Representações de Grafos

É possível diversas representações dos grafos. Vamos estudar as três utilizadas mais comumente: as matrizes de adjacência, as listas de adjacência e as multilistas de adjacência.

A escolha de determinada representação dependerá da aplicação que se tem em vista e das funções que se espera realizar no grafo.

A representação de listas de adjacências em geral é preferida porque ela fornece um modo compacto de representar grafos esparsos. Contudo, uma representação de matriz de adjacências pode ser preferível quando o grafo é denso, ou quando precisamos ter a possibilidade de saber com rapidez se existe uma aresta conectando dois vértices dados.

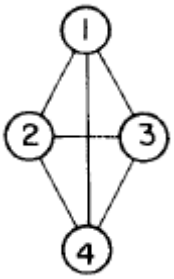
Matriz de Adjacências

Seja G = (V, E) um grafo com *n* vértices, onde $n \geq 1$. Supomos que os vértices são numerados 1, 2, ..., |V| de alguma maneira arbitrária. Então a representação de matriz de adjacências de um grafo G consiste em uma matriz $|V| \times |V|$ $A = (a_{ij})$ tal que

A(i,j) = 1 se a borda (vi, vj)({vi, vj} para um grafo dirigido[1]) fica em E(G).

A(i, j) = 0 se não houver tal borda em G.

A matriz de adjacência do grafo ao lado.



	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

A matriz de adjacências para um grafo não-dirigido[2] é simétrica, pois a borda (v_i, v_j) está em E(G), se a borda (v_i, v_j) também está em E(G). A matriz de adjacências de um grafo dirigido não necessita ser simétrica. É de n^2 bits o espaço necessário para representar um grafo, usando a matriz de adjacências.

A representação de matriz de adjacências pode ser usada no caso de grafos ponderados[3]. Por exemplo, se G = (V, E) é um grafo ponderado com função peso de aresta *w*, o peso $w(u, v)$ da aresta $(u, v) \in E$ é simplesmente armazenado como a entrada na fila u e na coluna v da matriz de adjacências.

A simplicidade de uma matriz de adjacências pode torná-la preferível quando os grafos são razoavelmente pequenos. Em lugar de usar uma palavra de memória de computador para cada entrada da matriz, a matriz de adjacências usa somente um bit por entrada.

---

[1] Num grafo dirigido cada borda é representada por um par dirigido (v_1, v_2), v_1 é o cabo e v_2 a cabeça da borda.

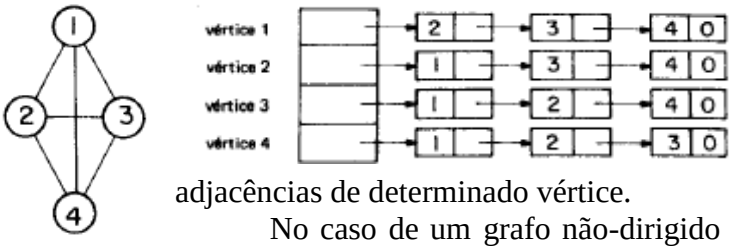
[2] Num grafo não-dirigido, não tem ordenação especial o par de vértices que representam qualquer borda

[3] Grafos nos quais cada aresta tem um peso associado a ela.

Listas de Adjacências

Nesta representação as *n* fileiras da matriz de adjacências são representadas como *n* listas interligadas. Existe uma lista para cada vértice em *G*. Os nós na relação *i* representam os vértices que são adjacentes do vértice *i*. Cada nó possui, pelo menos, dois campos: VERTEX e LINK.

Os campos VERTEX contém dois índices dos vértices adjacentes do vértice *i*. Cada relação possui um nó de cabeça. Os nós de cabeça são sequenciais, permitindo fácil acesso aleatório à lista de adjacências de determinado vértice.



No caso de um grafo não-dirigido com *n* vértices e bordas, essa representação requer *n* nós de cabeça a 2*e* nós de lista. Cada nó de lista possui dois campos. Em muitos casos, os nós podem ser condensados sequencialmente nas listas de adjacências eliminando-se os campos de ligação.

Uma Aplicação de Grafos

Suponha uma linha de entrada contendo quatro inteiros seguidos por qualquer número de linhas de entrada com dois inteiros cada uma. O primeiro inteiro na primeira linha, *n*, representa um número de cidades, que, para simplificar, serão numeradas de 0 a *n-1*. O segundo e o terceiro inteiro nessa linha estão entre 0 e *n-1* e representam duas cidades. Queremos sair da primeira cidade para a segunda usando exatamente *nr* estradas, onde *nr* é o quarto inteiro na primeira linha de entrada. Cada linha de entrada subsequente contém dois inteiros representando duas cidades, indicando que existe uma estrada da primeira cidade até a segunda. O problema é determinar se existe um percurso do tamanho solicitado pelo qual se possa viajar da primeira cidade para a segunda.

