Grafos – conceitos básicos e técnicas de percurso

Grafo é um tópico “unificador” em Ciência da computação, pela sua capacidade, através de uma representação abstrata, de se ajustar às mais variadas aplicações.

Um grafo pode descrever:

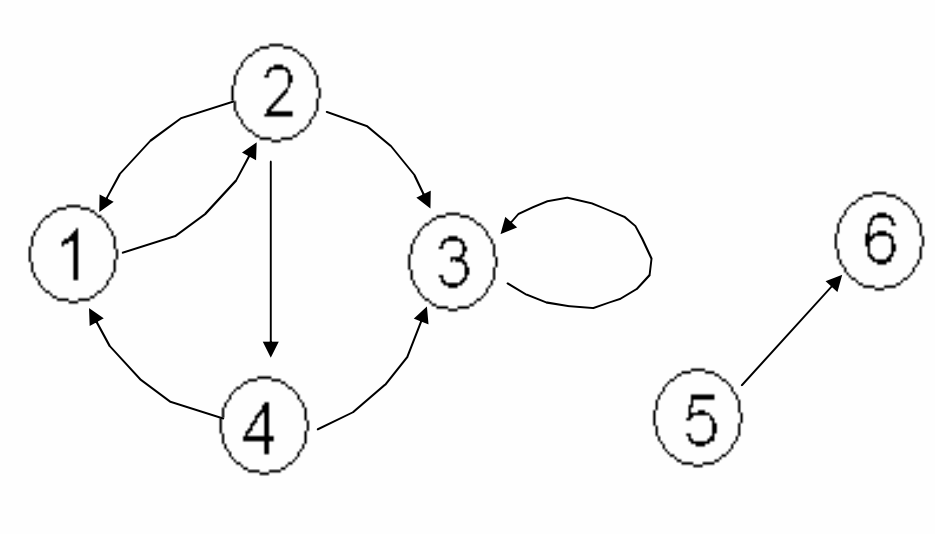
- sistemas de transporte;
- circuitos elétricos;
- interações humanas;
- redes de telecomunicação;
- etc.

1 Conceitos básicos de Grafos

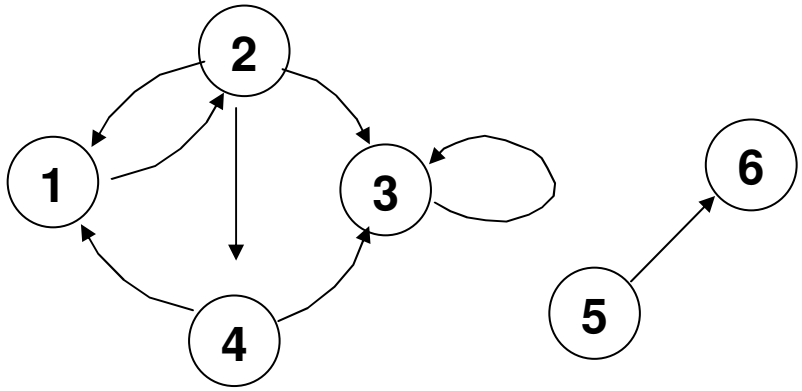
Um grafo $G = (N,A)$ é definido por:

- Um conjunto N de ***nós (ou vértices)***;
- Um conjunto A de ***arcos (ou arestas)***, consistindo de pares (ordenados ou não-ordenados) de vértices pertencentes a *N*.

Exemplo:



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



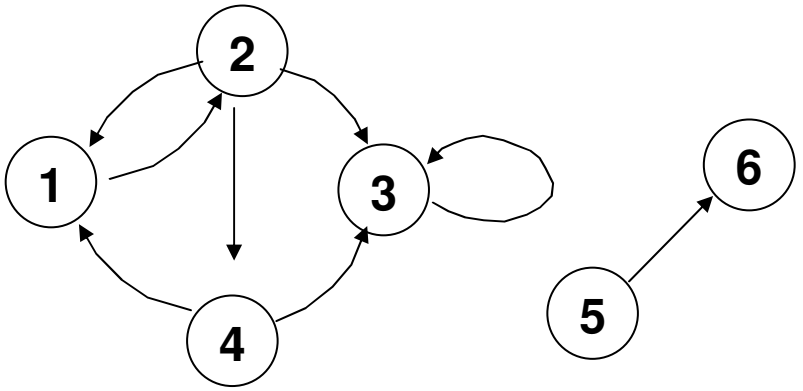
Nós ligados por arcos são ditos **adjacentes**: o nó 2 é adjacente a 1,3 e 4.

