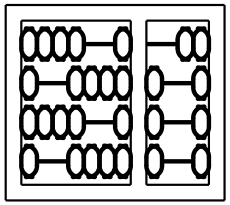


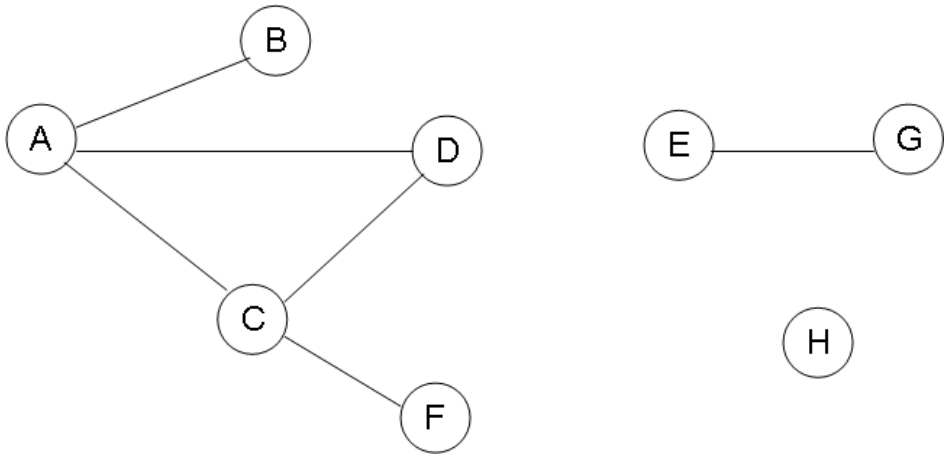
Universidade Estadual de Campinas - UNICAMP
Instituto de Computação - IC

Grafos

Tópicos

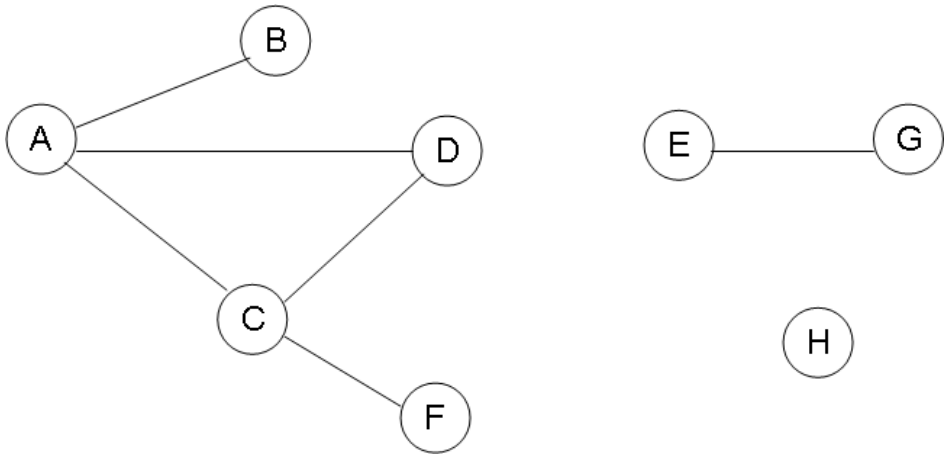
• Definição: Um grafo é uma estrutura de dados não-linear constituída de um conjunto de vértices ou nós, V, e um conjunto de arestas ou arcos, A, conectando pares de vértices. Cada arco é especificado por um par de nós.

Exemplo:



• Definição: Um grafo é uma estrutura de dados não-linear constituída de um conjunto de vértices ou nós, V, e um conjunto de arestas ou arcos, A, conectando pares de vértices. Cada arco é especificado por um par de nós.

Exemplo:

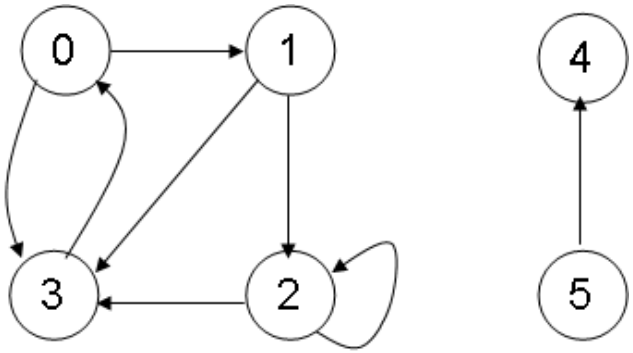


– Sequência de nós: $\{A, B, C, D, E, F, G, H\}$
– Sequência de arestas: $\{(A,B), (A,D), (A,C), (C,D), (C,F), (E,G)\}$

– Um grafo direcionado (ou dígrafo) G é um par (V,A), em que V é um conjunto finito de vértices e A é um conjunto de arestas com uma relação binária em V.

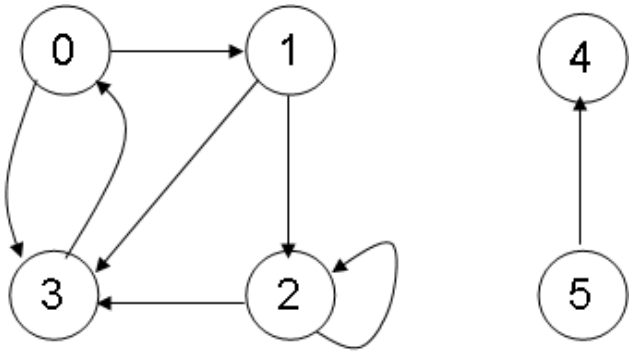
– Um grafo direcionado (ou dígrafo) G é um par (V,A), em que V é um conjunto finito de vértices e A é um conjunto de arestas com uma relação binária em V.

Exemplo:
V = $\{0, 1, 2, 3, 4, 5\}$
A = $\{ (0,1), (0,3), (1,2), (1,3), (2,2), (2,3), (3,0), (5,4)\}$



– Um grafo direcionado (ou dígrafo) G é um par (V,A), em que V é um conjunto finito de vértices e A é um conjunto de arestas com uma relação binária em V.

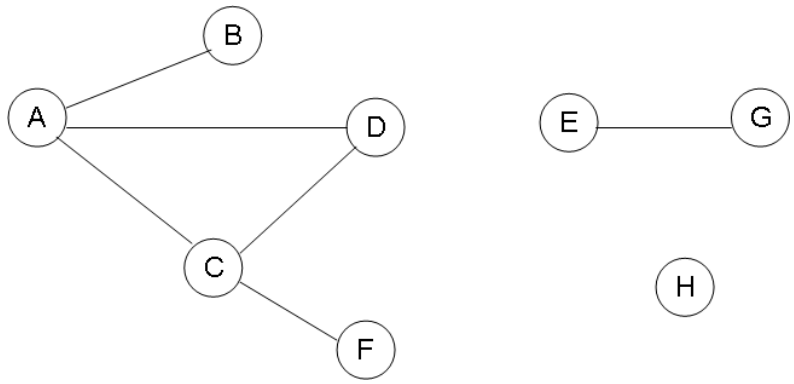
Exemplo:
V = {*0, 1, 2, 3, 4, 5* }
A = { *(0,1), (0,3), (1,2), (1,3), (2,2), (2,3), (3,0), (5,4)*}



– Se (u, v) é uma aresta no grafo $G = (V, A)$, o vértice v é dito **adjacente** ao vértice u.

• O **grau de um vértice** em um grafo não direcionado é o número de arestas que nele incidem.

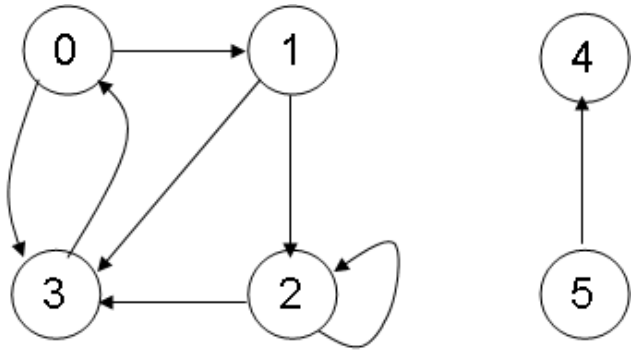
Exemplo:



– grau(C) = 3;
– grau(H) = 0 (**vértice isolado** ou **não conectado**)

• O **grau de um vértice** em um grafo direcionado é o número de arestas que saem do vértice (**grau de saída**) mais o número de arestas que chegam ao vértice (**grau de entrada**).

Exemplo:



– grau(2) = 4

• Uma **relação** R, num conjunto de vértices V, é uma sequência de pares ordenados de elementos de V.

