.
- Inicialmente, o conjunto de nós B contém somente um nó e o conjunto T está vazio. Em cada passo, o algoritmo olha para uma possível aresta de menor comprimento $\{u, v\}$ tal que $u \in V \setminus B$ e $v \in B$. Então, adiciona-se u a B e $\{u, v\}$ a T.

Algoritmo 12 Algoritmo de Prim — Determina uma AGM de um grafo $G = (V, E)$.

$T \leftarrow \emptyset$
$B \leftarrow$ *um nó arbitrário de* V
Enquanto $B \neq V$ **Faça**
  Determine $\{u, v\}$ de menor comprimento tal que $u \in V \setminus B$ e $v \in B$
  $T = T \cup \{u, v\}$
  $B = B \cup \{u\}$
Fim Enquanto

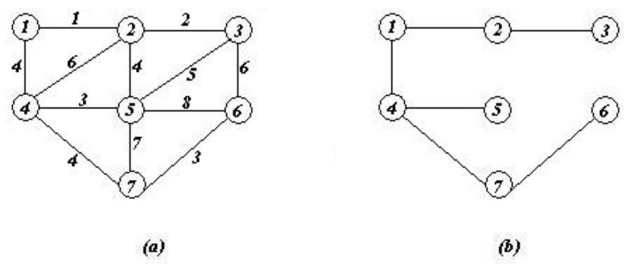


Figura 14: (a) Grafo G e sua (b) árvore geradora mínima T.

Passo	**Aresta**	**Componentes conexas**
inicialização	—	$\{1\}\ \{2\}\ \{3\}\ \{4\}\ \{5\}\ \{6\}\ \{7\}$
1	$\{1,2\}$	$\{1,2\}\ \{3\}\ \{4\}\ \{5\}\ \{6\}\ \{7\}$
2	$\{2,3\}$	$\{1,2,3\}\ \{4\}\ \{5\}\ \{6\}\ \{7\}$
3	$\{4,5\}$	$\{1,2,3\}\ \{4,5\}\ \{6\}\ \{7\}$
4	$\{6,7\}$	$\{1,2,3\}\ \{4,5\}\ \{6,7\}$
5	$\{1,4\}$	$\{1,2,3,4,5\}\ \{6,7\}$
6	$\{2,5\}$	*rejeitada*
7	$\{4,7\}$	$\{1,2,3,4,5,6,7\}$

Tabela 5: Acompanhamento do exemplo para o algoritmo Kruskal.

- Teste o Algoritmo 12 para o grafo da Figura 14.
- O *loop* principal do algoritmo de Prim é executado $n-1$ vezes; em cada iteração, o laço interno tem uma complexidade $O(n)$. Portanto, o algoritmo de Prim tem complexidade $O(n^2)$.
- Em grafos densos, o algoritmo de Prim é provavelmente melhor que o de Kruskal [4].

4.5 Exemplos / Exercícios

- **Exemplo 1** – Com base no algoritmo de Dijkstra apresentado em aula, determine a menor distância entre as cidades de *Boston* e *San Francisco* da Figura 15. Desenhe os passos necessários no desenvolvimento de sua solução. Quantas iterações foram necessárias para o cálculo dessa distância? (despreze o passo de inicialização)
- **Exemplo 2** – Determine a árvore geradora mínima do grafo da Figura 16 usando o algoritmo de Prim ou o algoritmo de Kruskal. Apresente o desenvolvimento de sua solução.
- **Exemplo 3** – O algoritmo de Kruskal foi apresentado informalmente neste capítulo. Pesquise na literatura sua representação formal e apresente-a através de um exemplo.

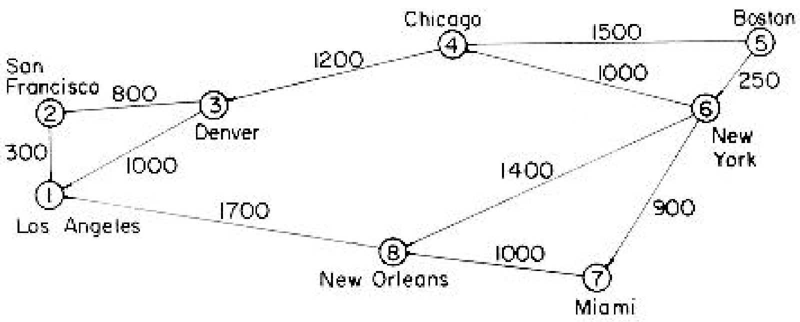


Figura 15: Determinação do caminho mínimo: alg. de Dijkstra.

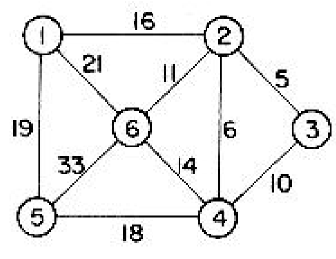


Figura 16: Determinação da árvore geradora mínima.

