GRAFOS 8

8.1 CONCEITOS

Um *grafo* G é constituído por um conjunto N de elementos e por uma relação binária A entre esses elementos. Escrevemos

$$G = (N, A)$$

Por sua vez, a relação A é também um conjunto cujos membros são pares ordenados (n_i, n_j), onde n_i e n_j são elementos de N. Formalmente,

$$A \subseteq N \times N$$

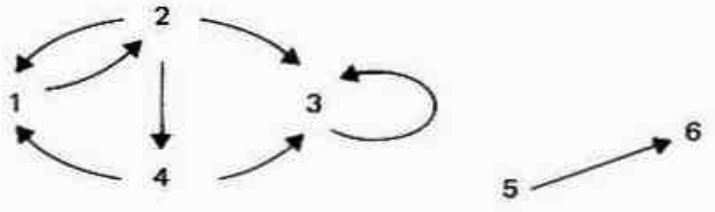
ou seja, A é um subconjunto do produto cartesiano de N por N.

Os elementos de N são denominados *nós* (ou vértices), enquanto os elementos de A são denominados *arcos* (ou arestas).

Como exemplo, sejam

$$N = \{1, 2, 3, 4, 5, 6\}$$
$$A = \{(1, 2), (2, 1), (2, 3), (2, 4), (3, 3), (4, 1), (4, 3), (5, 6)\}$$

G pode ser representado graficamente, sendo os arcos denotados por setas orientadas do primeiro para o segundo nó do respectivo par ordenado.



Nós ligados por arcos são ditos *adjacentes;* o nó 2 é adjacente a 1, 3 e 4. e 4. Diz-se também que arcos são *incidentes* de ou a determinados nós, conforme partam ou cheguem a eles; (1, 2) é incidente de 1 e (2, 1) é incidente a 1.

Note que, embora os nós 1 e 3 não estejam ligados por um arco, é possível a partir de 1 atingir 3, percorrendo os arcos (1, 2) e (2, 3). Tais arcos constituem o que se denomina de um caminho de 1 a 3. Um *caminho* é definido como uma seqüência de um ou mais arcos

$$< (a, n_1), (n_1, n_2), \ldots, (n_{i-1}, n_i), (n_i, b) >$$

em que o segundo nó de cada arco coincide com o primeiro do seguinte, permitindo a partir de um nó a atingir um nó b.

Quando a = b temos um *circuito*. No grafo do exemplo dado há um circuito unindo os nós 1, 2 e 4. Um circuito de um único arco é um laço, observado no exemplo para o nó 3.

O grafo do exemplo não tem a propriedade de ser conexo. Um grafo é *conexo* quando tem um nó do qual existem caminhos para todos os demais. No entanto, o subgrafo constituído pelos nós 1, 2, 3 e 4 é conexo, pois a partir do nó 1 é possível atingir os demais (o que também é verdade para 2 e 4, mas não para 3). Um grafo é dito *fortemente conexo* se de todos os nós é possível atingir todos os demais.

Um *subgrafo* se define como um subconjunto dos nós de um dado grafo, juntamente com todos os arcos cujas duas extremidades são nós desse subconjunto.

Assim, para o grafo G do exemplo, poderíamos definir

$$S = (N', A')$$

onde $N' = \{1, 4, 3, 5\}$; uma vez especificado N', A' fica determinado

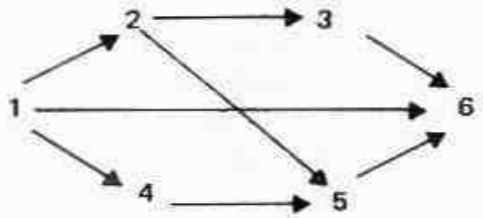
$$A' = \{(4, 1), (4, 3), (3, 3)\}$$

notando que 5 aparece como um nó isolado, já que não possui nenhum arco incidente.

Já em um *grafo parcial,* permanecem todos os nós do grafo original, mas é tomado um subconjunto de seus arcos. A figura mostra um grafo parcial P do grafo original G.

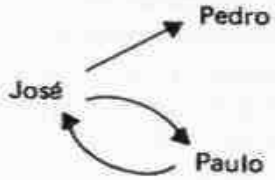
Observe que essa escolha de P tem uma propriedade interessante: todos os circuitos que havia em G foram removidos. Diz-se que P é um grafo *acíclico,* por não conter circuito algum.

Grafos acíclicos são o caso mais geral de ordem parcial, de que já vimos o caso particular das árvores. Um outro caso particular de ordem parcial são as *redes,* que possuem dois nós especiais: o nó fonte, do qual todos os demais são atingidos, e o nó sorvedouro, do qual não parte nenhum arco. Na figura abaixo o nó 1 é a fonte e o nó 6 é o sorvedouro.



Voltando à questão das relações binárias, observamos que todas as estruturas vistas anteriormente incorporam relações de ordem total ou parcial (restrita), tendo-se agora, com os grafos acíclicos, o caso geral de ordem parcial. Quando ocorrem circuitos, deixamos de ter relações de ordem, passando então ao caso mais amplo das relações binárias.

Um exemplo de relação binária é a relação ajudar. Se José ajuda Pedro, José ajuda Paulo e Paulo ajuda José, poderíamos representar essa situação pela figura abaixo.

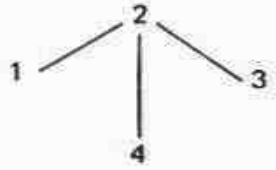


A relação ajudar não é simétrica, isto é, se (a, b) pertence à relação, isso não implica que (b, a) também pertença. Já a relação "ser parente de" é simétrica, pois se a é parente de b, então a recíproca tem de ser verdadeira.

Quando a relação é simétrica, haverá arcos sempre nos dois sentidos entre pares de nós para os quais vale a relação. Como uma simplificação, adota-se então a prática de ligar os nós por um traço sem cabeça de seta, o qual recebe a denominação de *aresta*. Por sua vez, grafos representando relações simétricas são denominados *grafos não-dirigidos* em oposição aos grafos dirigidos, vistos até agora. A figura mostra um grafo não-dirigido.