Arcos são **incidentes** de (ou a) determinados nós, conforme partam (ou cheguem) a eles: (1,2) é incidente de 1 (e a 2).

Um **caminho** é uma seqüência de um ou mais arcos...

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

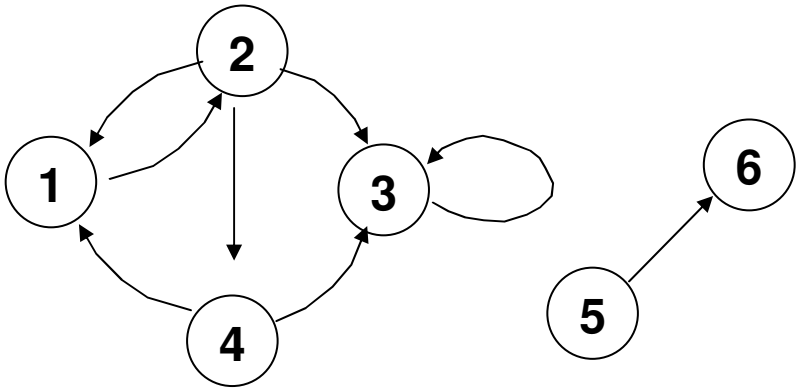
...que permite que, a partir de um nó **a**, se atinja o nó **b**.



Exemplos:

- embora 1 e 3 não sejam conectados, pode-se, a partir de 1, atingir 3, percorrendo os arcos (1,2) e (2,3).
- De 4 para 3, temos o caminho (4,3) e o caminho (4,1),(1,2) e (2,3).

  Observe que, no segundo exemplo, o caminho mais longo não é necessariamente o “pior” (pense em uma rede de estradas...).

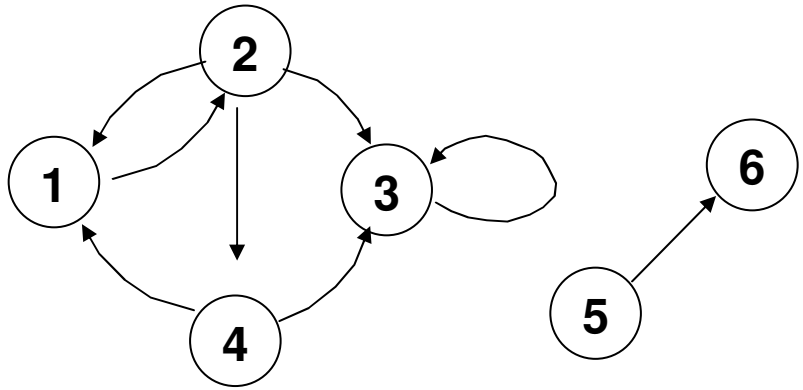


Em um caminho...

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

...quando a=b, temos um **circuito ou ciclo**. Ex: (1,2), (2,4), (4,1).

Um circuito de um único arco é um **laço**. Ex: (3,3).