Exemplo:
V $= \{3, 5, 6, 8, 10, 17\}$

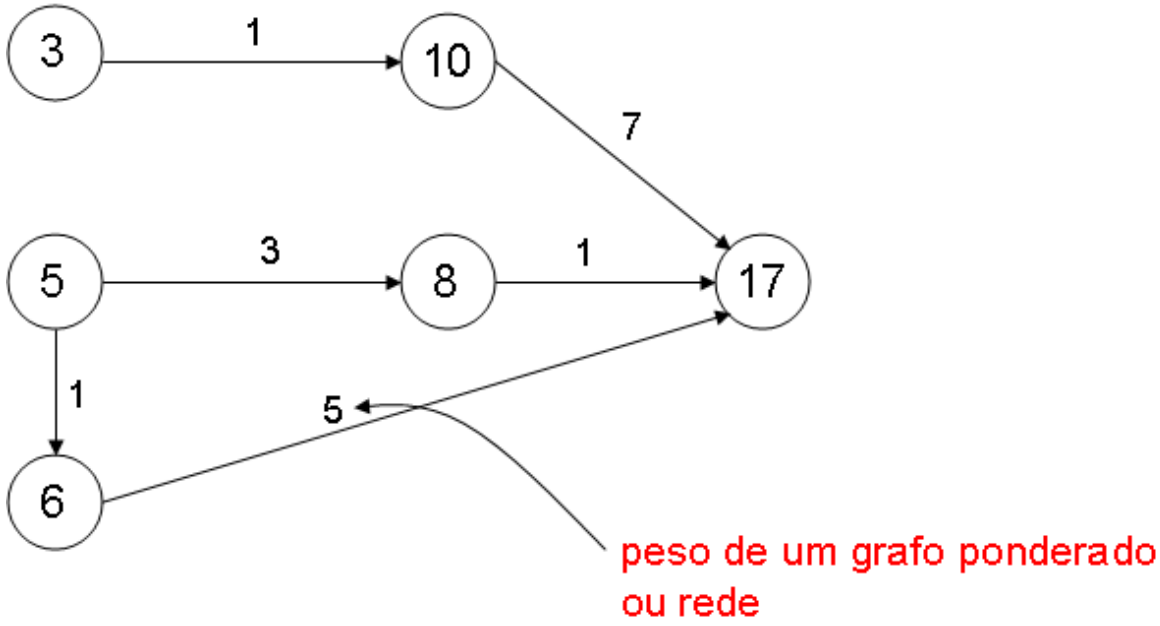
R $=$
$\{<3,10>,<5,6>,<5,8>,<6,17>,<8,17>,<10,17>\}$

$<x,y>$ $=$ x está relacionado com y em R.

R $=$ x está relacionado com y se x for menor que y e o resto da divisão de y por x é ímpar.

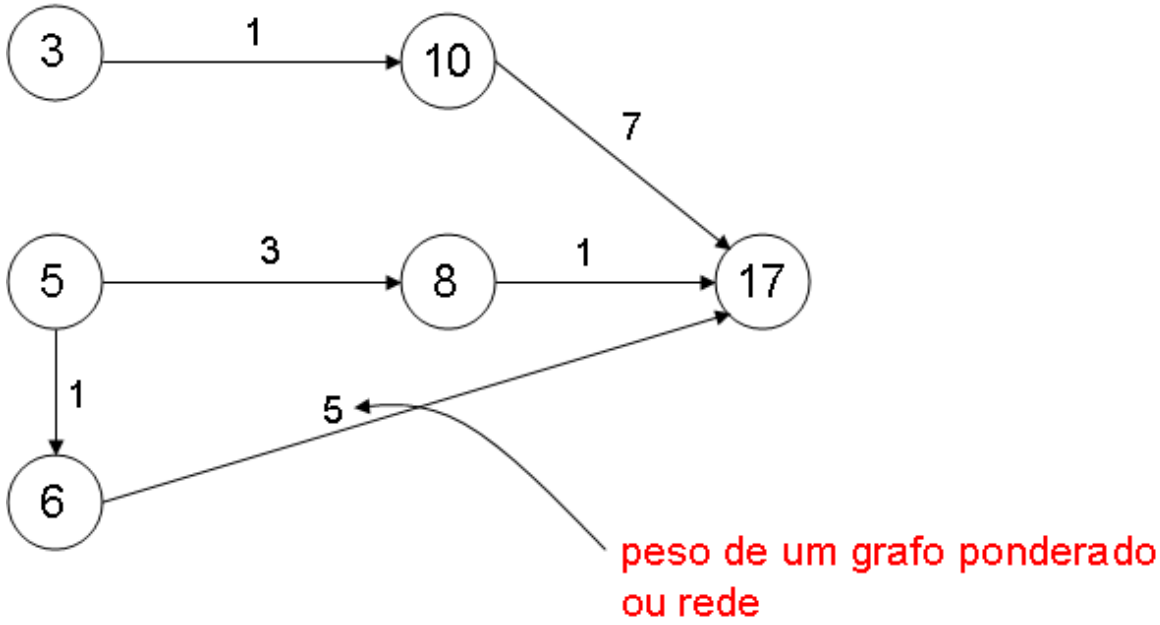
• Uma relação pode ser representada por um grafo no qual os nós representam o conjunto V e os arcos, os pares ordenados da relação:

Exemplo: R = $\{<3,10>,<5,6>,<5,8>,<6,17>,<8,17>,<10,17>\}$



• Uma relação pode ser representada por um grafo no qual os nós representam o conjunto V e os arcos, os pares ordenados da relação:

Exemplo: R = $\{<3,10>,<5,6>,<5,8>,<6,17>,<8,17>,<10,17>\}$



– Um **grafo ponderado** possui pesos associados a suas arestas que podem representar, por exemplo, custos ou distâncias.

O TAD grafos

- Exemplo de operações básicas:
  - *InsereAresta(a, b, Grafo)*: introduz uma aresta de *a* até *b*.
  - *InsereArestaP(a, b, x, Grafo)*: cria uma aresta de *a* até *b* com peso *x*, num grafo ponderado.
  - *Remove(a, b, Grafo)*: remove um arco de *a* até *b*.
  - *RemoveP(a,b,x,Grafo)*: remove um arco de *a* até *b* com peso *x*.
  - *Adjacente(a,b, Grafo)*: retorna *TRUE* se *a* e *b* são adjacentes e *FALSE* em caso contrário.

O TAD grafos

- Exemplo de operações básicas:
  - *InsereAresta(a, b, Grafo)*: introduz uma aresta de *a* até *b*.
  - *InsereArestaP(a, b, x, Grafo)*: cria uma aresta de *a* até *b* com peso *x*, num grafo ponderado.
  - *Remove(a, b, Grafo)*: remove um arco de *a* até *b*.
  - *RemoveP(a,b,x,Grafo)*: remove um arco de *a* até *b* com peso *x*.
  - *Adjacente(a,b, Grafo)*: retorna *TRUE* se *a* e *b* são adjacentes e *FALSE* em caso contrário.

O TAD grafos

- Exemplo de operações básicas:
  - *InsereAresta(a, b, Grafo)*: introduz uma aresta de *a* até *b*.
  - *InsereArestaP(a, b, x, Grafo)*: cria uma aresta de *a* até *b* com peso *x*, num grafo ponderado.
  - *Remove(a, b, Grafo)*: remove um arco de *a* até *b*.
  - *RemoveP(a,b,x,Grafo)*: remove um arco de *a* até *b* com peso *x*.
  - *Adjacente(a,b, Grafo)*: retorna *TRUE* se *a* e *b* são adjacentes e *FALSE* em caso contrário.

O TAD grafos

- Exemplo de operações básicas:
  - *InsereAresta(a, b, Grafo)*: introduz uma aresta de *a* até *b*.
  - *InsereArestaP(a, b, x, Grafo)*: cria uma aresta de *a* até *b* com peso *x*, num grafo ponderado.
  - *Remove(a, b, Grafo)*: remove um arco de *a* até *b*.
  - *RemoveP(a,b,x,Grafo)*: remove um arco de *a* até *b* com peso *x*.
  - *Adjacente(a,b, Grafo)*: retorna *TRUE* se *a* e *b* são adjacentes e *FALSE* em caso contrário.

O TAD grafos

- Exemplo de operações básicas:
  - *InsereAresta(a, b, Grafo)*: introduz uma aresta de *a* até *b*.
  - *InsereArestaP(a, b, x, Grafo)*: cria uma aresta de *a* até *b* com peso *x*, num grafo ponderado.
  - *Remove(a, b, Grafo)*: remove um arco de *a* até *b*.
  - *RemoveP(a,b,x,Grafo)*: remove um arco de *a* até *b* com peso *x*.
  - *Adjacente(a,b, Grafo)*: retorna *TRUE* se *a* e *b* são adjacentes e *FALSE* em caso contrário.