A terminologia para grafos não-dirigidos apresenta outras modificações além do uso do termo aresta em vez de arco. Por exemplo, fala-se em ciclo em vez de circuito (na figura, os nós 2, 3 e 4 formam um ciclo).

Ainda quanto à mesma figura, diz-se que o grafo não é conexo, sendo formado por dois componentes conexos $\{1, 2, 3, 4\}$ e $\{5, 6\}$. Se eliminarmos do primeiro componente conexo os arcos $\{1, 4\}$ e $\{3, 4\}$ a conexidade entre os quatro nós se mantém; diz-se que o grafo parcial mostrado abaixo



é uma árvore geradora do componente conexo $\{1, 2, 3, 4\}$. De um modo geral, obtém-se uma *árvore geradora* removendo arestas até eliminar os ciclos, mas mantendo a conexidade. É fácil ver que se um grafo tem n nós, então uma árvore geradora do grafo terá exatamente n − 1 arestas. Observe-se ainda que um grafo pode admitir diferentes árvores geradoras, conforme a escolha de arestas a eliminar.

Exercício 8.1 — Represente todas as demais árvores geradoras do grafo dado.

Grafos dirigidos ou não podem ter valores associados aos nós e/ou às arestas. Tais grafos são ditos *valorados;* um exemplo de grafo valorado será visto na seção 8.5.

8.2 CRITÉRIOS PARA PERCORRER GRAFOS

Vimos no capítulo sobre árvores os critérios mais usuais para percorrer árvores binárias. Veremos aqui os dois critérios mais comuns para percorrer grafos: percurso em amplitude ("breadth first search") e percurso em profundidade ("depth first search").

Em ambos os casos, parte-se de um nó v escolhido arbitrariamente e "visita-se" o nó. Em seguida, considera-se cada um dos nós w adjacentes a v. No *percurso em amplitude,* para cada um desses nós w:

a) visita-se o nó w

b) coloca-se o nó w em uma fila

e ao terminar de visitar os nós w, toma-se o nó que estiver na frente da fila e repete-se o processo visitando cada um dos nós adjacentes a ele e colocando-os na fila. Deve-se acrescentar que quando um nó é visitado ele é "marcado", de modo que se for encontrado novamente no percurso não será visitado outra vez nem colocado na fila.

No *percurso em profundidade,* logo que é encontrado o primeiro w adjacente a v:

a) visita-se o nó w

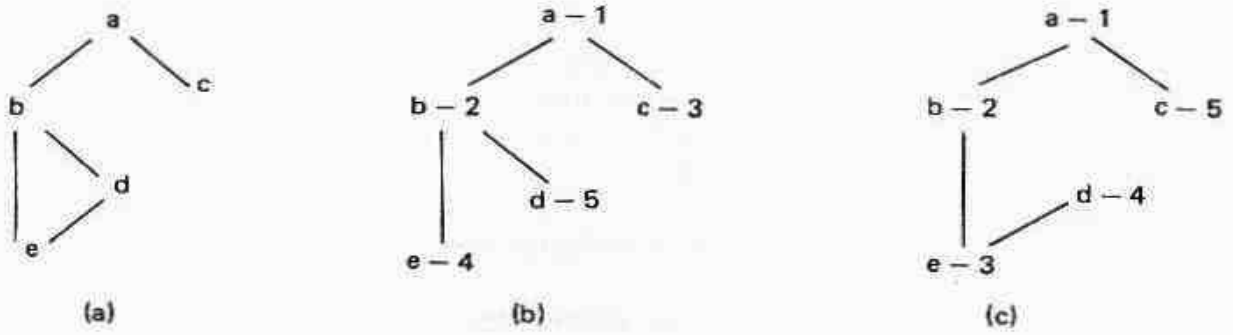
b) coloca-se o nó v em uma pilha

c) faz-se v ← w

sendo que a conseqüência do passo c é que o processo de considerar nós adjacentes do (antigo) nó v é interrompido, passando-se a procurar nós adjacentes a w. Com a repetição desse processo vai-se o mais "longe" possível no grafo, donde a denominação do percurso em profundidade. A marcação de nós visitados é feita como no outro tipo de percurso. Após visitar um nó e tomá-lo como (novo) nó v, se ele não tiver nós adjacentes ainda não visitados, toma-se como (novo) nó v aquele colocado no topo da pilha, e se recomeça o processo de considerar seus nós adjacentes w (isto é, sobe-se de um nível no grafo).

Na figura, é dado um grafo (a) e por meio de números indicamos a ordem em que seria percorrido, iniciando o percurso pelo nó a em amplitude (b) e em profundidade (c).

Note que em ambos os casos os percursos determinam uma árvore geradora, o que imediatamente sugere como se pode proceder a essa determinação (o que será feito na seção 8. 4).



Um procedimento para o percurso em amplitude, pressupondo algum procedimento adequado para visitar nós e procedimentos para manipular filas, seria:

```
proc ampl (v: nó)_
    var t, w: nó;
    var f: fila;
    início
        execute visite (v);
        execute marque (v);
        execute enfileire (v, f);
        enquanto ¬ fila_vazia (f) faça
            início
                t ← frente (f);
                execute desenfileire (f);
                para cada w adjacente a t faça
                    se ¬ marcado (w) então
                        início
                            execute visite (w);
                            execute marque (w);
                            execute enfileire (w, f)
                        fim
            fim
    fim
```

Exercício 8.2 – "Execute" manualmente o procedimento ampl sobre o grafo acima, representando o conteúdo da fila f a cada passo.

Um procedimento para o percurso em profundidade, pressupondo como antes um procedimento para visitar nós e procedimentos para manipular pilhas, seria:

```
proc prof (v: nó)_
    var t, w: nó;
    var p: pilha;
    início
        execute visite (v);
        execute marque (v);
        execute empilhe (v, p);
```

```
    enquanto ¬ pilha_vazia (p) faça
        início
            t ← topo (p);
            execute desempilhe (p);
            para cada w adjacente a t faça
                se ¬ marcado (w) então
                    início
                        execute visite (w);
                        execute marque (w);
                        execute empilhe (t, p);
                        t ← w
                    fim
        fim
fim
```

Exercício 8.3 — "Execute" manualmente o procedimento prof sobre o grafo do exemplo, representando o conteúdo da pilha p a cada passo.