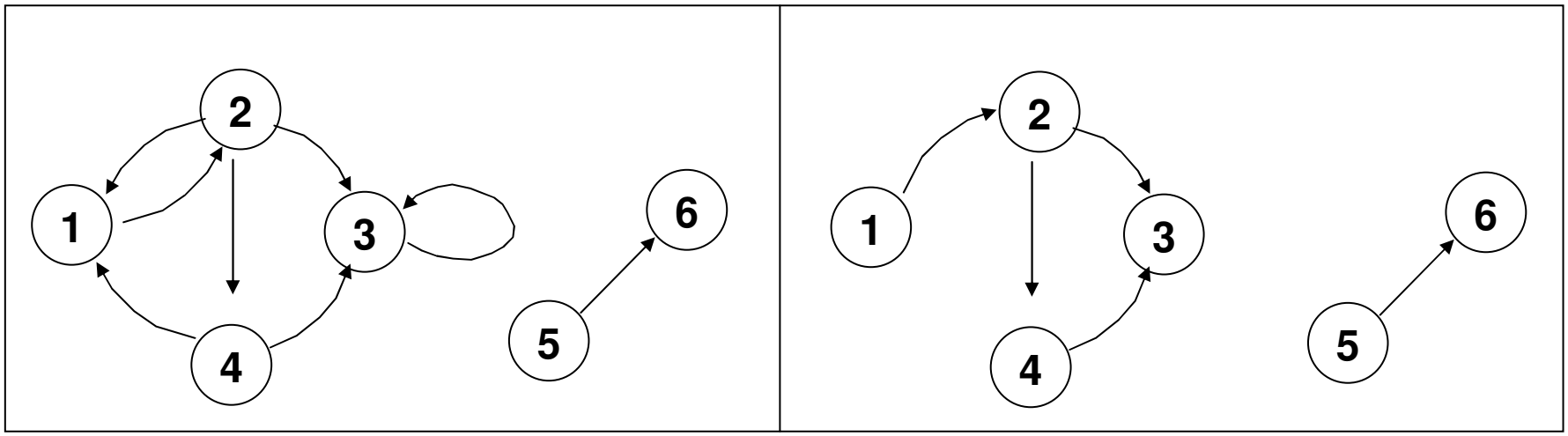
Um grafo é **conexo** quando possui um nó a partir do qual há caminhos para todos os demais.

>> O grafo acima não é conexo. Entretanto, o **subgrafo** contituído pelos nós 1,2,3,4 é conexo.

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

Um grafo é dito **fortemente conexo** se de todos os nós pode-se atingir todos os demais.

>> O subgrafo citado (1,2,3,4) não é fortemente conexo



a b

Em um **grafo parcial**, permanecem todos os nós do grafo original, mas é tomado um subconjnto de seus arcos

>> na figura acima, em b, tem-se um grafo parcial do primeiro.

Uma das propriedades do grafo parcial acima obtido é que ele é **acíclico**: não contém nenhum circuito.

Potencial de modelagem:

Na modelagem de uma **rede de estradas**, os nós podem representar cidades (e junções de estradas), onde alguns pares são conectados por estradas.

Na análise de **interações humanas**, os nós tipicamente representam pessoas, e os arcos conectam pares de pessoas relacionadas.

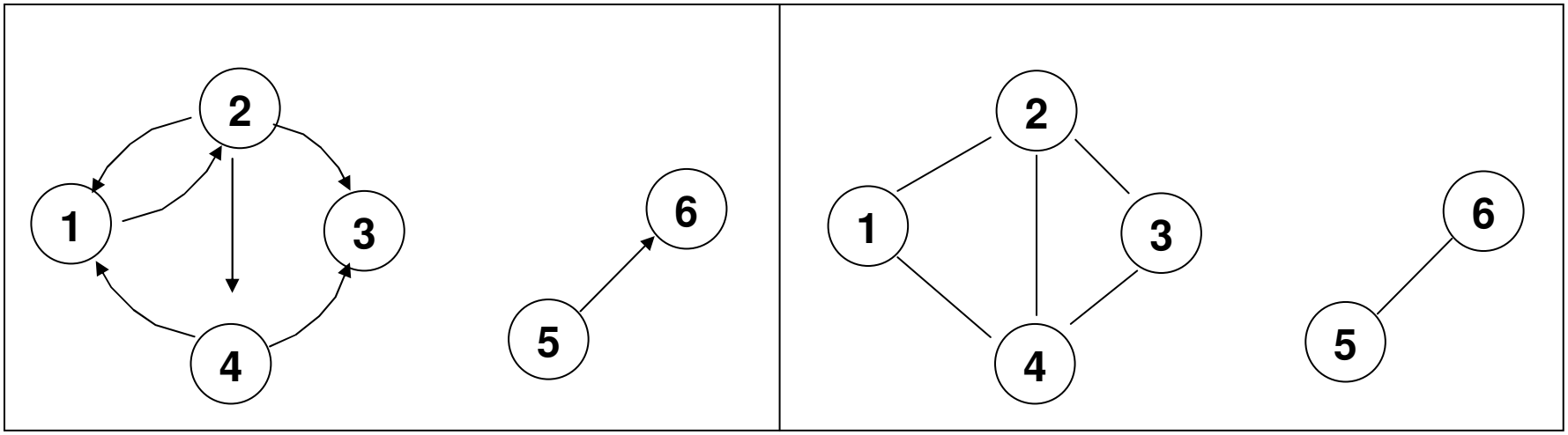
E assim sucessivemente...