O TAD grafos

- Exemplo de operações básicas:
  - *InsereAresta(a, b, Grafo)*: introduz uma aresta de *a* até *b*.
  - *InsereArestaP(a, b, x, Grafo)*: cria uma aresta de *a* até *b* com peso *x*, num grafo ponderado.
  - *Remove(a, b, Grafo)*: remove um arco de *a* até *b*.
  - *RemoveP(a,b,x,Grafo)*: remove um arco de *a* até *b* com peso *x*.
  - *Adjacente(a,b, Grafo)*: retorna *TRUE* se *a* e *b* são adjacentes e *FALSE* em caso contrário.

• Um **caminho de comprimento** k do nó a ao nó b é definido como uma sequência de $k+1$ nós, $n_0, n_1, ..., n_k$, tal que $n_0 = a$, $n_k = b$ e $Adjacente(n_i, n_{i+1}, Grafo)$ é $TRUE$ para todo i entre 0 e $k-1$.

• Se para algum inteiro k existir um caminho de comprimento k entre a e b, dizemos que existe um **caminho** de a até b.

• Em um grafo direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos uma aresta.

• Em um grafo não direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos três arestas.

• Um grafo sem ciclos é um grafo **acíclico**.

• Um **caminho de comprimento** k do nó a ao nó b é definido como uma sequência de $k+1$ nós, $n_0, n_1, ..., n_k$, tal que $n_0 = a$, $n_k = b$ e $Adjacente(n_i, n_{i+1}, Grafo)$ é $TRUE$ para todo i entre 0 e $k-1$.

• Se para algum inteiro k existir um caminho de comprimento k entre a e b, dizemos que existe um **caminho** de a até b.

• Em um grafo direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos uma aresta.

• Em um grafo não direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos três arestas.

• Um grafo sem ciclos é um grafo **acíclico**.

• Um **caminho de comprimento** k do nó a ao nó b é definido como uma sequência de $k+1$ nós, $n_0, n_1, ..., n_k$, tal que $n_0 = a$, $n_k = b$ e $Adjacente(n_i, n_{i+1}, Grafo)$ é $TRUE$ para todo i entre 0 e $k-1$.

• Se para algum inteiro k existir um caminho de comprimento k entre a e b, dizemos que existe um **caminho** de a até b.

• Em um grafo direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos uma aresta.

• Em um grafo não direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos três arestas.

• Um grafo sem ciclos é um grafo **acíclico**.

• Um **caminho de comprimento** k do nó a ao nó b é definido como uma sequência de $k+1$ nós, $n_0, n_1, ..., n_k$, tal que $n_0 = a$, $n_k = b$ e $Adjacente(n_i, n_{i+1}, Grafo)$ é $TRUE$ para todo i entre 0 e $k-1$.

• Se para algum inteiro k existir um caminho de comprimento k entre a e b, dizemos que existe um **caminho** de a até b.

• Em um grafo direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos uma aresta.

• Em um grafo não direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos três arestas.

• Um grafo sem ciclos é um grafo **acíclico**.

• Um **caminho de comprimento** k do nó a ao nó b é definido como uma sequência de $k+1$ nós, $n_0, n_1, ..., n_k$, tal que $n_0 = a$, $n_k = b$ e $Adjacente(n_i, n_{i+1}, Grafo)$ é $TRUE$ para todo i entre 0 e $k-1$.

• Se para algum inteiro k existir um caminho de comprimento k entre a e b, dizemos que existe um **caminho** de a até b.

• Em um grafo direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos uma aresta.

• Em um grafo não direcionado, um caminho $(n_0, n_1, ..., n_k)$ forma um **ciclo** se $n_0 = n_k$ e o caminho contiver pelo menos três arestas.

• Um grafo sem ciclos é um grafo **acíclico**.

- Exemplo de aplicação:

– Identificar se existe um caminho de comprimento k (estradas) ligando duas cidades A e B.

– As cidades são numeradas de 0 a $n - 1$

• Exemplo de aplicação:

– Identificar se existe um caminho de comprimento k (estradas) ligando duas cidades A e B.

– As cidades são numeradas de 0 a $n-1$

– Entrada dos dados:

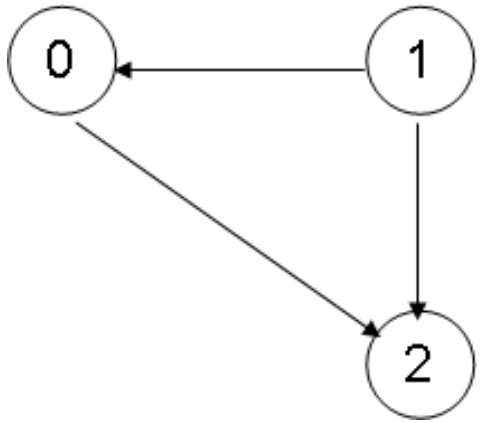
```
número n de cidades (nós)   A   B   k
cidade1   cidade2
cidade3   cidade4
    .         .   <---- cidades com estradas interligando-as
    .         .
    .         .
```

- Exemplo:

```
3          1    2    2
1  2
1  0
0  2
```

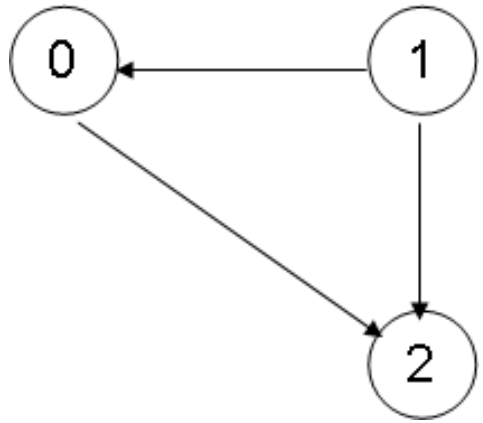
- Exemplo:

3 1 2 2
1 2
1 0
0 2



- Exemplo:

```
3          1    2    2
1  2
1  0
0  2
```



– Ideia: verificar se existe um caminho de comprimento 1 de A até um nó C, e de C até B cujo comprimento é $k-1$. Fazer isto recursivamente.

- Algoritmo:

```
Crie Grafo com n nós numerados de 0 a n-1

Leia os dados de entrada de um arquivo

while(scanf("%d %d", &cid1, &cid2) != EOF)
    InsereAresta(cid1, cid2, Grafo) ;

if(findpath(k, A, B)) printf("Existe um caminho de %d até %d
                              em %d estradas", A, B, k) ;

else
  printf("Não existe o caminho solicitado") ;
```

- A função *findpath*(k, a, b):

```
if(k==1)  /* procura um caminho de comprimento 1 */
  return (Adjacente(a, b, Grafo)) ;

for(c=0; c<n; c++)  /* determina se existe um caminho
                       através de c */
 if(Adjacente(a,c,Grafo) && findpath(k-1,c,b)) return(TRUE);

return(FALSE) ; /* não existe o caminho procurado */
```

- Matrizes de Adjacência

– A **matriz de adjacência** de um grafo $G = (V, A)$, contendo n vértices, é uma matriz $n \times n$ em que $A[i, j]$ é 1 (verdadeiro) se e somente se existir um arco do vértice i ao vértice j.

Para grafos ponderados, $A[i, j]$ contém o rótulo ou peso associado à aresta. Se não existir uma aresta de i para j, um *flag* especial deve ser usado para indicar tal ocorrência.

Implementação

```
#define  MAXNOS  xxx ;
struct node {
  /* informação relativa ao nó */
} ;

struct arc {
  int adj ;
  / * informação relativa a cada arco */
} ;

struct grafo {
  struct node nos[MAXNOS] ;
  struct arc arcos[MAXNOS][MAXNOS] ;
} ;

struct grafo Grafo ;
```

- Assim:

$$Grafo.arcos[i][j].adj = \begin{cases} TRUE, & \text{se } i \text{ adjacente a } j \\ FALSE, & \text{senão} \end{cases}$$

- Assim:

$$Grafo.arcos[i][j].adj = \begin{cases} TRUE, & \text{se } i \text{ adjacente a } j \\ FALSE, & \text{senão} \end{cases}$$

$$Grafo.arcos[\,][\,] \Rightarrow \text{MATRIZ DE ADJACÊNCIA}$$

- Assim:

$$Grafo.arcos[i][j].adj = \begin{cases} TRUE, & \text{se } i \text{ adjacente a } j \\ FALSE, & \text{senão} \end{cases}$$

$$Grafo.arcos[\,][\,] \Rightarrow \text{MATRIZ DE ADJACÊNCIA}$$

– Frequentemente, os nós são numerados de 0 a $MAXNOS - 1$ e se não existem informações sobre os arcos, a declaração se torna:

$$int \;\; adj[MAXNOS][MAXNOS] \;;$$

e o grafo é totalmente descrito por sua matriz de adjacência.