É interessante observar a semelhança dos dois procedimentos. Em ambos há dois esquemas de repetição: o mais externo, prosseguindo até que a estrutura auxiliar (fila ou pilha) esteja vazia; e o mais interno, tomando sucessivamente os nós adjacentes ao nó t sendo considerado. No caso do percurso em profundidade, esse esquema mais interno é interrompido.

Exercício 8.4 — Indique qual o comando no procedimento prof que provoca a interrupção do esquema interno de repetição, e mostre como esse esquema é retomado mais tarde, através do uso da pilha p.

Uma forma bem mais simples do procedimento para percurso em profundidade é a forma recursiva (como já foi visto antes, a utilização de pilhas é um modo de obter o efeito de chamadas recursivas):

```
proc profr (v: nó)_
    var w: nó;
    início
        execute visite (v);
        execute marque (v);
        para cada w adjacente a v faça
            se ¬ marcado (w) então execute profr (w)
    fim
```

8.3 REALIZAÇÕES

Um grafo pode ser representado por uma matriz de adjacência a. Denotando por números inteiros os nós do grafo, então se i e j são nós:

a [i, j] = V se i é adjacente a j.
F em caso contrário

E a matriz, para um grafo de 10 nós, seria do tipo:

tipo matriz_adjacência : : *vet* [1 . . 10,1 . . 10] *de log*

A matriz de incidência b indica os dois nós correspondentes a cada aresta; supondo as arestas também denotadas por números inteiros, então se i é um nó e k é uma aresta:

b [i, k] = V se i é incidente a k
F em caso contrário

Para um grafo com 10 nós e 15 arestas, temos:

tipo matriz_incidência : : *vet* [1 . . 10, 1 . . 15] *de log*

Outra representação é constituída pelas listas de adjacência, em que a cada nó se associa a lista dos nós que lhe são adjacentes:

tipo célula : : *reg* (nó: *int*, outro: *ref* célula);
tipo listas_adjacência : : *vet* [1 . . 10] *de ref* célula

Para o grafo do início deste capítulo, a matriz de adjacência seria:

	1	2	3	4	5	6
1		V				
2	V		V	V		
3			V			
4	V		V			
5					V	
6						

onde omitimos os valores F por simplicidade. (Para grafos não dirigidos, note que a matriz seria simétrica).

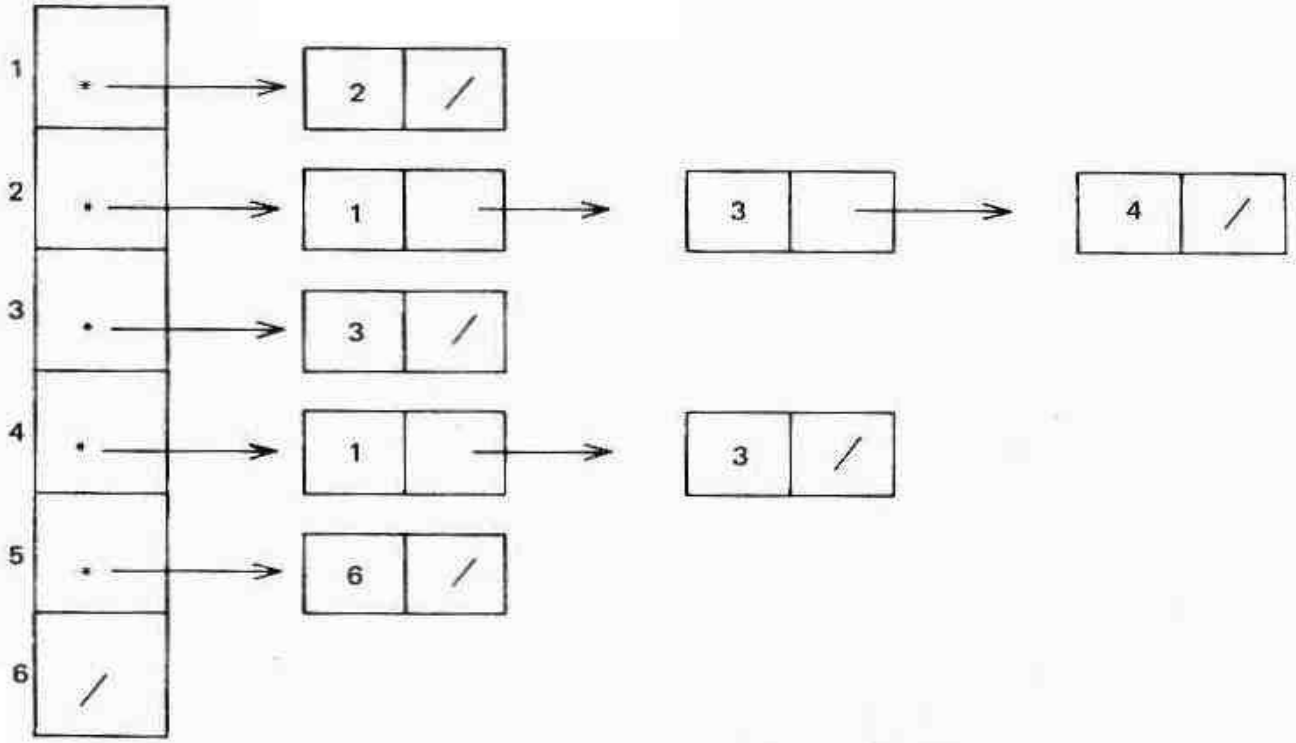
A matriz de incidência seria:

	1	2	3	4	5	6	7	8
1	V	V				V		
2	V	V	V	V				
3			V		V		V	
4				V		V	V	
5								V
6								V

A representação não distingue entre nó de origem e de destino dos arcos, o que não teria importância para grafos não-dirigidos. Se a distinção é importante, podemos utilizar os inteiros — 1 para denotar origem, 1 para denotar destino, e 0 para os nós que não par-

ticipam do arco, com o que teremos uma matriz de *inteiros* ao invés de valores *lógicos*. O caso especial de laços, como no nó 3 (arco 5), deve ser previsto; poderíamos colocar – 1 ou 1 na posição correspondente.

