A[i,j] contém o rótulo ou peso associado à aresta e, nesse caso, a matriz não é de bits. Se não existir uma aresta de i para j, então é necessário utilizar um valor que não possa ser usado como rótulo ou peso, tal como o valor 0 ou branco, conforme ilustra a Figura 7.7.

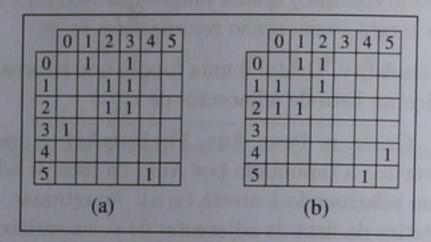


Figura 7.7 Representação por matrizes de adjacência. (a) Representação para o grafo direcionado da Figura 7.1(a); (b) Representação para o grafo não direcionado da Figura 7.1(b).

A representação por matrizes de adjacência deve ser utilizada para grafos densos, em que |A| é próximo de $|V|^2$. Nessa representação, o tempo necessário para acessar um elemento é independente de |V| ou |A|. Logo, essa representação é muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices. A maior desvantagem de usar matrizes de adjacência para representar grafos é que a matriz necessita $\Omega(|V|^2)$ de espaço. Isso significa que simplesmente ler ou examinar a matriz tem complexidade de tempo $O(|V|^2)$.

Em um tipo estruturado arranjo de duas dimensões, os itens são armazenados em posições contíguas de memória, e a inserção de um novo vértice ou retirada de um vértice já existente pode ser realizada com custo constante. O campo Mat é o principal componente do registro TipoGrafo mostrado no Programa 7.2. Os itens são armazenados em um array de duas dimensões de tamanho suficiente para armazenar o grafo. As constantes MaxNumVertices e MaxNumArestas definem o maior número de vértices e de arestas que o grafo pode ter.

Programa 7.2 Estrutura do tipo grafo implementado como matriz de adjacência

```
const MAXNUMVERTICES = 100;
    MAXNUMARESTAS = 4500;

type

TipoValorVertice = 0..MAXNUMVERTICES;
TipoPeso = integer;
TipoGrafo = record

Mat: array[TipoValorVertice, TipoValorVertice]
    of TipoPeso;
    NumVertices: 0..MAXNUMVERTICES;
    NumArestas : 0..MAXNUMVERTICES;
    NumArestas : 0..MAXNUMARESTAS;
    end;
TipoApontador = TipoValorVertice;
```

Uma possível implementação para as primeiras sete operações definidas anteriormente é mostrada no Programa 7.3.

Programa 7.3 Operadores sobre grafos implementados como matrizes de adjacência

```
procedure FGVazio (var Grafo: TipoGrafo):
var i, j: integer;
  for i := 0 to Grafo.NumVertices do
   for j := 0 to Grafo. NumVertices do
     Grafo.mat[i, j] := 0;
procedure InsereAresta (V1, V2: TipoValorVertice;
                       Peso : TipoPeso; var Grafo : TipoGrafo);
 Grafo.Mat[V1, V2] := peso;
function ExisteAresta (Vertice1, Vertice2: TipoValorVertice;
                     var Grafo: TipoGrafo): boolean;
  ExisteAresta := Grafo.Mat[Vertice1, Vertice2] > 0;
end; { ExisteAresta }
{- Operadores para obter a lista de adjacentes --}
function ListaAdjVazia (Vertice: TipoValorVertice;
                      var Grafo: TipoGrafo): boolean;
var Aux: TipoApontador; ListaVazia: boolean;
 ListaVazia := true; Aux := 0;
  while (Aux < Grafo.NumVertices) and ListaVazia do
   if Grafo.Mat[Vertice, Aux] > 0
   then ListaVazia := false
   else Aux := Aux + 1;
 ListaAdjVazia := ListaVazia = true;
end; { ListaAdjVazia }
function PrimeiroListaAdj (Vertice: TipoValorVertice;
                         var Grafo: TipoGrafo): TipoApontador;
var Aux: TipoApontador; ListaVazia: boolean;
 ListaVazia := true; Aux := 0;
 while (Aux < Grafo. NumVertices) and ListaVazia do
   if Grafo.Mat[Vertice, Aux] > 0
   then begin PrimeiroListaAdj := Aux; ListaVazia := false; end
   else Aux := Aux + 1;
 if Aux = Grafo. NumVertices
 then writeln ('Erro: Lista adjacencia vazia (PrimeiroListaAdj)');
end; { PrimeiroListaAdj }
```