Funções básicas

```
-- InsereAresta(int no1, int no2, int adj[ ][MAXNOS]) {
  /* inclui um arco de no1 a no2 */
  adj[no1][no2] = TRUE ;
}

-- Remove(int no1, int no2, int adj[ ][MAXNOS]) {
  /* elimina arco de no1 a no2 */
  adj[no1][no2] = FALSE ;
}

-- Adjacente(int no1, int no2, int adj[ ][MAXNOS]) {
  /* identifica se dois arcos são adjacente */
  return((adj[no1][no2] == TRUE) ? TRUE:FALSE) ;
  }
```

Funções básicas

• Um grafo ponderado e com um número fixo de nós pode ser declarado por:

```
 struct arc {
    int adj ;
    int peso ;
} ;

struct arc Grafo[MAXNOS][MAXNOS] ;
```

E assim:

```
-- InsereArestaP(int no1, int no2, int x,
                  struct arc Grafo[ ][MAXNOS]) {

     Grafo[no1][no2].adj = TRUE ;
     Grafo[no1][no2].peso = x ;
}
```

Funções básicas

• Um grafo ponderado e com um número fixo de nós pode ser declarado por:

```
 struct arc {
    int adj ;
    int peso ;
} ;

struct arc Grafo[MAXNOS][MAXNOS] ;
```

E assim:

```
-- InsereArestaP(int no1, int no2, int x,
                  struct arc Grafo[ ][MAXNOS]) {

     Grafo[no1][no2].adj = TRUE ;
     Grafo[no1][no2].peso = x ;
}
```

Fechamento transitivo

- Seja um grafo com dois nós, 0 e 1, e matriz de adjacência:

$$int \quad adj[MAXNOS][MAXNOS] \Rightarrow matriz\ 2 \times 2$$

Fechamento transitivo

- Seja um grafo com dois nós, 0 e 1, e matriz de adjacência:

$$int \;\; adj[MAXNOS][MAXNOS] \Rightarrow matriz \; 2 \times 2$$

adj :

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \qquad a[i,j] = \begin{cases} 1, & \text{se adj[i][j]} = \text{TRUE} \\ 0, & \text{senão} \end{cases}$$

Fechamento transitivo

- Seja um grafo com dois nós, 0 e 1, e matriz de adjacência:

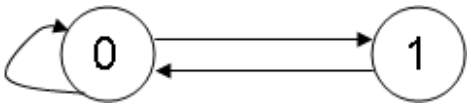
$$int \;\; adj[MAXNOS][MAXNOS] \Rightarrow matriz \; 2 \times 2$$

adj :

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$$

$$a[i,j] = \begin{cases} 1, & \text{se adj[i][j]} = \text{TRUE} \\ 0, & \text{senão} \end{cases}$$

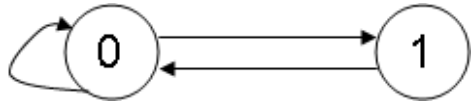
Exemplo:



adj_1 :

	0	1
0	1	1
1	1	0

Exemplo:



adj_1 :

	0	1
0	1	1
1	1	0

– Observe que a seguinte expressão:

$(adj[i][0]$ && $adj[0][j])$ || $(adj[i][1]$ && $adj[1][j])$, para $i, j = 0, 1$

indica, quando $(adj[i][k]$ && $adj[k][j]) = TRUE$, que existe um caminho de comprimento 2 de i até j passando por k.

• O produto booleano das matrizes adj_1 e adj_1 (produto considerando-se a multiplicação como a *conjunção* && e a adição como a *disjunção* ||) fornece a matriz de caminhos de comprimento 2 do nó 0 ao nó 1.

Assim:

$adj_1 \times adj_1 = adj_2$:

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

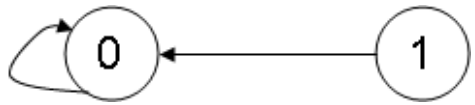
• O produto booleano das matrizes adj_1 e adj_1 (produto considerando-se a multiplicação como a *conjunção* && e a adição como a *disjunção* ||) fornece a matriz de caminhos de comprimento 2 do nó 0 ao nó 1.

Assim:

$adj_1 \times adj_1 = adj_2$:

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

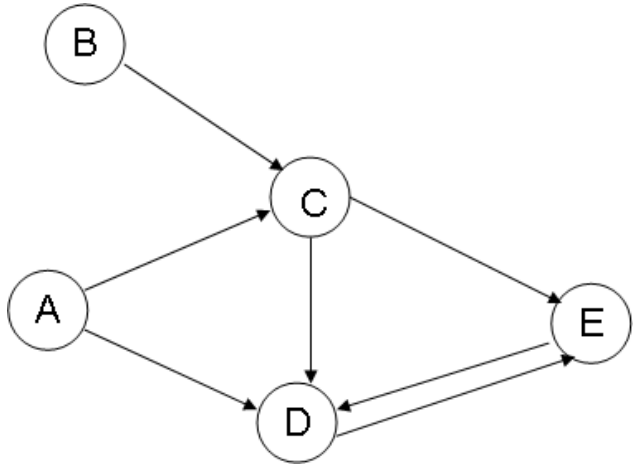
– Exemplo 2:



$adj_1 \times adj_1 = adj_2$:

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

– Exemplo 4:



adj_1 :

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

adj_2 :

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

- Um caminho de comprimento igual a 3 será dado por:

$$adj_3 = adj_2 \quad produto\_booleano \quad adj_1$$

De modo geral:

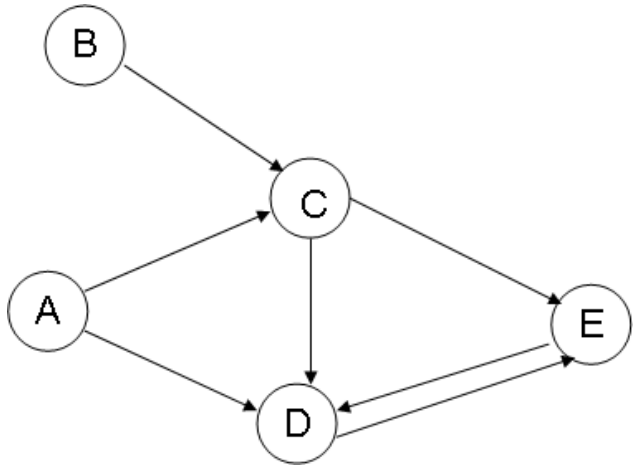
$$adj_l = adj_{l-1} \quad produto\_booleano \quad adj_1$$

- Um caminho de comprimento igual a 3 será dado por:

$$adj_3 = adj_2 \quad produto\_booleano \quad adj_1$$

De modo geral:

$$adj_l = adj_{l-1} \quad produto\_booleano \quad adj_1$$



adj_3 :

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

adj_4 :

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

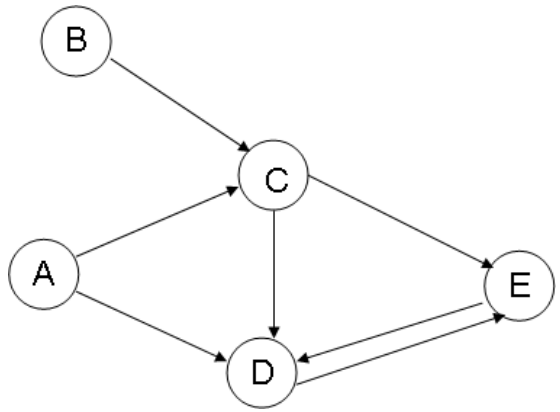
• Um caminho de comprimento menor ou igual a 3, entre dois nós de um grafo, é dado por:

$$adj_1[i][j] \;||\; adj_2[i][j] \;||\; adj_3[i][j]$$

• Um caminho de comprimento menor ou igual a 3, entre dois nós de um grafo, é dado por:

$$adj_1[i][j] \;||\; adj_2[i][j] \;||\; adj_3[i][j]$$

Exemplo:



	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

- Para um caminho de n nós, uma matriz $\underline{path}$ tal que $path[i][j] = TRUE$ se e somente se existir um caminho do nó i ao nó j, de qualquer comprimento, é dada por:

$$path[i][j] = adj_1[i][j] \;||\; adj_2[i][j] \;||\; \cdots \;||\; adj_n[i][j]$$

• Para um caminho de n nós, uma matriz *path* tal que $path[i][j] = TRUE$ se e somente se existir um caminho do nó i ao nó j, de qualquer comprimento, é dada por:

$$path[i][j] = adj_1[i][j] \;||\; adj_2[i][j] \;||\; \cdots \;||\; adj_n[i][j]$$

A matriz *path* é denominada fechamento transitivo da matriz de adjacências.

Percursos em grafos

• Os nós devem ser visitados apenas uma vez (sem laços infinitos, como no caso de grafos cíclicos).

• Cada vértice visitado pode ser marcado, evitando novas visitas ao mesmo.