As listas de adjacências seriam:



Podemos também representar as listas utilizando uma matriz com uma linha para cada nó e uma coluna para o maior número de nós adjacentes a qualquer nó mais 1. O encontro do primeiro valor 0 corresponde ao *nil* que assinala fim de cada lista:

	1	2	3	4
1	2	0	0	0
2	1	3	4	0
3	3	0	0	0
4	1	3	0	0
5	6	0	0	0
6	0	0	0	0

Quanto a grafos valorados, vamos considerar o caso de valores correspondentes aos arcos. Todas as representações vistas podem ser adaptadas para essa situação:

- matriz de adjacência: colocar o valor do arco i para j na posição a [i, j]
- matriz de incidência: para um arco k, do nó i para o nó j, com valor v, fazer a [i, k] = – v e a [j, k] = v;

— listas de adjacência: adicionar um terceiro componente, valor, do tipo adequado, a cada célula; na lista do nó i, o valor do arco (i, j) estaria na célula cujo componente (nó) correspondesse a j.

Muitas outras representações são possíveis para obter maior eficiência na resolução de problemas específicos. Além disso, há representações para aspectos particulares de um grafo, como por exemplo a conexidade. Em uma matriz de conexidade c, se i e j são nós, temos:

c [i, j] = V se existe algum caminho de i a j
F em caso contrário

e para um grafo com 10 nós teríamos:

tipo matriz_conexidade : : *vet* [1 . . 10, 1 . . 10] *de log*

Exercício 8.5 — Seria possível utilizar a matriz de conexidade para representar completamente a estrutura de um grafo? Prove que isso é possível, ou mostre um contra-exemplo, desenhando dois grafos com conjuntos diferentes de arestas, mas com a mesma matriz de conexidade.

8.4 APLICAÇÃO: Árvores geradoras e componentes conexos

Vimos que ao percorrer um grafo em amplitude ou em profundidade, tomando o cuidado de não retomar nós já visitados, estamos determinando indiretamente uma árvore geradora. Para que essa determinação se efetive, basta armazenar ou simplesmente imprimir a identificação dos nós (números inteiros, se for essa a representação usada) que constituem as extremidades das arestas que fazem parte da árvore.

Quando o grafo não for conexo, deveremos determinar uma árvore geradora de c da componente (e com isso estaremos também determinando os próprios componentes).

O algoritmo que veremos a seguir é uma extensão simples do procedimento recursivo para percurso em profundidade. O grafo estará representado por listas de adjacência (lisadj) e a marcação de nós visitados se fará em um *vetor lógico* vis, onde vis [i] = V indica que o nó i foi visitado. Faremos menção a um

tipo v : : *vet* [1 . . 10] *de log*

que será o tipo da variável vis.

O algoritmo envolve um procedimento ger que a cada passo percorre vis (inicializada com F para todos os elementos) procurando o primeiro vis [i] = F e chamando o procedimento recursivo de percurso em profundidade profr, com argumento i, como ponto inicial do percurso (raiz da árvore geradora).

O procedimento profr foi modificado para imprimir as identificações dos nós das arestas da árvore geradora; como profr marca os nós que conseguiu atingir, se o procedimento ger (após o retorno de profr) ainda encontrar algum vis [i] = F, então trata-se de um outro componente conexo para o qual profr deve ser novamente chamado, para determinar a respectiva árvore geradora.

```
proc ger ( )_
    var i: int;
    para i de 1 até 10 faça
    se vis [i] = F então
        início
            escreva "novo componente";
            execute profr (i)
        fim
proc profr (i: int)_
    var j: int;
    var j: int;
    var t: ref célula;
    início
        vis [i] ← V;
        t ← lisadj [i];
        enquanto t ≠ nil faça
            início
                j ← t ↑ . nó;
                se ¬vis [j] então
                    início
                        escreva i, j;
                        execute profr (j)
                    fim;
                t ← t ↑ . outro
            fim
    fim
```

8.5 APLICAÇÃO: Caminho máximo ou mínimo

A aplicação a ser vista nesta seção se refere a uma rede valorada. Vimos que uma rede é um grafo dirigido sem circuitos, possuindo dois nós especiais: a fonte (nó 1) e o sorvedouro (nó n).

Consideremos que aos arcos são associados valores com alguma significação (distâncias entre os locais denotados pelos nós, por exemplo). Podemos calcular os valores totais dos caminhos de 1 a n. Um problema de importância prática é determinar qual (ou quais) desse(s) caminho(s) tem o valor total máximo ou mínimo; um tal caminho se denomina caminho máximo ou caminho mínimo, conforme o caso.

Um algoritmo conveniente procuraria evitar o método óbvio, mas trabalhoso, de achar e comparar todos os caminhos. Veremos aqui o método de Bellman-Kalaba, que parte do princípio de que um caminho ótimo (máximo ou mínimo) de 1 a n passando por algum nó i deve necessariamente utilizar aquele dos subcaminhos de i a n que seja ótimo, não sendo portanto necessário considerar os demais subcaminhos.

Vamos descrever o algoritmo para o caso de caminho máximo. Seja d [i, j] o valor associado ao arco de i para j e seja v [i] o valor total de algum caminho de i a n sendo considerado. Pelo algoritmo, tomamos na iteração inicial k = 1:

$$v^{(1)}[i] \leftarrow d[i, n] \qquad \text{para } i = 1, 2, \ldots, n-1$$
$$v^{(1)}[n] \leftarrow 0$$

e nas k iterações sucessivas:

$$v^{(k)}[i] \leftarrow \max_j \left\{ d[i, n] + v^{(k-1)}[j] \right\} \qquad \text{para } i = 1, 2, \ldots, n-1$$
$$v^{(k)}[n] \leftarrow 0$$

encerrando-se o processo quando

$$v^{(k)}[i] = v^{(k-1)}[i] \qquad \text{para } i = 1, 2, \ldots, n$$

estando o valor do caminho máximo em v [1].

É fácil ver que o processo parte dos nós adjacentes a n (caminhos de um arco), e a cada iteração k considera todos os caminhos dos vários nós até n com k arcos.