Uma solução é criar um grafo com as cidades como nós e as estradas como arcos.

Para achar um caminho de comprimento *nr* do nó A ao nó B, procure um nó C de modo que exista um arco de A até C e um caminho de comprimento *nr-1* de C até B. Se essas condições forem atendidas para um nó C, o caminho desejado existirá; se elas não forem atendidas para qualquer nó C, o caminho não existirá.

Representações de Grafos em C

Os vértices de um grafo serão representados por números inteiros no intervalo `0..v-1`. O conjunto de vértices será, portanto, `0 1 2 3...v-1`.

Vértices são representados por objetos do tipo `Vertex`.

```
#define Vertex int
```

Os arcos dos grafos serão representados por `structs` do tipo `Edge`. Um objeto do tipo `Edge` representa um arco com ponta inicial `v` e ponta final `w`.

```
typedef struct{
      Vertex v;
      Vertex w;
} Edge;
```

Se `e` é um arco então `e.v` é a ponta inicial e `e.w` é a ponta final do arco. A construção de arcos ficará a cargo de uma função `EDGE`. A função `EDGE` recebe dois vértices `v` e `w` e devolve um arco com ponta inicial `v` e ponta final `w`.

```
Edge EDGE (Vertex v, Vertex w){
      Edge e;
```

```
        e.v = v;
        e.w = w;
        return e;
}
```

Um grafo será representado por uma `struct graph` que contém o número de vértices, o número de arcos e a matriz de adjacências do grafo. O campo `adj` é um ponteiro para a matriz de adjacência do grafo. O campo `V` contém o número de vértices e o campo `E` contém o número de arcos do grafo.

```
struct graph{
        int V;
        int E;
        int **adj;
};
```

Um objeto do tipo `Graph` contém o endereço de um `graph`.

```
typedef struct graph *Graph;
```

Funções básicas

Esta função devolve o endereço de um novo grafo com vértices `0,..,V-1` e nenhum arco.

```
Graph GRAPHinit (int V){
        Graph G = malloc(sizeof *G);
        G->V = V;
        G->E = 0;
        G->adj = MATRIXint(V, V, 0);
        return G;
}
```

A função abaixo aloca uma matriz com linhas `0..r-1` e colunas `0..c-1`. Cada elemento da matriz recebe valor `val`.

```
int **MATRIXint (int r, int c, int val){
        Vertex i, j;
        int **m = malloc(r* sizeof(int *));
        for (i = 0; i < r; i++)
                m[i] = malloc(c* sizeof(int));
        for (i = 0; i < r; i++)
                for (j = 0; j < c; j++)
                        m[i][j]= val;
        returm m;
}
```

Esta função insere um arco `v-w` no grafo `G`. Se `v == w` ou o grafo já tem arco `v-w`, a função nada faz. `v` e `w` não podem ser negativos e deve ser menores que `G->V`.

```
void GRAPHinsertE (Graph G, Vertex v, Vertex w){
        if (v != w && G->adj[v][w] == 0){
                G->adj[v][w] = 1;
                G->E++;
        }
}
```

Esta função remove do grafo `G` o arco que tem ponta inicial `v` e ponta final `w`. Se não existe tal arco, a função não faz nada.

```
void GRAPHremoveE (Graph G, Vertex v, Vertex w){
        if (G->adj[v][w] == 1){
                G->E--;
                G->adj[v][w] = 0;
        }
}
```

Para cada vértice `v` do grafo `G`, esta função imprime, em uma linha, todos os vértices adjacentes a `v`.

```
void GRAPHshow (Graph G){
```

```
        Vertex v, w;
        for (v = 0; v < G->V; v++){
                printf("%2d:", v);
                for (w = 0; w < G->V; w++)
                        if (G->adj[v][w] == 1)
                                printf(" %2d", w);
                printf("\n");
        }
}
```

Vetor de listas de adjacência

O vetor de listas de adjacência de um grafo tem uma lista encadeada para cada vértice do grafo. A lista do vértice `v` contém todos os vértices vizinhos de `v`.

Listas encadeadas

Uma lista encadeada é uma representação de uma sequência de objetos na memória do computador. Cada elemento da sequência é armazenado em uma célula da lista: o primeiro elemento na primeira célula, o segundo na segunda e assim por diante.

Uma lista encadeana é uma sequência de células; cada célula contém um objeto de algum tipo e o endereço da célula seguinte.