– Busca em profundidade

- Cada vértice v é visitado, assim como seus vizinhos adjacentes não visitados.
- O procedimento, no caso de um vértice v sem adjacentes ou com todos eles já visitados, retorna ao predecessor de v.
- O percurso encerra-se com o retorno do procedimento ao vértice do início do percurso.
- No caso de vértices isolados, o percurso reinicia-se a partir de vértices não visitados.

Percursos em grafos

• Os nós devem ser visitados apenas uma vez (sem laços infinitos, como no caso de grafos cíclicos).
• Cada vértice visitado pode ser marcado, evitando novas visitas ao mesmo.

– Busca em profundidade

- Cada vértice v é visitado, assim como seus vizinhos adjacentes não visitados.
- O procedimento, no caso de um vértice v sem adjacentes ou com todos eles já visitados, retorna ao predecessor de v.
- O percurso encerra-se com o retorno do procedimento ao vértice do início do percurso.
- No caso de vértices isolados, o percurso reinicia-se a partir de vértices não visitados.

Percursos em grafos

• Os nós devem ser visitados apenas uma vez (sem laços infinitos, como no caso de grafos cíclicos).
• Cada vértice visitado pode ser marcado, evitando novas visitas ao mesmo.

– Busca em profundidade

- Cada vértice v é visitado, assim como seus vizinhos adjacentes não visitados.
- O procedimento, no caso de um vértice v sem adjacentes ou com todos eles já visitados, retorna ao predecessor de v.
- O percurso encerra-se com o retorno do procedimento ao vértice do início do percurso.
- No caso de vértices isolados, o percurso reinicia-se a partir de vértices não visitados.

Percursos em grafos

• Os nós devem ser visitados apenas uma vez (sem laços infinitos, como no caso de grafos cíclicos).
• Cada vértice visitado pode ser marcado, evitando novas visitas ao mesmo.

– Busca em profundidade

- Cada vértice v é visitado, assim como seus vizinhos adjacentes não visitados.
- O procedimento, no caso de um vértice v sem adjacentes ou com todos eles já visitados, retorna ao predecessor de v.
- O percurso encerra-se com o retorno do procedimento ao vértice do início do percurso.
- No caso de vértices isolados, o percurso reinicia-se a partir de vértices não visitados.

Percursos em grafos

• Os nós devem ser visitados apenas uma vez (sem laços infinitos, como no caso de grafos cíclicos).
• Cada vértice visitado pode ser marcado, evitando novas visitas ao mesmo.

– Busca em profundidade

- Cada vértice v é visitado, assim como seus vizinhos adjacentes não visitados.
- O procedimento, no caso de um vértice v sem adjacentes ou com todos eles já visitados, retorna ao predecessor de v.
- O percurso encerra-se com o retorno do procedimento ao vértice do início do percurso.
- No caso de vértices isolados, o percurso reinicia-se a partir de vértices não visitados.

Percursos em grafos

• Os nós devem ser visitados apenas uma vez (sem laços infinitos, como no caso de grafos cíclicos).
• Cada vértice visitado pode ser marcado, evitando novas visitas ao mesmo.

– Busca em profundidade

- Cada vértice v é visitado, assim como seus vizinhos adjacentes não visitados.
- O procedimento, no caso de um vértice v sem adjacentes ou com todos eles já visitados, retorna ao predecessor de v.
- O percurso encerra-se com o retorno do procedimento ao vértice do início do percurso.
- No caso de vértices isolados, o percurso reinicia-se a partir de vértices não visitados.

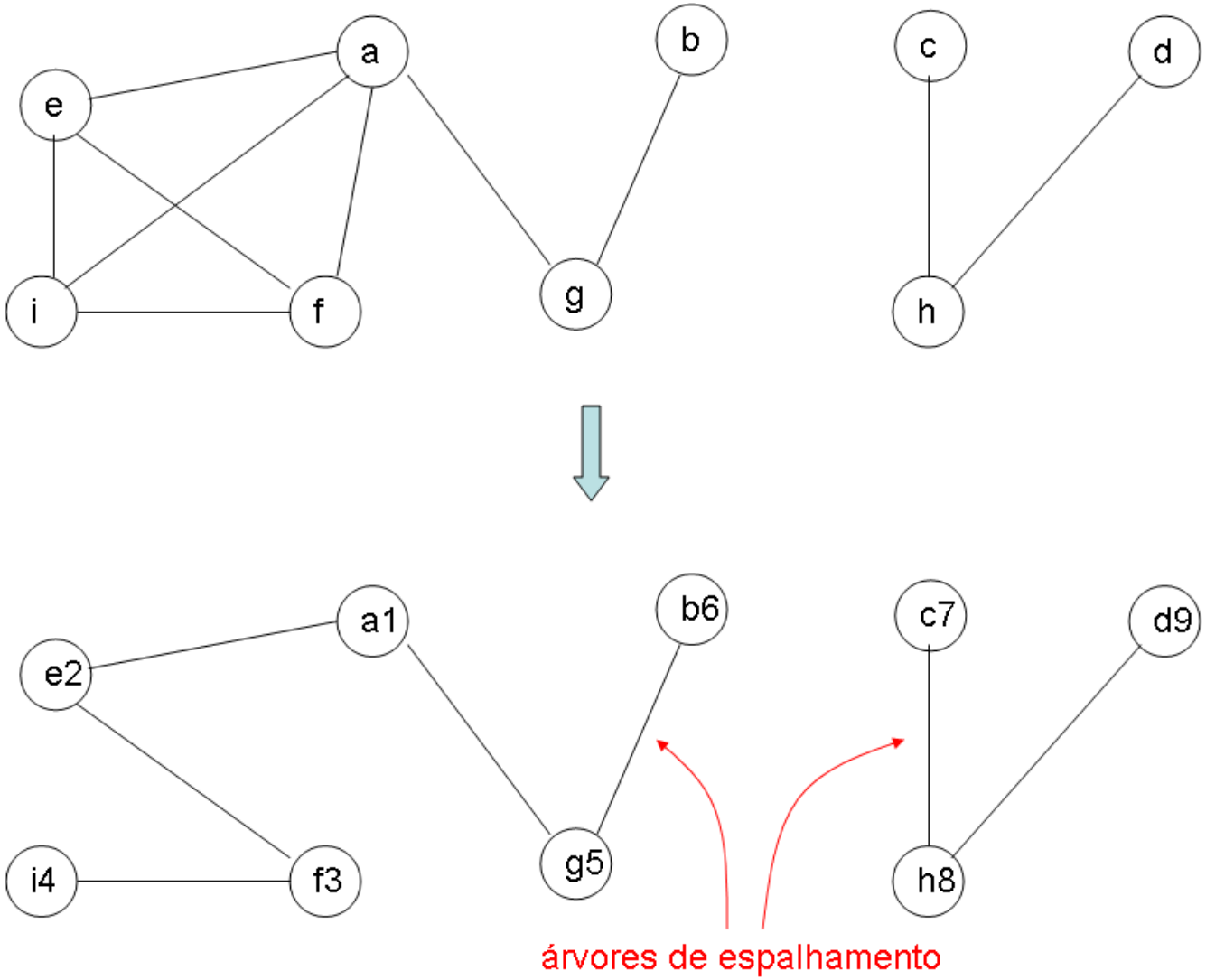
- Algoritmo:

```
BuscaProfundidade(Grafo)

  para todos os vértices v
    num(v) = 0 ;  /* zera flag dos nós  */
  conjunto_de_arestas = 0 ; /* zera uma matriz de adjacências
                               auxiliar */
  i = 1 ;
  Enquanto existir um vértice v tal que num(v) = 0
    VisitaProfundidade(v) ;

 Imprime conjunto_de_arestas ;
```

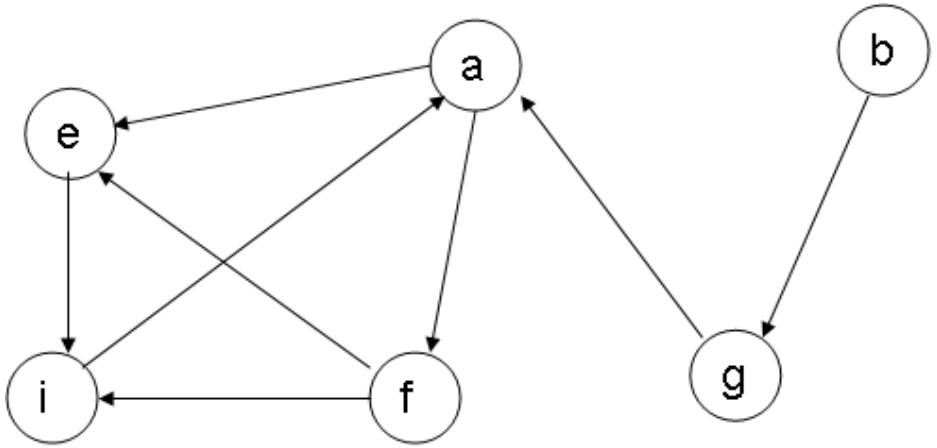
- A função *VisitaProfundidade*:

```
VisitaProfundidade(v)

  num(v) = i++ ;     /* usa i e incrementa */
  para todos os vértices u adjacentes a v
    if num(u) = 0 ;
      introduza a aresta (u,v) ao conjunto_de_arestas ;
      VisitaProfundidade(u) ;
```