Propriedades fundamentais dos grafos

- ***Dirigido x não-dirigido***

Um grafo $G = (N,A)$ é *não-dirigido* se: arco $(x, y) \in A$ implica que (y, x) também pertence a A.

Caso contrário, dizemos que o grafo é *dirigido*.



a b

Um grafo dirigido (em a) e outro não dirigido (em b)

Exemplos de situações

Redes de estradas *entre* cidades: tipicamente, um grafo não-dirigido: estradas em geral são de mão-dupla.

Redes de ruas e*m uma cidade: quase sempre um grafo* dirigido: algumas ruas podem ser de mão-única.

Em geral:

Relações simétricas levam a grafos não dirigidos. Ex: “ser parente de”.

Já relações que não são simétricas (chefiar, ajudar, etc.) levam a grafos dirigidos.

• *Valorados x não-valorados*

Em grafos valorados, a cada arco (ou nó) de G é associado um valor numérico, ou peso.

Em grafos não-valorados, não há distinção valorativa entre os vários nós e arcos.

Para determiner o caminho mais curto entre dois nós...

- Para grafos não valorados, o caminho mais curto é o que tem o menor número de arcos.
- Grafos valorados requerem algoritimos mais sofisticados.

- ***Simples x Complexo***

Certos tipos de arcos tornam mais complexa a tarefa de trabalhar com grafos.

Laços (*self-loop):* arcos (*x, x*) envolvendo um único vértice.

Um arco (*x, y*) é um *multi-arco* se ele ocorre mais de uma vez em um grafo.

Essas particularidades requerem cuidado especial na implementação de algoritmos...

OBS: as características acima discutidas influenciam a escolha das estruturas de dados usadas para melhor representá-los.

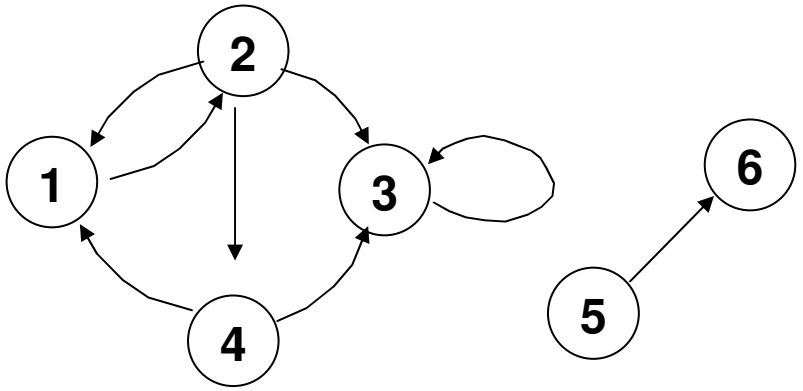
2 Estruturas de dados para grafos

Seja o grafo $G = (N, A)$ com ***n*** nós e m arcos.

• ***Matriz de adjacência*** — representa-se G *por* uma matriz M de ordem ***n×n***, onde o elemento $M[i, j]$ é V, se (i, j) é um arco de G, e F caso contrário.

Obs: é comum empregar-se 1 ou 0 ao invés de V ou F.