Também é fácil concluir que se trata de um percurso em amplitude, iniciado no nó n e percorrendo a rede no sentido oposto ao dos arcos.

No procedimento caminho, que apresentaremos, não utilizaremos nenhuma representação específica para a rede (a representação por listas de adjacência estendida para conter os valores dos arcos seria adequada). Suporemos um processo de marcação de nós para indicar os nós que no momento estão na fila, para evitar que um nó apareça simultaneamente mais de uma vez na fila. Suporemos também alguma estrutura auxiliar para indicar que de um nó i deve-se seguir para um nó j, no caminho sendo determinado. Finalmente, é necessária alguma estrutura para guardar os valores v. O procedimento retorna o valor total do caminho máximo.

```
proc caminho:int (n:nó)_
    var t, w: nó;
    var f: fila;
    início
        execute enfileire (n, f);
        execute marque (n);
        enquanto ¬fila_vazia (f) faça
            início
                t ← frente (f);
                execute desenfileire (f);
                execute desmarque (t);
                para cada w em um arco (w, t) faça
                    se v [w] < d [w, t] + v [t] então
                        início
                            v [w] ← d [w, t] + v [t];
```

```
                    no subcaminho de w a n coloque
                    t como seguinte a w;
                    se ¬marcado [w] então
                       início
                          execute enfileire (w, f);
                          execute marque (w)
                       fim
                 fim
           fim;
        retorne v [1]
     fim
```

Note que se algum v [w] é alterado, por se ter encontrado um subcaminho de valor maior, então a condição de parada

$$v^{(k)}[i] = v^{(k-1)}[i] \qquad \text{para } i = 1, 2, \ldots, n$$

não foi verificada. No procedimento, w é colocado na fila e assim a condição de fila_vazia não ocorre; isso porque, como v [w] foi modificado, então é o caso de, para todo nó u tal que o arco (u, w) exista, reconsiderar o subcaminho de u a n passando por w.

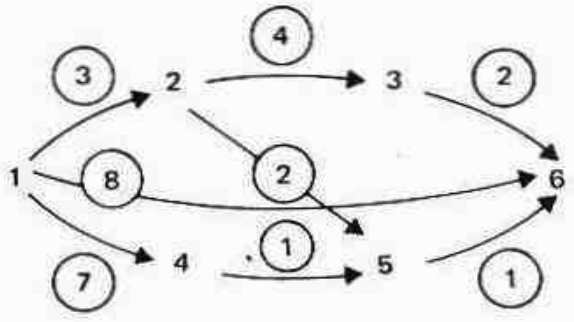
Antes de chamar o procedimento, é necessário inicializar todos os elementos de v com o valor zero.

Se for desejado o caminho mínimo, a adaptação do procedimento é simples, bastando alterar a comparação dos valores para

se v [w] > d [w, t] + v [t] *então* . . .

e inicializar os elementos de v com um valor $-\infty$ (na prática escolhendo um valor extremamente pequeno, menor de qualquer forma do que o menor valor em qualquer arco), fazendo porém v [n] = 0, como no caso do caminho máximo.

Exercício 8.6 — Acompanhe a execução do procedimento sobre a rede abaixo, procurando um dos caminhos máximos e o (único, no caso) caminho mínimo.



Exercício 8.7 — Complete o procedimento, especificando uma estrutura para indicar o nó seguinte a cada nó. Essa estrutura deverá permitir recompor não apenas um caminho ótimo de 1 a n, como também um caminho ótimo de cada um dos outros nós a n.

8.6 EXERCÍCIOS ADICIONAIS

1. O procedimento profr da seção 8.2 foi escrito de modo independente da representação de grafos a ser usada. Reescreva o procedimento, considerando a representação matricial de listas de adjacência.

2. Escreva um procedimento para a obtenção da matriz de conexidade de um grafo dirigido, a partir da sua matriz de adjacência. Sugestão: considere o fato de que se um nó i é adjacente a um nó j, então i é conexo a todos os nós conexos a j.

3. A representação de listas de adjacência utilizando vetor e referência é adequada para grafos que são alterados apenas quanto às arestas. Proponha uma representação que permita também acrescentar ou retirar nós.

4. Mostre que, se os nós forem convenientemente numerados, todo grafo dirigido acíclico pode ser representado em uma matriz de adjacência triangular superior.

5. Denomina-se grafo de interseção associado a conjuntos, todo grafo não-dirigido em que cada nó corresponde a um dos conjuntos e, dados dois conjuntos i e j, há uma aresta entre i e j sempre que os dois conjuntos possuírem pelo menos um elemento em comum. Escreva um procedimento para obter o grafo de interseção associado a n conjuntos dados. O procedimento deverá trabalhar sobre uma matriz *lógica* m por m, em que as linhas correspondem aos elementos e as colunas aos conjuntos, e em que m (k, i) = V significa que o elemento k pertence ao conjunto i; para o grafo, utilize a representação de matriz de adjacência.

9 PESQUISA DE DADOS

O processamento de uma aplicação usualmente envolve a manipulação de um grande número de estruturas chamadas *tabelas*, onde são registrados os dados a serem alterados ou consultados.

Uma tabela é formada por uma coleção de *entradas*, cada uma delas formada por um conjunto de *campos*. Usualmente, uma tabela armazena informações sobre vários objetos de um mesmo tipo, a cada objeto correspondendo uma entrada. Podemos ter, por exemplo, uma tabela contendo informações sobre funcionários de uma empresa e, neste caso, teríamos uma entrada para cada funcionário, como no exemplo abaixo:

	NÚMERO	NOME	DATA NASC.	SALÁRIO
1	10130	JOSE	25 07 50	12000
2	10850	PAULO	23 12 51	11000
3	12700	PEDRO	10 06 53	7000
4	11300	MARIO	12 09 48	8000
5	13400	ALVARO	20 05 52	13000
6	12100	CARLOS	13 08 47	17000
7	13500	JOAO	11 08 47	14000
8	13700	MARIA	10 09 49	11500
9	12400	WILSON	07 01 50	12700
10	11600	EDISON	05 03 47	9800

Fig. 9.1 Exemplo de Tabela.