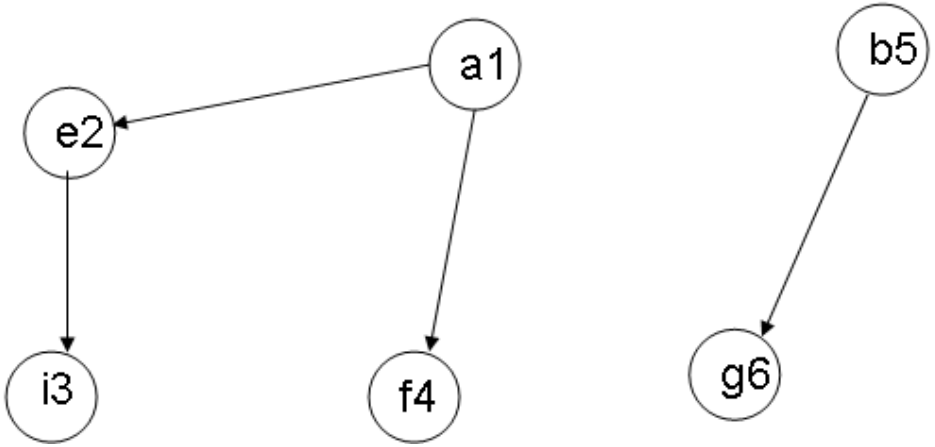
- Exemplo 1:



- Exemplo 2: Um dígrafo



um grafo



duas árvores de espalhamento

- Busca em largura

– Considera uma *fila*, tal como nas árvores, que guarda os nós a serem visitados.

• Busca em largura

– Considera uma *fila*, tal como nas árvores, que guarda os nós a serem visitados.

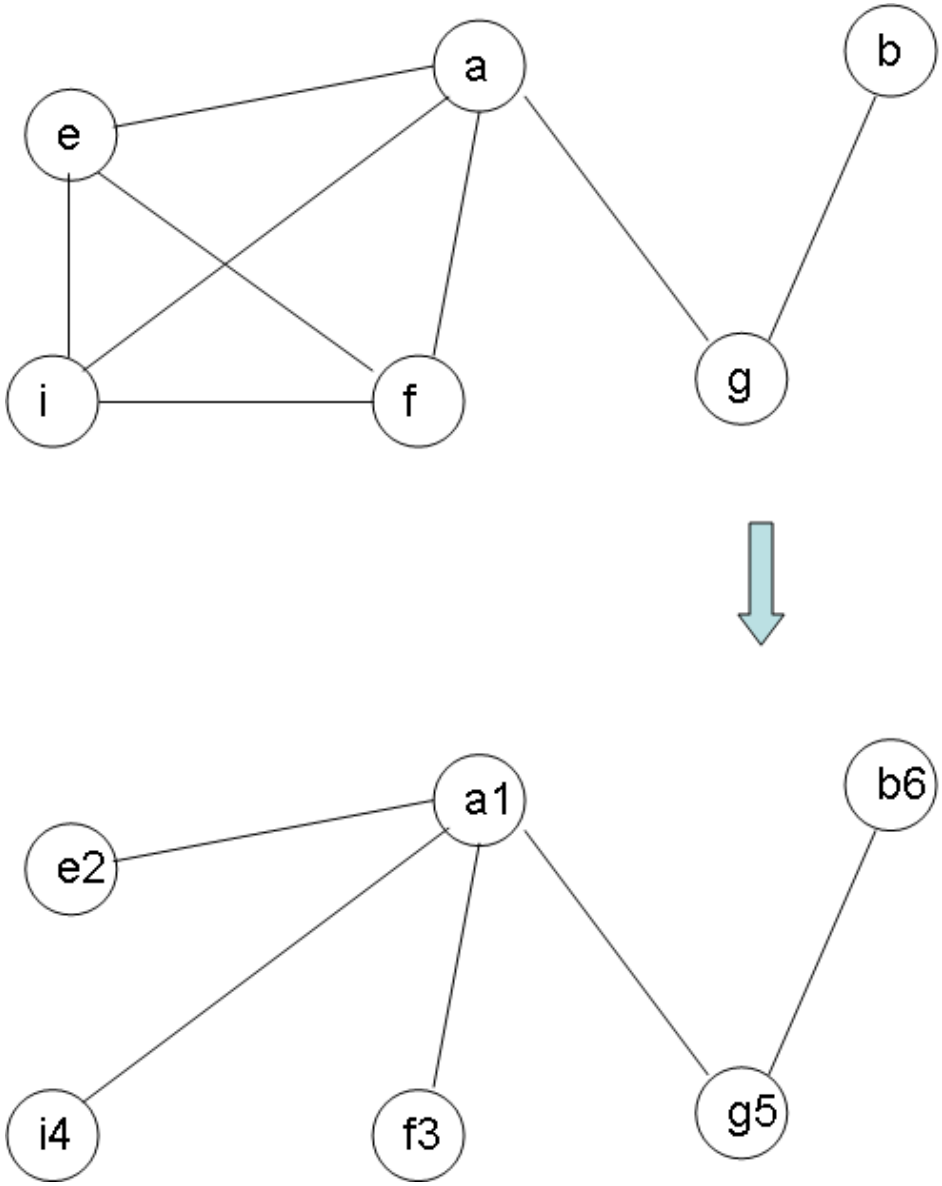
– Algoritmo:

```
BuscaEmLargura(Grafo)

 para todos os vértices u
   num(u) = 0 ;
 conjuntos_de_arestas = 0 ; /* zera uma matriz de adjacências
                               auxiliar */
 i = 1 ;
```

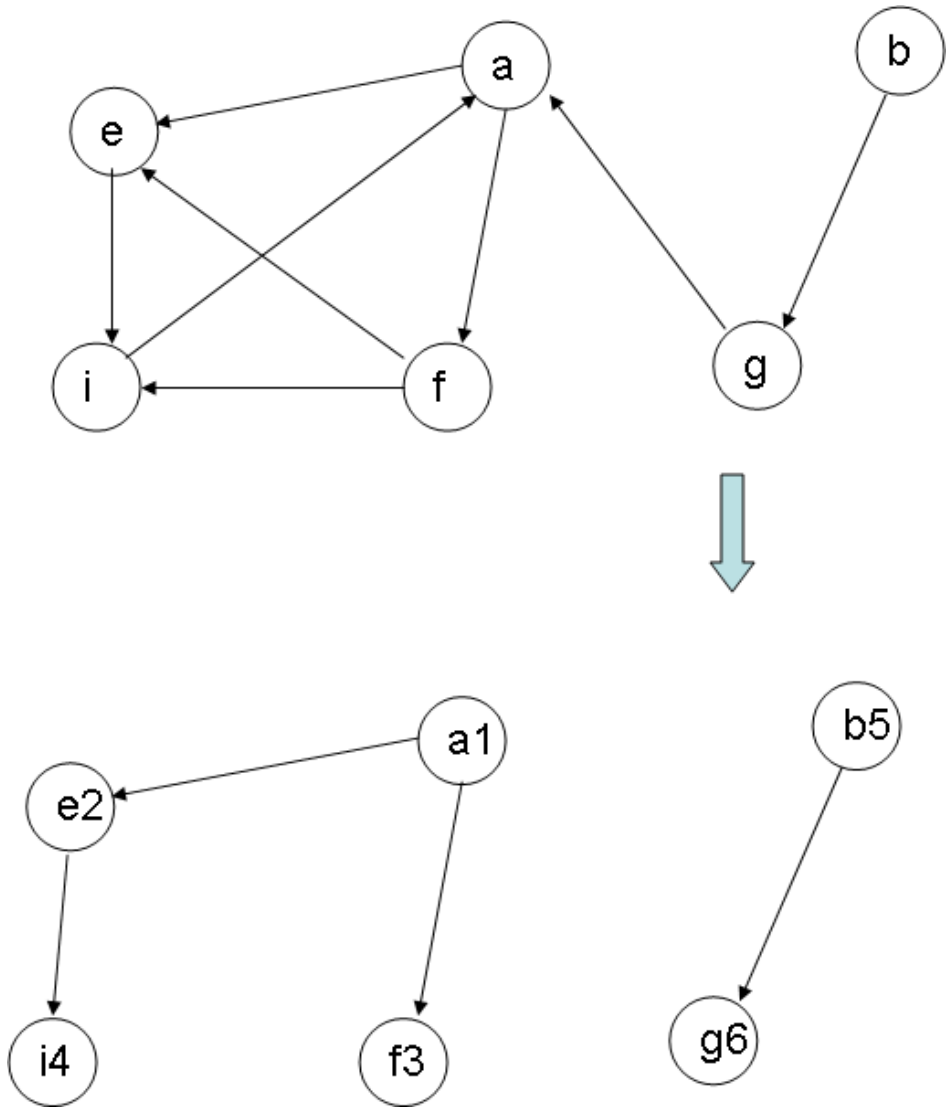
- ... continuação

```
Enquanto há vértice v tal que num(v) = 0
  num(v) = i++ ;  /* usa i e incrementa */
  insere_fila(v) ;
  Enquanto fila_não_vazia
    v = retira_fila() ;
    Para todos os vértices u adjacentes a v
      if num(u) = 0
        num(u) = i++ ;
        insere_fila(u) ;
        introduz aresta (v,u) ao conjunto_de_arestas

Imprime conjunto_de_arestas
```