5 Métodos de projetos de algoritmos

5.1 Introdução

- Um problema fica bem caracterizado quando responde às seguintes questões [6]:
  Quais são os possíveis dados? (entrada)
  Quais são os resultados (saída) que podem ser esperados?
  Quando um resultado é uma resposta aceitável para um dado?

- Por exemplo, em um problema de encontrar raízes de polinômios devemos esclarecer os seguintes fatos:
  de quais polinômios vamos tratar? (coeficientes reais, graus)
  que resultados podem ser esperados? (inteiros, reais, complexos)
  quando um resultado é aceitável para um dado? (qualquer raiz, menor raiz)

- Os métodos a serem considerados neste capítulo (divisão e conquista, algoritmos gulosos e programação dinâmica) baseiam-se na idéia de decomposição de problemas complexos em outros mais simples, cujas soluções são combinadas para fornecer uma resposta ao problema original.

- Seria interessante que essas metodologias pudessem ser descritas através de um algoritmo genérico, de tal forma que se calculássemos a complexidade deste algoritmo, teríamos a complexidade de todos os outros algoritmos baseados nesta metodologia.

5.2 Divisão e conquista

- Consiste na decomposição de um problema em subproblemas independentes, resolvendo-os e combinando as soluções obtidas para formar a solução do problema original. Os algoritmos classificados como pertencentes à metodologia de divisão e conquista são *recursivos*.

- O processo de divisão e conquista funciona da seguinte maneira: dada uma entrada, se ela é suficientemente simples, obtemos diretamente a saída correspondente. Caso contrário, ela é decomposta em entrada mais simples, para as quais aplicamos o mesmo processo.

- Portanto, ocorrem 3 etapas: Divisão (dividir o problema original em subproblemas menores); conquista (resolver cada subproblema recursivamente); combinação (combinar as soluções encontradas, compondo a solução para o problema original).

- **Exemplo** — Encontrar o maior elemento de uma lista $L[1..n]$. São apresentados dois algoritmos: o primeiro (Algoritmo 13) é considerado como sendo uma solução ingênua, simples; e o segundo (Algoritmo 14), trata o problema através da método de divisão e conquista.

Algoritmo 13 Determina o maior elemento em uma lista - Solução ingênua

$max \leftarrow L(1)$
Para i de 2 até n **Faça**
  Se $L(i) > max$ **Então**
    $max \leftarrow L(i)$
  Fim Se
Fim Para

Algoritmo 14 Determina o maior elemento em uma lista - Solução por divisão e conquista

function $Maximo\,(x, y)$
Se $y - x \leq 1$ **Então**
  return $max\,(L\,(x)\,, L\,(y))$
Se Não
  $max1 \leftarrow Maximo\,(x, (x + y)\,/2)$
  $max2 \leftarrow Maximo\,((x + y)\,/2 + 1, y)$
  return $max\,(max1, max2)$
Fim Se

- **Exemplo** — Encontrar o índice do elemento x em uma lista ordenada (assumindo que ele exista). A solução ingênua seria fazer uma busca seqüencial do elemento na lista e parar a busca quando este for encontrado. Uma solução por divisão e conquista é dada no algoritmo 15. A solução consiste em comparar x com o elemento médio e depois buscar x na metade esquerda ou direita.

Algoritmo 15 Encontrar o índice do elemento x em uma lista ordenada - solução por divisão e conquista

function $Busca\,(L, x, l, r)$
Se $l = r$ **Então**
  return l
Se Não
  $m \leftarrow (l + r)\,/2$
  Se $x < L\,(m)$ **Então**
    $Busca\,(L, x, l, m)$
  Se Não
    $Busca\,(L, x, m + 1, r)$
  Fim Se
Fim Se

- Outros exemplos são os algoritmos de ordenação do tipo *quicksort* e *mergesort*.