Para o grafo ao lado 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						

- rápida resposta à pergunta "(*i, j*) pretence a *G*?"
- ágil atualização quando da inserção e exclusão de arcos.

Desvantagem: pode usar espaço excessivo de memória para grafos com muitos nós e relativamente poucos arcos...

Considere um grafo representando o mapa de ruas de Manhattan (Nova York). Cada junção de duas ruas sera um nó do grafo, com junções vizinhas conectadas por arcos.

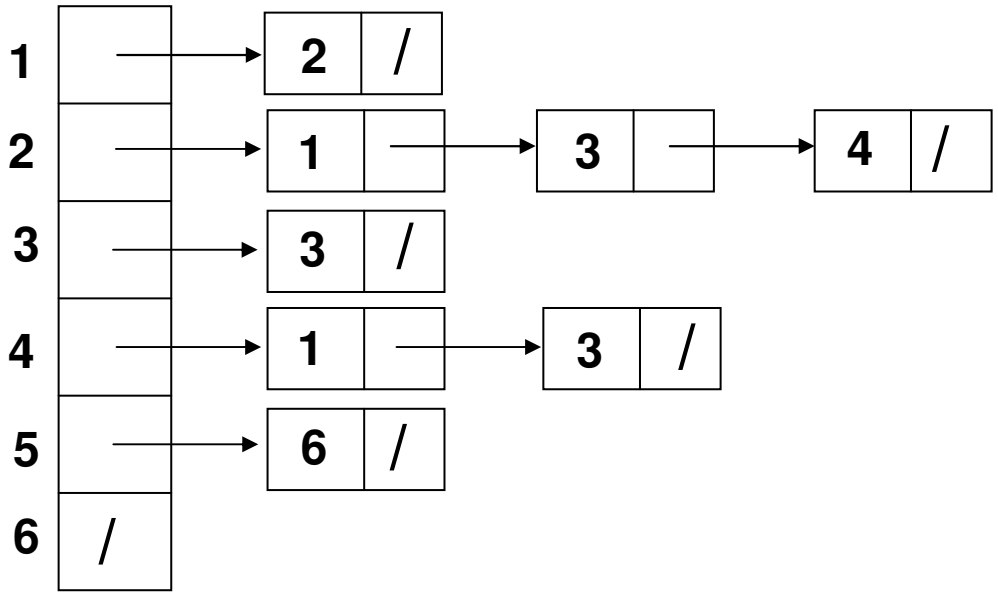
Manhattan possui 15 avenidas, que cortam em torno de 200 ruas. Aproximadamente...

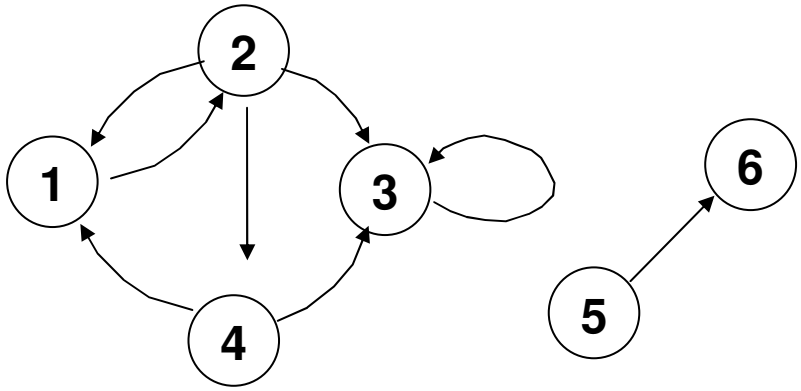
- 3,000 nós
- 6,000 arcos

A matriz de adjacência terá 3.000 × 3.000 = 9.000.000 de células (quase todas vazias!)

☞ Para um número muito grande de nós, essa representação pode ser problemática!

• ***Listas de adjacência (por encadeamento)*** — representam mais eficientemente grafos esparsos empregando listas encadeadas para armazenar os adjacentes de cada nó.





Obs: a segunda lista representa os arcos (2,1), (2,3) e (2,4) e não (2,1), (1,3) e (3,4)...