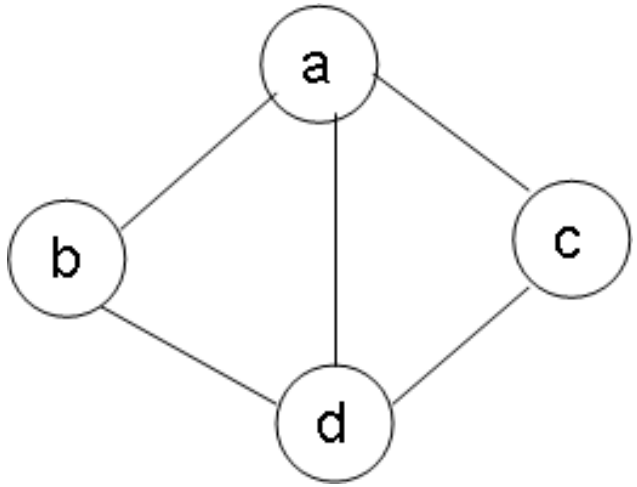
- Exemplo 1:



- Exemplo 2: Um dígrafo



Outras implementações de grafos

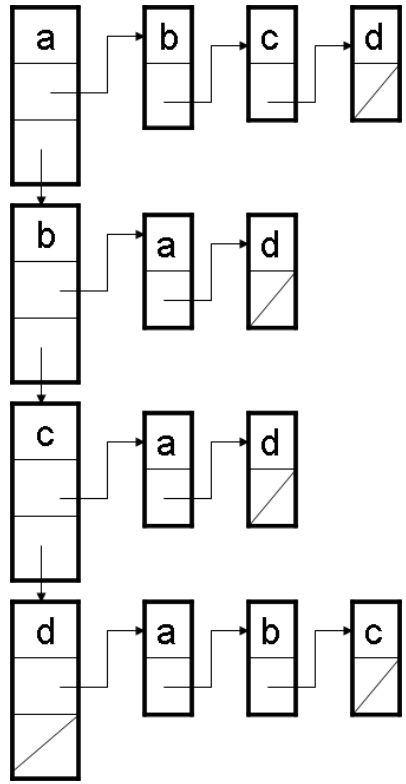


– matriz de adjacências:

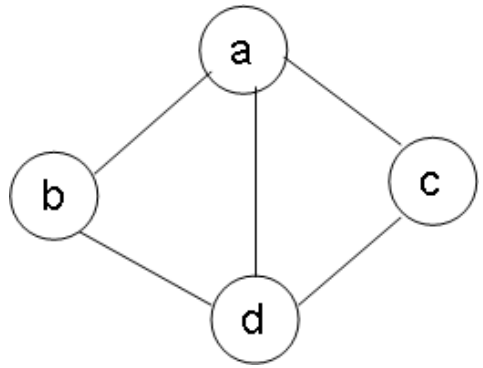
	a	b	c	d
a	0	1	1	1
b	1	0	0	1
c	1	0	0	1
d	1	1	1	0

– lista de adjacências:

a	b	c	d
b	a	d	
c	a	d	
d	a	b	c



– matriz de incidência:



	ab	ac	ad	bd	dc
a	1	1	1	0	0
b	1	0	0	1	0
c	0	1	0	0	1
d	0	0	1	1	1

$$a_{ij} = \begin{cases} 1, & \text{se a aresta } e_j \text{ é incidente com o vértice } v_i \\ 0, & \text{caso contrário} \end{cases}$$

Caminhos mais curtos

- Encontrar os caminhos mais curtos de um vértice até todos os demais de um grafo.

Caminhos mais curtos

- Encontrar os caminhos mais curtos de um vértice até todos os demais de um grafo.

- Para um vértice v, define rótulos permanentes do tipo:

$$\text{rótulo}(v) = (\text{DistânciaCorrente}(v),\ \textit{predecessor}(v))$$

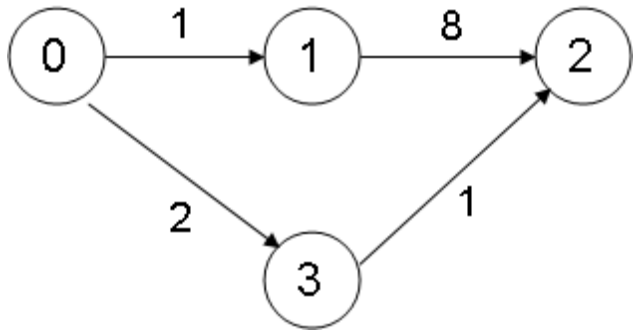
- Algoritmo 1 (Gallo e Pallotino):

```
MenorCaminho (Grafo ponderado, primeiro vértice)

  para todos os vértices v
    DistCorr(v) = "infinito" ;

  DistCorr(primeiro) = 0 ;

  Inicializa Conjunto_a_ser_checado com primeiro ;
```

Continua...

```
Enquanto Conjunto_a_ser_checado != 0
  v = um vértice de Conjunto_a_ser_checado ;
  remove v de Conjunto_a_ser_checado ;

  para todos os vértices u adjacentes a v
    se (DistCorr(u) > DistCorr(v) + peso(aresta(vu)))
      DistCorr(u) = DistCorr(v) + peso(aresta(vu)) ;
      predecessor(u) = v ;
      adicione u a Conjunto_a_ser_checado caso o mesmo
      ainda não se encontre nele.
```

- Exemplo:



iteração:	inicial	1	2	3	4
vértice ativo:		0	1	3	2
0	0				
1	∞	1			
2	∞	∞	9	3	
3	∞	2			

- Algoritmo de Dijkstra

  - Visando uma maior eficiência do algoritmo anterior, estabelece-se uma ordem de tal forma que um número de caminhos $p_1, ..., p_n$, a partir de um vértice v, é testado e, a cada teste, o menor caminho é escolhido entre eles, o que significa que um mesmo caminho p_i pode ser aumentado adicionando-se mais uma aresta ao mesmo.

- Algoritmo de Dijkstra

  - Visando uma maior eficiência do algoritmo anterior, estabelece-se uma ordem de tal forma que um número de caminhos $p_1, ..., p_n$, a partir de um vértice v, é testado e, a cada teste, o menor caminho é escolhido entre eles, o que significa que um mesmo caminho p_i pode ser aumentado adicionando-se mais uma aresta ao mesmo.
  - Se for mais longo que qualquer outro caminho que possa ser tentado, p_i é abandonado e um outro caminho é seguido a partir do ponto anterior, adicionado-se mais uma aresta a ele.

- Algoritmo de Dijkstra

  - Visando uma maior eficiência do algoritmo anterior, estabelece-se uma ordem de tal forma que um número de caminhos $p_1, ..., p_n$, a partir de um vértice v, é testado e, a cada teste, o menor caminho é escolhido entre eles, o que significa que um mesmo caminho p_i pode ser aumentado adicionando-se mais uma aresta ao mesmo.
  - Se for mais longo que qualquer outro caminho que possa ser tentado, p_i é abandonado e um outro caminho é seguido a partir do ponto anterior, adicionado-se mais uma aresta a ele.
  - Novos caminhos são criados a partir dos novos vértices introduzidos. Cada vértice é pesquisado apenas uma vez.

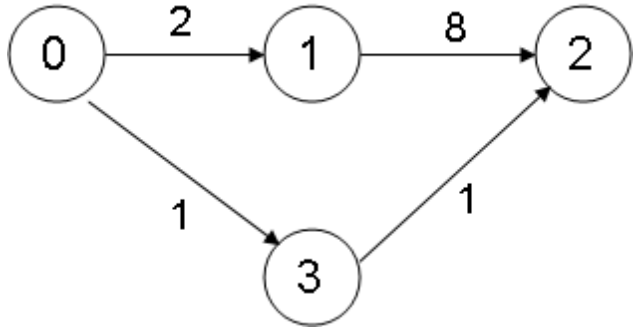
- Algoritmo:

```
AlgoritmoDijkstra (Grafo ponderado, primeiro vértice)

  para todos os vértices v
    DistCorr(v) = "infinito";

  DistCorr(primeiro) = 0 ;

 Conjunto_a_ser_checado = todos os vértices ;
```

Continua ...