5.3 Algoritmos gulosos

- A solução de um problema é alcançada através de uma seqüência de decisões.
- As decisões são tomadas de forma isolada, em cada passo da solução: seleciona-se um elemento e decide-se se é viável ou não.
- A estratégia é, portanto, pegar a melhor opção em cada momento (solução ótima local). Quando o algoritmo acaba, espera-se que tenha ocorrido a melhor solução.
- Normalmente esses algoritmos são de fácil implementação e eficientes.
- **Exemplo (O problema do trôco)** — No Brasil, temos moedas de 1 real, 50, 25, 10, 5 e 1 centavos. O *problema do trôco* consiste em pagar um trôco com a menor quantidade possível de moedas. Por exemplo, se tivéssemos que pagar um trôco de R$ 2,78, a solução ótima seria: 2 moedas de 1 real, 1 moeda de 50 centavos, 1 moeda de 25 centavos e 3 moedas de 1 centavo (total de 7 moedas). Esse problema é considerado guloso, porque a cada passo escolhe a maior moeda possível; uma vez escolhida a moeda, esta não será trocada.
- **Exemplo (O problema do empacotador)** — Semelhante ao problema do trôco, com a diferença que se deseja otimizar a colocação de produtos em sacolas de farmácia ou supermercado, por exemplo.
- **Exemplo (O código de Huffman)** — A codificação proposta por Huffman(1952) têm sido uma técnica importante utilizada para comprimir dados e, portanto, economizar espaço de armazenamento de arquivos [1]. O problema consiste em: *dado um texto qualquer (uma seqüência de caracteres), determinar um código para cada um dos caracteres de forma que minimize a quantidade de bits necessária para codificar o texto*. Suponha que um texto com 250 caracteres contenha seis tipos de caracteres A, B, C, D, E e F, com freqüências (quantidade de caracteres no texto) iguais a 50, 20, 30, 40, 100 e 10, respectivamente. Determine um código para cada um deles segundo o método de Huffman. Observações:

  1. Se usarmos o código ASCII extendido (8 bits), o número total de bits é igual a $250 \times 8 = 2000$ bits.
  2. Se usarmos um código de tamanho fixo de 3 bits para representar os 6 caracteres, o número total de bits é $250 \times 3 = 750$ bits.
  3. Com o código de Huffman, o número de bits cai para **580 bits!!**

- Outro exemplo de método guloso é a determinação de árvore geradora mínima, como visto no capítulo anterior (algoritmos de Kruskal e Prim).

5.4 Programação dinâmica

- A programação dinâmica resolve um problema, dividindo-o em subproblemas menores, solucionando-os e combinando soluções intermediárias, até resolver o problema original.

- A particularidade desta metodologia diz respeito à divisão do problema original: o problema é decomposto somente uma vez e os subproblemas menores são gerados antes dos subproblemas maiores; dessa forma, esse método é claramente ascendente (problemas recursivos são descendentes).

- **Exemplo (Multiplicação de cadeia de matrizes)** — Esse problema trata do cálculo do produto:

$$M = M_1 \times M_2 \times M_3 \times \ldots M_n \tag{5}$$

Um algoritmo de multiplicação de uma matriz $p \times q$ por outra $q \times r$ derivado da fórmula de multiplicação de matrizes requer $p \times q \times r$ multiplicações de elementos. Utilizando o conceito de programação dinâmica e sabendo-se que a multiplicação de matrizes é associativa, esse número reduz bastante. Considere o exemplo: (entre parênteses é dada a dimensão da matriz)

$$M = M_1\,(100 \times 3) \times M_2\,(3 \times 10) \times M_3\,(10 \times 50) \times M_4\,(50 \times 30) \times M_5\,(30 \times 5) \tag{6}$$

Calcular M por meio de $\{[(M_1 \times M_2) \times M_3] \times M_4\} \times M_5$ resulta em 218000 operações.

Agrupando-se da forma $M_1 \times \{M_2 \times [(M_3 \times M_4) \times M_5]\}$, a quantidade de operações cai para 18150.

O problema está em determinar a seqüência ótima de multiplicações, cuja quantidade de possibilidade de agrupamentos é grande. Um algoritmo de programação dinâmica decompõe o problema em partes menores, guarda valores intermediários de produtos (na diagonal de uma matriz) e, por último, combinaria a saída dessas multiplicações. Uma análise detalhada deste problema é dada em [2] e [6].

- **Exemplo (O problema do caminho mínimo)** — Baseado no algoritmo de Dijkstra, esse problema trata do cálculo não só da distância mínima entre dois pontos, como também do caminho entre eles.

5.5 Exemplos / Exercícios

- **Exemplo 1** – Algoritmo de Huffman: Dado o texto a seguir, com 229 caracteres (incluindo o espaço), determine a codificação de Huffman para cada um deles. Calcule também a quantidade de bits para o caso de se utilizar o código ASCII Extendido e caso se utilize a codificação de Huffman.

textotextotexto texto textotexto textotextotexto texto texto texto textotexto texto textotextotexto texto texto texto textotexto textotexto textotextotextotexto textotexto texto texto texto textotexto texto textotexto texto texto

- **Exemplo 2** – Escreva um algoritmo (ou uma seqüência de passos) para resolver o problema do trôco. Esse problema consiste em: dado um montante X qualquer, que corresponde ao trôco devido, e sabendo que existem moedas de 1 real, 50, 25, 10, 5 e 1 centavos, deve-se pagar o trôco com a menor quantidade possível de moedas.