Listas de adjacência tornam mais complexa a tarefa de responder se um dado arco (*i, j*) pertence a *G*, já que temos que percorrer a lista apropriada...

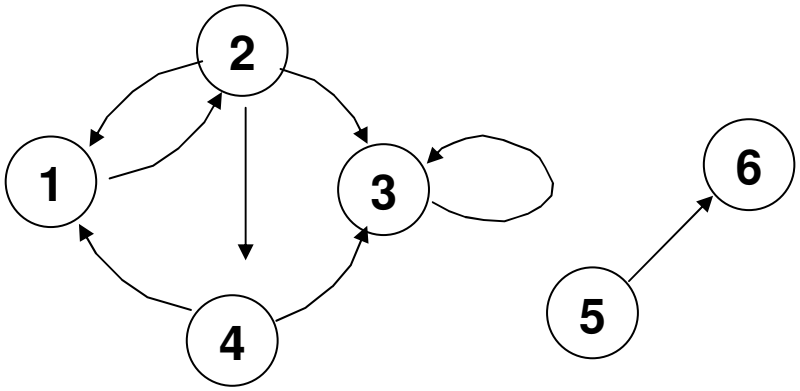
Contudo, há técnicas apropriadas para contornar esse problema.

- ***Listas de adjacência em matrizes***

Podemos representar uma lista em um vetor (uma linha de uma matriz)...

Dimensionamento da matriz: uma linha para cada nó, e o número de colunas igual ao maior número de vértices mais 1.

	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



Obs: o primeiro zero encontrado indica o fim da lista...

OBS:

- não é tão simples determinar se um arco (*i, j*) pertence a *G.*
- se um único nó tiver muitas ligações, isso levará a um grande número de colunas;
- pode limitar a inserção de novos arcos.

	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

- uma variante dessa representação é manter um contador k para cada linha com o número dos elementos, que estarão armazenados nas primeiras k posições.

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

	1	2	3	4
1	**2**			
2	**1**	**3**	**4**	
3	**3**			
4	**1**	**3**		
5	**6**			
6				

De forma geral essa estrutura parece combinar as piores propriedades das matrizes de adjacência (muito espaço) e das litas de adjacência (dificil localização dos arcos). Ainda assim podem se prestar a aplicações específicas (especialmente grafos estáticos).

• *Tabela de Arcos* — estrutura mais simples, que simplesmente mantém um vetor ou lista encadeada dos arcos.

Assim como a estrutura anterior…

- Não facilita responder a pergunta “quem é adjacente ao vértice x?”
- Mas pode se prestar a aplicações específicas simples.

• *Matrizes de incidência* — indica, em cada coluna, os dois nós correspondentes a cada uma das arestas (que são representadas por números inteiros).

	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**

OBS: note o laço (3,3) na coluna 5.

Representação de grafos valorados:

a) Caso de valores correspondentes aos arcos

É possível adaptar todas as estruturas anteriores. Ex:

- matriz de adjacência: colocar o valor do arco de i para j na posição M[i,j], e não simplesmente *true* ou *false*.
- listas de adjacência: adicionar a cada célula um terceiro componente, valor, do tipo adequado.
- matriz de incidência: para um arco k do nó i para o nó j, com valor V, fazer a posição M[i,k] = -V e M[j,k] = V

b) valores correspondentes aos nós: manter um vetor (ou lista encadeada) para guardar o(s) valor(es) associado(s) a cada nó.