Na tabela do exemplo, as entradas são divididas em quatro campos (NÚMERO, NOME, DATA NASC. e SALÁRIO), associados aos quatro atributos correspondentes da entidade (empregado) representada.

Usualmente, pelo menos um dos campos da tabela constitui uma *chave primária*, ou seja, apresenta, obrigatoriamente, um valor diferente para cada entrada. No exemplo anterior, supondo que NÚMERO é uma chave primária, concluímos que dado um valor de NÚMERO fica identificada, de maneira não ambígua, uma entrada da tabela, a menos que não haja uma entrada com tal valor de chave.

Chave de uma entrada é o valor do campo chave naquela entrada.

Uma das operações mais freqüentes e importantes em processamento de dados é a de consulta (ou acesso) a dados armazenados em uma tabela. Esta operação consiste no acesso a uma particular entrada da tabela, dado o valor da chave desta entrada.

Trataremos, nesta seção, dos métodos de pesquisa que nos permitem localizar, em uma tabela, aquela entrada que possui o valor de chave fornecido como argumento de pesquisa.

Apresentaremos aqui os métodos de pesquisa seqüencial, pesquisa binária e pesquisa por cálculo de endereço. O método de pesquisa em árvore binária é apresentado na seção 7.10.2.

Nesta seção serão entendidos como definimos os tipos *tabela*, *entrada* e *chave*, sendo este último o tipo do campo usado como chave primária da tabela. A referência ao valor da chave da i-ésima entrada de uma tabela t será denotada por t[i] . chave, sendo *chave* o seletor do campo, que é a chave primária da tabela t.

9.1 PESQUISA SEQÜENCIAL

Este é o método mais simples de pesquisa, e consiste em uma varredura serial da tabela, durante a qual o argumento de pesquisa é comparado com a chave de cada entrada até ser encontrada uma que seja igual, ou ser atingido o final da tabela, caso a chave procurada não esteja presente na tabela. A seguir é apresentado um algoritmo de pesquisa em uma tabela não ordenada. O desempenho deste algoritmo é bastante modesto, já que o número médio de comparações para a localização de uma entrada arbitrária é dado por:

$$NC = (n + 1)/2$$

considerando-se que todas as entradas possuem a mesma probabilidade de serem solicitadas.

Exercício 9.1 — Mostre que o número médio de comparações para a localização de uma entrada, por pesquisa seqüencial, é dado por $(n + 1)/2$, sendo n o número de entradas da tabela.

```
proc seqüencial
   (t: tabela; arg: chave; n, i: int) —
     {t: tabela onde será feita a pesquisa,
      arg: argumento de pesquisa,
      n: número de entradas da tabela,
      i: índice da entrada procurada (i = 0: não existe a entrada}
   var r: int;
   início
      i ← 0;
      para r de 1 incr 1 até n faça
         se t[r].chave = arg
         então
            início
               i ← r;
               escape
```

```
        fim
fim     {seqüencial}
```

A pesquisa seqüencial apresenta um desempenho um pouco melhor se a tabela estiver ordenada pelo valor da chave. O algoritmo *seg-ordenado* apresentado a seguir executa pesquisa seqüencial sobre uma tabela ordenada pelo valor da chave.

```
proc seq_ordenado
   (t: tabela; arg: chave; n, i: int)_
     {t: tabela onde será feita a pesquisa,
      arg: argumento de pesquisa,
      n: número de entradas de t,
      i: índice da entrada procurada (i = 0: não existe a entrada)}
   var r: int;
   início
      i ← 0;
      para r de 1 incr 1 até n faça
         se t[r].chave ⩾ arg
         então
            início
               se t[r].chave = arg
               então
                  i ← r;
               escape
            fim
fim     {seq_ordenado}
```

Até agora apenas consideramos tabelas nas quais todas as entradas são solicitadas com a mesma probabilidade; no entanto, é bastante comum o fato de algumas entradas serem mais solicitadas do que outras. Assim, se tivermos as entradas mais solicitadas colocadas nas posições iniciais da tabela, o número médio de comparações será menor do que se tivermos as entradas dispostas aleatoriamente.

Consideremos, como exemplo, uma tabela com quatro entradas com as seguintes chaves e freqüências relativas de acesso:

A(0, 50) B(0, 30) C(0, 15) D(0, 05)

O número médio de comparações para a localização de uma entrada, considerando-se que elas aparecem na seqüência dada, dentro da tabela, será dado por:

$$NC^* = 1 \cdot 0{,}5 + 2 \cdot 0{,}3 + 3 \cdot 0{,}15 + 4 \cdot 0{,}05 = 1.75 \text{ comparações}$$

Se tivéssemos as entradas distribuídas aleatoriamente na tabela, o número médio de comparações seria dado por:

$$NC = (n + 1)/2 = (4 + 1)/2 = 2{,}5 \text{ comparações}$$

Como é difícil conhecermos antecipadamente a distribuição das freqüências de acesso às entradas, podemos fazer com que durante o processo de pesquisa haja uma migração

das entradas mais solicitadas para o início da tabela. Uma estratégia consiste em mover a entrada para o início da tabela a cada vez que ela for solicitada. Isto é viável nos casos em que a tabela é organizada como uma lista encadeada. O algoritmo *seq-auto-org* executa pesquisa seqüencial sobre uma tabela encadeada e move a entrada acessada para o início da lista. Supõe-se a existência de um campo (elo) adicional em cada entrada, cujo conteúdo é o endereço da próxima entrada na lista.

```
proc seq_auto_org (arg: chave; i, t: ref entrada)_
  {t: endereço da primeira entrada,
   arg: argumento de pesquisa,
   i: endereço da entrada procurada}
  var i1, r: ref entrada;
  início
     i ← t;
     enquanto i ≠ nil faça
        início
           se i↑.chave = arg
           então
              se i ≠ t
              então
                 início
                    i1↑.elo ← i↑.elo;
                    i↑.elo ← t;
                    t ← i;
                    escape
                 fim
           i1 ← i;
           i ← i↑.elo
        fim
fim    {seq_auto_org}
```

Exercício 9.2 – Reescreva o algoritmo seq_auto_org, para uma tabela organizada como um vetor de entradas, ao invés de uma lista encadeada.