6 NP completeza

6.1 Introdução

- Ainda existem diversos problemas que não possuem soluções ou mesmo algoritmos eficientes.
- O tempo de execução da maioria dos algoritmos é limitado por alguma função polinomial do tamanho da entrada.
- Esses algoritmos são denominados *eficientes* e os correspondentes problemas são chamados de problemas *tratáveis*.
- A classe de problemas que podem ser resolvidos de maneira eficiente são denotados por **P** (tempo polinomial).
- Apesar disso, alguns algoritmos de ordem exponencial, tal como o 2^n, pode ser mais eficiente que um algoritmo de ordem polinomial do tipo n^{10}, por exemplo. A função exponencial não ultrapassa a polinomial antes de n alcançar o valor 59.
- A taxa de crescimento de uma função exponencial é tão explosiva que é dito que um problema é *intratável* quando só existirem soluções de ordem exponencial para este problema.

6.2 Problemas NP

- Existe uma série de problemas cujo tempo de execução é conhecidamente não polinomial (NP).
- A questão $P \times NP$ surgiu em 1971 com o artigo *The complexity of theorem-proving procedures*.
- Esses problemas são agrupados numa denominação de NP-completo e tem a seguinte característica: *existe um algoritmo eficiente para um problema NP-completo se, e somente se, existir algoritmos eficientes para todos os problemas NP-completos.*
- Acredita-se que não existe algoritmos que satisfaçam a característica acima, mas não há prova formal disso.
- **Exemplo (Determinação de um clique)** — Dado um grafo não direcional G, um clique C é um subgrafo de G tal que todos os nós em C sejam multuamente adjacentes, como mostra a Figura 17.
- **Exemplo (Determinação de um ciclo hamiltoniano)** — Um ciclo hamiltoniano de um grafo orientado G é um ciclo simples que contém todos os vértices de G. Veja um exemplo na Figura 18.

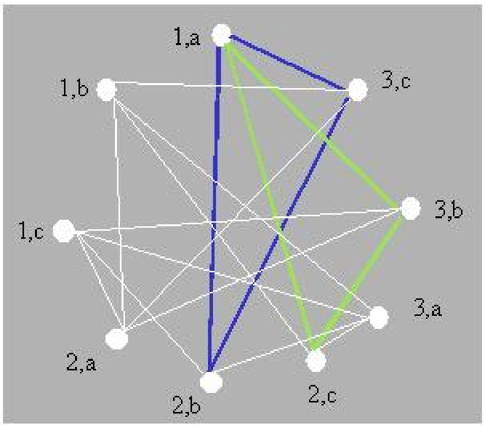


Figura 17: Clique máximo.

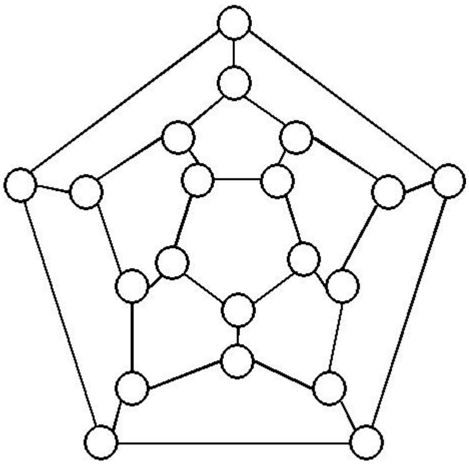


Figura 18: Determinar o ciclo hamiltoniano no grafo acima.

- Outros exemplos são o problema do caixeiro viajante, problema de coloração de grafos.

Referências

[1] U. Manber. Introduction to algortihms: A creative approach. Addison-Wesley, 1989.

[2] T. Cormen, C. Leiserson, R. Rivest. Introduction to algorithms. MIT Press, 1990.

[3] T. Cormen, C. Leiserson, R. Rivest, C. Stein. Algoritmos: Teoria e prática. Tradução da segunda edição, V. D. de Souza, Editora Campus, 2002.

[4] G. Brassard, P. Bratley. Algorithmics, theory & practice. Prentice-Hall, 1988.

[5] A. Aho, J. Hopcroft, J. Ullman. The design and analysis of computer algorithms. Addison-Wesley, 1975.

[6] L. V. Toscani, P. A. S. Veloso. Complexidade de algoritmos. Editora Sagra Luzzato, 2001.

[7] L. Kronsjo. Algorithms: Their complexity and efficiency, John Wiley, 1987.

[8] J. C. A. Figueiredo. Análise e técnicas de Algoritmos, Notas de aula (apostila) - UFPB, 2002.

[9] M. Gondran, M. Minoux. Graphes et Algorithmes, Collection de la Direction des Études et Recherche d´Électricité de France, Ed. Eyrolles, 1995.

[10] R. Gould. Graph Theory, The Benjamim/Cummings Publishing Company, 1988.