Exemplo de representação:

- matriz de adjacência

```
const int MxNmNos = 30;
typedef int MtAdj[MxNmNos][MxNmNos];
bool visit[MxNmNos];
```

Obs:

- para grafos não dirigidos, a matriz deverá ser simétrica em relação à DP.
- o emprego da linha 0 e coluna 0, dependendo da aplicação poderá não ser direto (possibilidade: ignorar essas partes da matriz)

Operações básicas

```
void inicGrafo(MtAdj mat)
{
    // inicializar matriz de adjacencias
    memset(mat, 0, sizeof(MtAdj));    // att

    // inicializar vetor de controle de percurso
    memset(visit, false, sizeof(visit));
}
```

```
void leGrafo(MtAdj mat, int nmArcos, bool dirig)
{
     int i,v1,v2;

     inicGrafo(mat);
     for (i=0; i< nmArcos; i++)
     {
        cin >> v1 >> v2;
        mat[v1][v2] = 1;
        if (! dirig)
           mat[v2][v1] = 1;
     }
}
```

3 Percursos em grafos

Idéia básica: percorrer o grafo completa e sistematicamente.

-> “visitar” vértices (e arcos) em alguma ordem bem estabelecida.

Obs: “visitar” pode se referir às mais variadas ações, como imprimir, contar, etc.

Há dois principais algoritmos de percurso: em amplitude (*breadth-first search* - BFS) e em profundidade (*depth-first search* - DFS).

Em ambos os casos, parte-se de um nó V escolhido arbitrariamente (“raiz”) e “visita-se” o nó; em seguida, considera-se cada um dos nós W_i adjacentes a V.

I) No percurso em amplitude:

a) visita-se o nó W_i;
b) coloca-se o nó W_i em uma fila
c) ao se terminar de visitar os nós W_i, toma-se o nó que estiver na frente da fila e repete-se o processo...

Em ambos os casos, parte-se de um nó V escolhido arbitrariamente (“raiz”) e “visita-se” o nó; em seguida, considera-se cada um dos nós W_i adjacentes a V.

I) No percurso em amplitude:

d) visita-se o nó W_i;
e) coloca-se o nó W_i em uma fila
f) ao se terminar de visitar os nós W_i, toma-se o nó que estiver na frente da fila e repete-se o processo...

II) No percurso em profundidade:

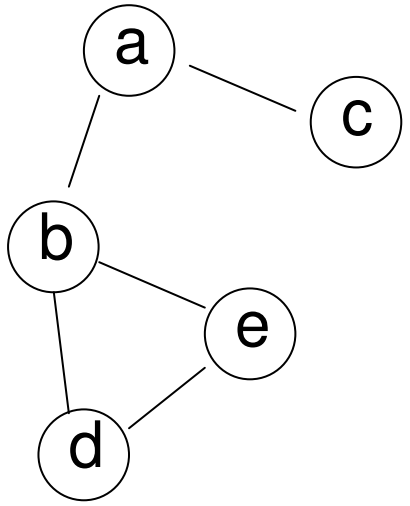
a) visita-se o nó W;
b) coloca-se o nó V em uma pilha;
c) faz-se V ← W

Com o passo “c”, o processo de considerar nós adjacentes ao nó v (original) é interrompido...

Efeito do percurso em amplitude:

Visita-se a “raiz”, todos os nós separados por um arco da “raiz”, depois todos os separados por dois arcos, e assim sucessivamente.

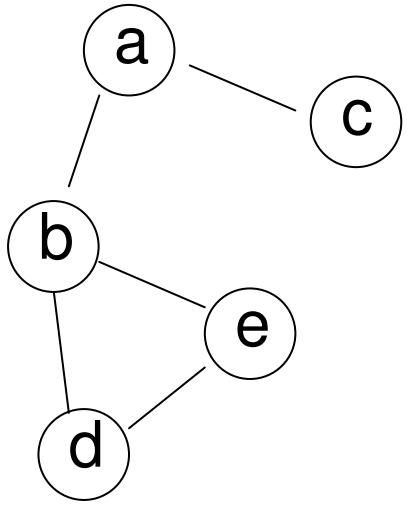
Ex: partindo-se de a, percorreríamos:
a, b, c, d, e



Efeito do percurso em profundidade:

Visita-se a raiz, um dos adjacentes à raiz, e parte-se para nós cada vez mais distantes da “raiz”, que vão sendo empilhados; ao não haver como continuar, toma-se o nó do topo da pilha...

Ex: partindo-se de a, percorreríamos:
a, b, d, e, c



Obs: nós já visitados são marcados.