Exercício 9.3 – Experimente a idéia de busca seqüencial auto-organizada em alguns exemplos na tabela do início do capítulo.

9.2 PESQUISA BINÁRIA

A pesquisa binária é um método que pode ser aplicado a tabelas ordenadas, armazenadas em dispositivos de acesso direto. O método consiste na comparação do argumento de pesquisa (arg) com a chave daquela entrada localizada no meio da tabela (entrada de ordem n *div* 2). Se *arg* for igual à chave da entrada, a pesquisa termina com sucesso. Se *arg* for maior, o processo é repetido para a metade superior da tabela e se for menor, para a metade inferior.

Assim, a cada comparação, a área de pesquisa é reduzida à metade do número de elementos. A pesquisa binária é ilustrada a seguir, por meio de um exemplo:

arg = 32

endereços / chaves:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	15	17	30	32	34	40	50	80	90	95	97	99	101	105
			2º	4º	3º		1º							

O número máximo de comparações será, portanto, igual a

$$\lfloor \log_2 n \rfloor + 1,$$

para a localização de uma entrada ou para a constatação de que ela não está presente na tabela.

A seguir, é apresentado o algoritmo *pesq-binária*, que implementa este método.

```
proc pesq_binária
   (t: tabela; arg: chave; n, i: int)_
  {t: tabela onde será feita a pesquisa,
   arg: argumento de pesquisa,
   n: número de entradas da tabela,
   i: índice da entrada procurada}
   var r, s, v, u: int;
   início
      i ← 0; r ← 1; v ← n;
      enquanto r ≤ v .e. i = 0 faça
         início
            u ← (r + v) div 2;
            se arg = t[u].chave
               então
                  i ← u
               senão
                  se arg > t[u]↓.chave
                  então
                     r ← u + 1
                  senão
                     v ← u − 1
         fim
   fim    {pesq_binária}
```

Exercício 9.4 – Calcule o número médio de comparações necessário para localizar uma entrada em tabelas com 15, 127 e 32767 entradas, para pesquisa seqüencial e para pesquisa binária.

9.3 CÁLCULO DE ENDEREÇO

O método de cálculo de endereço ("hashing") não é apenas um método de pesquisa, mas também um método de organização física de tabelas. Ele consiste, basicamente, no

armazenamento de cada entrada em um endereço calculado pela aplicação de uma função à chave da entrada. O processo de pesquisa sobre uma tabela organizada desta maneira é similar ao processo de inserção de uma entrada e consiste na aplicação da função de cálculo de endereço ao argumento de pesquisa, obtendo como resultado o endereço da entrada procurada. O esquema que segue ilustra este tipo de organização.

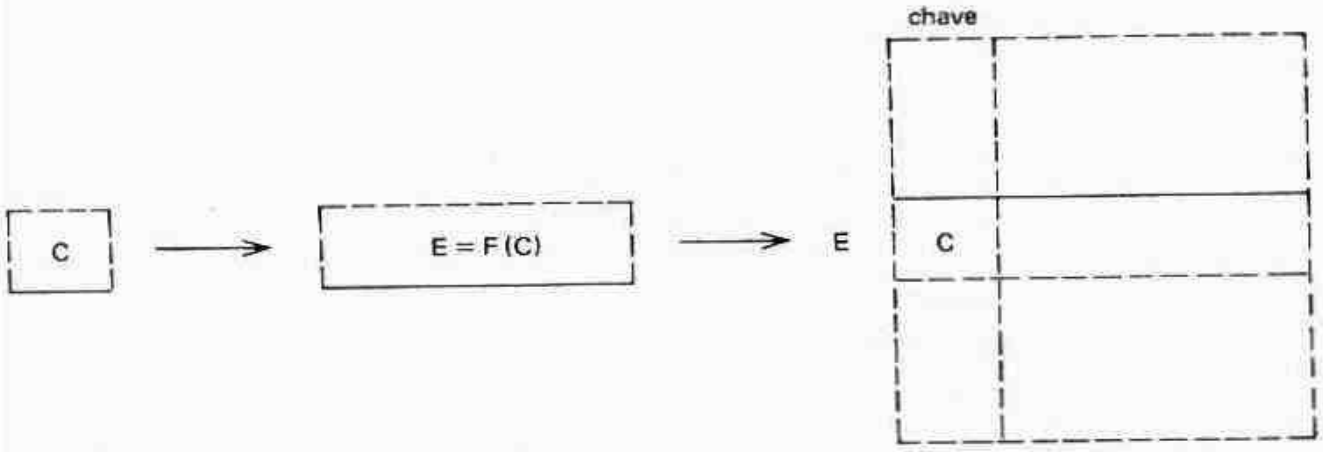


Fig. 9.2 Organização de Tabela usando "hashing".

A eficiência da pesquisa neste tipo de organização depende fundamentalmente da função de cálculo de endereço. A função ideal seria aquela que gerasse um endereço diferente (entre 1 e n) para cada um dos n diferentes valores da chave presentes na tabela. No entanto, normalmente não é possível conseguir este tipo de função e somos obrigados a usar funções que geram colisões, isto é, que atribuem o mesmo endereço a diferentes valores da chave.

Tomemos, a título de ilustração, uma tabela que admita um máximo de 53 entradas e que tenha o campo chave apresentando valores contidos no intervalo [0 . . 1000]. Tomemos, também a título de exemplo, a seguinte função de cálculo de endereço:

$$E(C) = (C \; mod \; 53) + 1$$

Portanto, a função E recebe um argumento que é um valor entre 0 e 1000, e calcula um endereço entre 1 e 53. A seguir, consideremos os seguintes valores de chave e os endereços correspondentes:

chave	383	487	235	527	510	320	203	108	563	500	646	103	63
endereço	12	10	23	50	33	2	44	2	33	23	10	50	10

Neste exemplo, notamos que o endereço 33 é gerado para as chaves 510 e 563, enquanto o endereço 10 é gerado para as chaves 487, 646 e 63.

Exercício 9.5 — Calcule os endereços gerados pela função

$$E'(C) = \lfloor (C/1000) * 52.9 + 1 \rfloor$$

para os valores de chave da tabela acima. Note que a função E' preserva a ordem; ou seja, $C1 \geqslant C2 \Rightarrow E'(C1) \geqslant E'(C2)$.