Exemplo do conjunto de arestas de um grafo e o vetor de listas de adjacências do grafo:

```
0-1 0-5 1-5 2-4 3-1 5-3
  0     :     5     1
  1     :     3     5     0
  2     :     4
  3     :     5     1
  4     :     2
  5     :     3     1     0
```

Exemplo de um dígrafo e seu vetor de listas de adjacência:

```
0-1 0-5 1-0 1-5 2-4 3-1 5-3
  0     :     5     1
  1     :     5     0
  2     :     4
  3     :     1
  4     :
  5     :     3
```

Estruturas

Um dígrafo é representado por uma `struct digraph` que contém o número de vértices, o número de arcos e o vetor de listas de adjacência do dígrafo.

A estrutura `digraph` represente um dígrafo. O campo `adj` é um ponteiro para o vetor de listas de adjacência do dígrafo. O campo `V` contém o número de vértices e o campo `A` contém o número de arcos do dígrafo.

```
struct digraph {
        int V;
        int A;
        link *adj;
};
```

Um objeto do tipo Digraph contém o endereço de um digraph.

```
typed struct digraph *Digraph;
```

A lista de adjacência de um vértice `v` é composta por nós do tipo `node`. Um `link` é um ponteiro para um `node`. Cada nó da lista contém um vizinho `w` e `v` e o endereço do nó seguinte da lista.

```
typedef struct node *link;
struct node {
      Vertex w;
      link next;
};
```

A função `NEW` recebe um vértice `w` e o endereço `next` de um nó e devolve um novo nó `x` com `x.w = w` e `x.next = next`.

```
link NEW (Vertex w, link next){
      link x = malloc(sizeof *x);
      x->w = w;
      x->next = next;
      return x;
}
```

Essa mesma estrutura é usada para representar grafos, nesse caso escrevemos `graph` e `Graph` no lugar de `digraph` e `Digraph`.

Essas estruturas não devem ser consideradas definitivas. Elas poderão ser modificadas e adaptadas conforme as necessidades.

Funções básicas

Algumas funções básicas de construção e manipulação de dígrafos representados por listas de adjacência:

```
Digraph DIGRAPHinit (int V){
      Digraph G = malloc(sizeof *G);
      G->V = V;
      G->A = 0;
      G->adj = malloc(V * sizeof(link));
      for (v = 0; v < V; v++)
            G->adj[v] = NULL;
      return G;
}
```

A função acima devolve o endereço de um novo dígrafo com vértices 0, ..., V-1 e nenhum arco.

```
void DIGRAPHinsertA (Digraph G, Vertex v, Vertex w){
      link p;
      if (v == w) return;
      for (p = G->adj[v]; p != NULL; p = p->next)
            if (p->w == w) return;
      G->adj[v] = NEW(w, G->adj[v]);
      G->A++;
}
```

A função acima insere um arco `v-w` no dígrafo `G`. Se `v == w` ou o dígrafo já tem arco `v-w`, a função não faz nada.

A execução da função `DIGRAPHinsertA` consome muito tempo, pois percorre toda a lista de vizinhos de `v`.

A função abaixo transfere a responsabilidade de evitar laços e arcos paralelos ao usuário, que provavelmente tem condições de fazer isso de maneira eficiente.

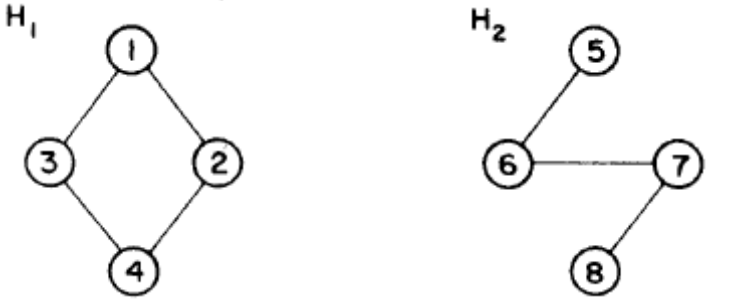
```
void DIGRAPHinsertA (Digraph G, Vertex v, Vertex w){
      if (v == w) return;
      G->adj[v] = NEW(w, G->adj[v]);
      G->A++;
}
```

Exercícios

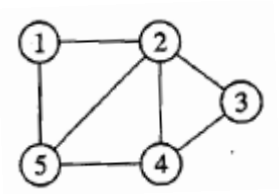
1. Dados os grafos a seguir represente as matrizes de adjacências.

a)

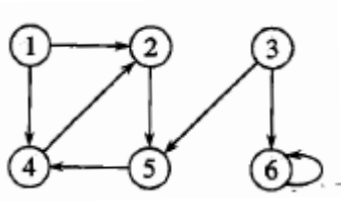
b)