OBS:

1) para certos problemas, não faz qualquer diferença em usar um ou outro método; já em outros casos a distinção é crucial.

Ex: para a determinação de menor caminho em grafos não valorados o método em amplitude é o indicado.

2) em ambos os métodos, é necessário marcar os nós já visitados, para não explorá-los novamente (em geral, um vetor para essa finalidade é usado).

3.1 Percurso em profundidade (versão recursiva)

Será usado um vetor de booleanos para registrar a situação de cada vértice no grafo (visitado ou não).

```
void processarNo(int i);

void PercProfund(MtAdj mat, int nmNos, int part)
{
  int i;

  processarNo(part);
  visit[part] = true;
  for(i=1; i <= nmNos; ++i)
    if (mat[part][i] == 1 && ! visit[i])
      PercProfund(mat,nmNos,i);
}
```

```
void processarNo(int i)
{
      cout << i << ' ';
}
```

Versão não-recursiva:

Estruturas/funções auxiliares

```
const int MxNmNos = 30;
const int TamPilha = 30;
typedef int MtAdj[MxNmNos][MxNmNos];
bool visit[MxNmNos];

typedef struct {
        int elem[TamPilha];
        int topo;
        } Pilha;
```

```
void InicializaPilha(Pilha &P)
{
   P.topo = 0;
}

void Empilhar(Pilha &P, int v)
{
  if (P.topo < TamPilha)
  {
    P.elem[P.topo] = v;
    P.topo++;
  }
}
void Desempilhar(Pilha &P)
{
  if (P.topo >0)
     P.topo--;
}
```

```
int ElemTopo(Pilha P)
{
    return P.elem[P.topo-1];
}
```

```
void PercProfundidade(MtAdj mat, int nmNos, int part)
{
    int i;
    Pilha P;
    int no;

    InicializaPilha(P);
    processarNo(part);
    visit[part] = 1;
    Empilhar(P,part);
```

```
    while (P.topo > 0)    // pilha não vazia
    {
      no = ElemTopo(P);
      Desempilhar(P);
      for(i=1; i<=nmNos; i++)
      {
         if ( (mat[no][i] == 1) && ! visit[i])
         {
             processarNo(i);
             visit[i] = 1;
             Empilhar(P,no);
             Empilhar(P,i);
             break;
         }
      }
    }
}
```

```
int main(int argc, char *argv[])
{
  int nmNos, nmArc, noPart;
  MtAdj mat;

  cin >> nmNos >> nmArc >> noPart;
  while (nmNos !=  0)
  {
    leGrafo(mat,nmArc,false); // grafo nao dirigido
    PercProfund(mat,nmNos,noPart);
    cout << endl;
    cin >> nmNos >> nmArc;
  }
  return 0;
}
```

3.2 Percurso em amplitude

Estruturas/funções auxiliares

```
typedef struct {
        int elem[TamFila];
        int ini,fim, tam;
        } Fila;

void InicializaFila(Fila &F)
{
  F.fim = 0;
  F.ini = 1;
  F.tam = 0;
}
```

```
void Enfileirar(Fila &F, int v)
{
  if (F.tam <TamFila)
  {
    F.fim = (F.fim+1) % TamFila;
    F.elem[F.fim] = v;
    F.tam++;
  }
}

void Desenfileirar(Fila &F)
{
  if (F.tam >0)
  {
    F.ini = (F.ini+1) % TamFila;
    F.tam--;
  }
}
```

```
int Frente(Fila F)
{
    return F.elem[F.ini];
}


void PercAmplitude(MtAdj mat, int nmNos, int part)
{
   int i;
   Fila F;
   int no;
   InicializaFila(F);
   Enfileirar(F,part);
   processarNo(part);
   visit[part] = true;
```

```
    while (F.tam > 0)
    {
      no = Frente(F);
      Desenfileirar(F);
      for(i=1; i<=nmNos; i++)
      {
         if ( (mat[no][i] == 1) && ! visit[i])
         {
            processarNo(i);
            visit[i] = true;
            Enfileirar(F,i);
         }
      }
    }
}
```