Como em cada endereço apenas uma entrada pode ser armazenada, é necessário que se estabeleça uma solução para o problema das colisões; isto é, deve ser estabelecido um procedimento para encontrar um endereço livre onde armazenar aquelas entradas para as quais é gerado um endereço já ocupado. Um dos procedimentos mais simples e usuais para solucionar este problema consiste em procurar seqüencialmente, a partir do endereço gerado, o primeiro endereço livre e nele armazenar a nova entrada. Esta solução é chamada de "endereçamento aberto".

Para o exemplo anterior, os endereços efetivamente ocupados pelas entradas seriam os seguintes:

chave	383	487	235	527	510	320	203	108	563	500	646	103	63
Endereço calculado	12	10	23	50	33	2	44	2	33	23	10	50	10
Endereço efetivo	12	10	23	50	33	2	44	3	34	24	11	51	13

A seguir são apresentados os algoritmos de inserção e pesquisa para uma tabela organizada por cálculo de endereço, com o uso de endereçamento aberto para as colisões.

```
proc insere_cálculo
   (t: tabela; e: entrada; n: int)_
     {t: tabela onde será feita a inserção,
      e: entrada a ser inserida,
      n: número de endereços em t}
   var end: int;
   início
      end ← (e.chave mod n) + 1;
      enquanto t[end].ocupado faça
         end ← (end mod n) + 1;
      t[end] ← e;
      t[end].ocupado ← V;
      t[end].usado ← V
   fim     {inser-cálculo}
```

É suposta a existência de um campo denominado "ocupado" na tabela. Este campo é do tipo *log* e tem o valor V (verdadeiro) para todas as entradas ocupadas e o valor F (falso) para aqueles endereços livres na tabela. O campo "usado" possui o valor V em todos os endereços da tabela que estão ocupados ou que já estiveram desde a criação da tabela. Assim, ao ser excluída uma entrada da tabela, apenas o seu campo "ocupado" passa de V para F, enquanto o campo "usado" permanece com o valor V.

Exercício 9.6 — Execute manualmente o procedimento insere_cálculo para um exemplo.

Esta distinção entre endereços que estão livres porque não foram usados e endereços liberados pela exclusão de entradas é fundamental para o algoritmo de pesquisa, já que este detecta o fato de que um particular valor da chave não está presente na tabela ao

encontrar um endereço ainda não usado, durante o processo de busca. Este algoritmo é então mostrado a seguir.

```
proc pesq_cálculo
   (t: tabela; arg: chave; n, i: int)_
     {t: tabela onde será feita a pesquisa,
      arg: argumento de pesquisa,
      n: número de entradas de t,
      i: índice da entrada procurada}
   var end: int;
   início
      i ← 0;
      end ← (chave mod n) + 1;
      enquanto t[end].usado faça
         se t[end].chave = arg .e. t[end].ocupado
            então
               início
                  i ← end;
                  escape
               fim
            senão
               end ← (end mod n) + 1
   fim {pesq_cálculo}
```

Note que nestes dois algoritmos estamos usando a função de cálculo de endereço dada por

$$E(C) = (C \bmod n) + 1;$$

Para o uso de uma outra função, basta que se altere nos dois algoritmos a instrução onde é calculado o endereço. É importante notar que a mesma função deve ser usada nos dois algoritmos e que a troca de função após a criação da tabela exige que todas as entradas nela existentes (ocupadas) sejam reinseridas com o uso da nova função.

Uma solução alternativa para o problema das colisões consiste no uso de *encadeamento*. Neste caso, todas as entradas que colidem em um mesmo endereço são coletadas em uma mesma lista encadeada, que tem como "header" (cabeça) aquela entrada armazenada no endereço onde todas colidiram. O processo de inserção é semelhante àquele descrito para o endereçamento aberto, sendo a entrada inserida na lista correspondente, ao ser armazenada.

O processo de pesquisa é que tem a sua eficiência sensivelmente alterada, já que ao ser calculado um endereço *end*, a busca se restringe àquelas entradas encadeadas na lista encabeçada pela entrada que está no endereço *end*.

Exercício 9.7 — Descreva algoritmos para inserção e pesquisa para a organização acima.

A eficiência do método de cálculo de endereço para a organização de tabelas, no caso de encadeamento, depende apenas do número médio de colisões por endereço gerado, que é igual ao comprimento médio das listas.

Já no caso de endereçamento aberto, a eficiência depende muito da taxa de ocupação da tabela, que é dada por:

$$tx = NO/(NO + NL)$$

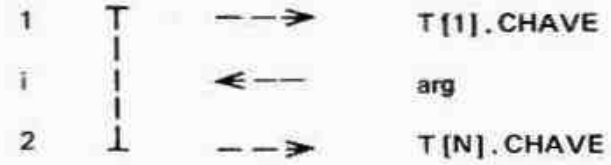
onde NO é o número de entradas ocupadas e NL o número de entradas livres (NO + NL = = n). Quanto menor o valor de tx, mais eficiente será a pesquisa. O número médio de comparações para a localização de uma particular entrada é dado por:

$$NC = (1 + 1/(1 - tx)) / 2$$

Assim sendo, é recomendável que seja reservado para a tabela um espaço correspondente a um número de entradas maior do que aquele que se pretende usar. O tamanho da folga que se irá deixar depende da eficiência em número de comparações que se deseja obter.

9.4 EXERCÍCIOS ADICIONAIS

1. O método da pesquisa por interpolação é semelhante ao da pesquisa binária. Sua idéia básica é usada quando se consulta um catálogo telefônico: abre-se o catálogo numa posição que depende do nome que estamos procurando. Se este começa por C, é perto do início; se começa por T, é perto do final. O método é o mesmo da pesquisa binária (cf. 9.2) só que arg é comparado com a chave localizada não no meio, mas na posição i, conforme sugerido pela figura



Descreva um procedimento para a busca por interpolação.

2. Examine as vantagens e desvantagens da busca por interpolação em confronto com a busca binária.

3. Compare métodos de organização de tabelas com métodos de representação de matrizes (esparsas) (cf. cap. 3).