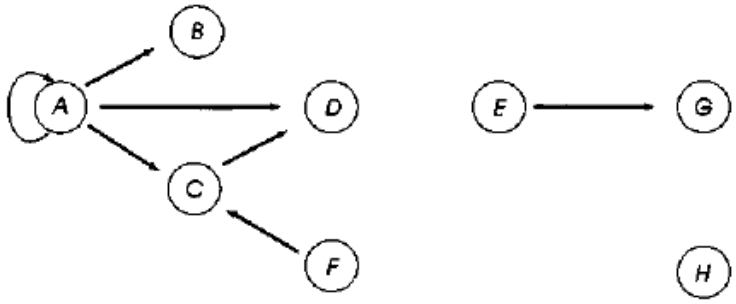
c)



d)

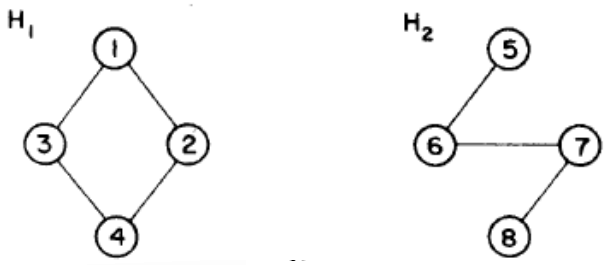


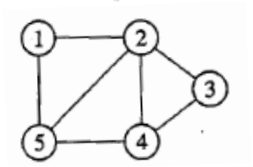
e)

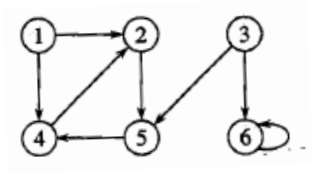


2. Dados os grafos abaixo represente as listas de adjacências.

a)

b) 

c) 

d) 

3. Escreva uma função `GRAPHedges` que armazene as arestas de um grafo num vetor fornecido pelo usuário.
4. Digamos que um arquivo é *grafico* se sua primeira linha contém um inteiro `V` e cada uma das demais linhas contém dois inteiros pertencentes ao intervalo `[0..V-1]` se parados por um caractere '`-`'. Eis o exemplo:

```
6
0-1
0-5
1-0
1-5
2-4
3-1
```

   Se interpretarmos cada linha do arquivo como uma aresta, podemos dizer que o arquivo define um grafo com vértices 0..V-1. Escreva uma função GRAPHconstruct que receba um arquivo gráfico e construa a matriz de adjacência do grafo. Use as funções GRAPHinit e GRAPHinsertE.
5. Siga as orientações acima para o mesmo exemplo e construa a lista de adjacência.
6. A figura abaixo sugere o vetor de listas de adjacência de um grafo. De quantas maneiras diferentes essas listas poderiam ser reescritas sem deixar de representar o mesmo grafo?

```
0  :     6     5     1     2
1  :     0
2  :     0
3  :     5     4
4  :     6     5     3
5  :     3     0     4
6  :     0     4
7  :     8
9  :     12    11    10
10 :     9
11 :     12    9
12 :     9     11
```

7. Escreva uma função que receba um vetor de arestas e devolva a representação por listas de adjacência do grafo definido por essas arestas.
8. Escreva uma função que receba um grafo representado por matriz de adjacência e construa a correspondente representação por listas de adjacência.

9. Escreva uma função que receba um grafo representado por listas de adjacência e construa a correspondente representação por matriz de adjacência.
10. Uma grade m-por-n é um grafo com m×n vértices distribuídos em m linhas e n colunas com arestas ligando vértices vizinhos na horizontal e na vertical da maneira óbvia. A figura ao lado sugere uma grade 3-por-4.

Escreva uma função que construa uma grade m×n. Faça duas versões: uma gera uma representação por matriz de adjacência e outra gera uma representação por listas de adjacência.

Bibliografia

Estrutura de Dados Usando C
Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein
São Paulo: Makron Books, 1995

Algoritmos: Teoria e Prática
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
Rio de Janeiro: Elsevier, 2002

Estrutura de Dados
Paulo Veloso, Clesio dos Santos, Paulo Azevedo, Antonio Furtado
Rio de Janeiro: Campus, 1983

Fundamentos de Estrutura de Dados 253
Ellis Horowitz, Sartaj Sahni
Rio de Janeiro: Campus, 1986

Uma Introdução Sucinta à Teoria dos Grafos 8
http://www.ime.usp.br/~pf/teoriadosgrafos/
P. Feofiloff, Y. Kohayakawa, Y. Wakabayashi

Algoritmos para Grafos em C via *Sedgewick*
http://www.ime.usp.br/~pf/algoritmos_para_grafos/
Paulo Feofiloff