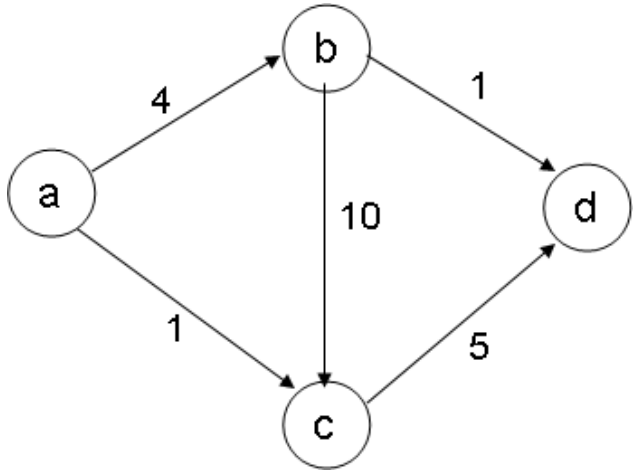
```
Enquanto Conjunto_a_ser_checado != 0
  v = um vértice em Conjunto_a_ser_checado
        com DistCorr(v) mínimo ;
  remove v de Conjunto_a_ser_checado ;

  para todos os vértices u adjacentes a v e
  em Conjunto_a_ser_checado
     se (DistCorr(u) > DistCorr(v) + peso(aresta(vu)))
        DistCorr(u) = DistCorr(v) + peso(aresta(vu)) ;
        predecessor(u) = v ;
```

- Exemplo 1:



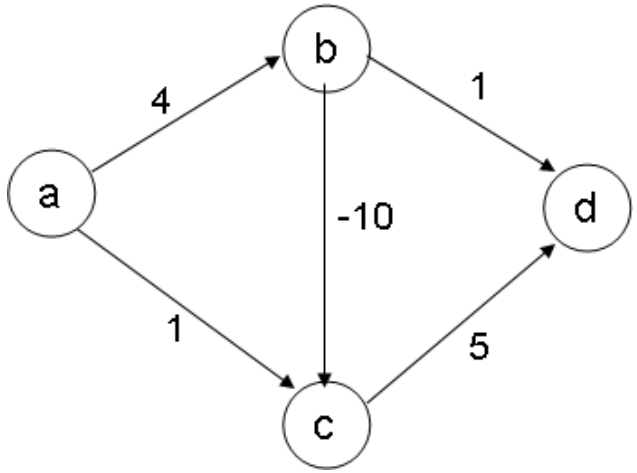
iteração:	inicial	1	2	3	4
vértice ativo:		0	3	1	2
0	0				
1	∞	2			
2	∞	∞	2		
3	∞	1			

- Exemplo 2:



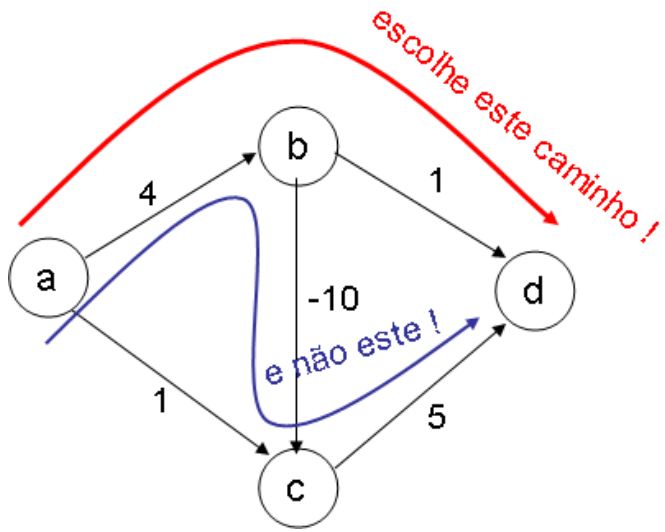
iteração:	inicial	1	2	3	4
vértice ativo:		a	c	b	d
a	0				
b	∞	4			
c	∞	1			
d	∞	∞	6	5	

- Exemplo 3:



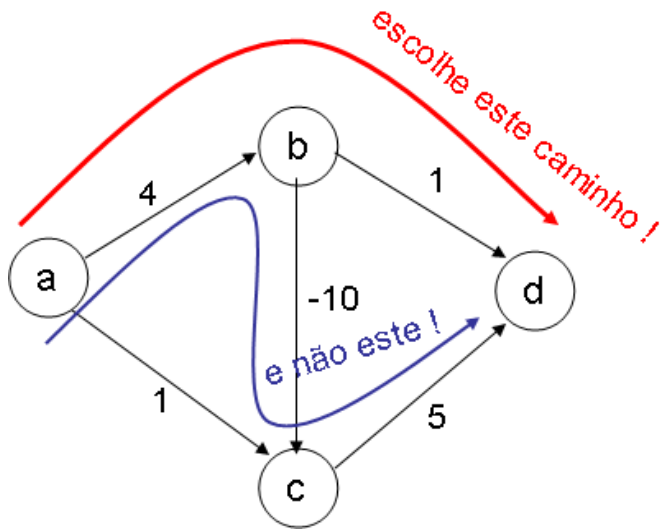
iteração:	inicial	1	2	3	4
vértice ativo:		a	c	b	d
a	0				
b	∞	4			
c	∞	1		-6	
d	∞	∞	6	5	

- Exemplo 3:



iteração:	inicial	1	2	3	4
vértice ativo:		a	c	b	d
a	0				
b	∞	4			
c	∞	1		-6	
d	∞	∞	6	5	

- Exemplo 3:



iteração:	inicial	1	2	3	4
vértice ativo:		a	c	b	d
a	0				
b	∞	4			
c	∞	1		-6	
d	∞	∞	6	5	

– Conclusão: o algoritmo pode falhar quando usado com pesos negativos.

Correção de rótulos

• Permite arestas com pesos negativos a partir de uma correção iterativa das distâncias (rótulos).

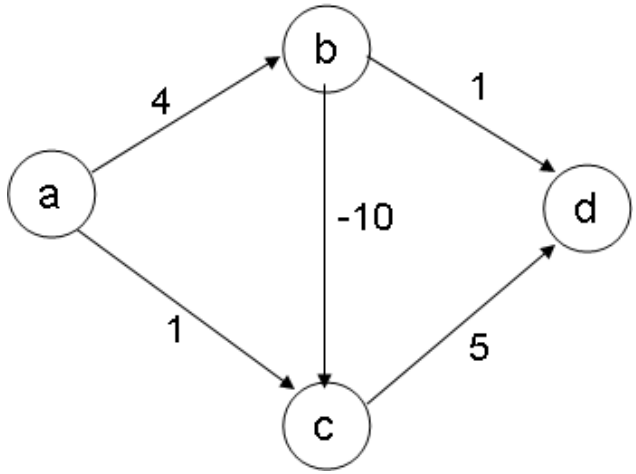
Correção de rótulos

• Permite arestas com pesos negativos a partir de uma correção iterativa das distâncias (rótulos).

– Algoritmo:

```
CorreçãoRótulo (grafo ponderado, primeiro vértice)

  para todos os vértices v
    DistCorr(v) = "infinito" ;

  DistCorr(primeiro) = 0 ;

  Enquanto houver uma aresta (vu) tal que
  DistCorr(u) > DistCorr(v) + peso(aresta(vu))
    DistCorr(u) = DistCorr(v) + peso(aresta(vu)) ;
```

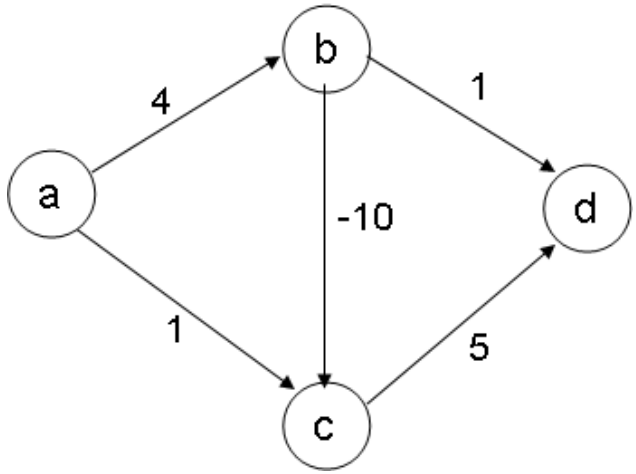
- Exemplo:



– Matriz de adjacências:

	a	b	c	d
a	0	1	1	0
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

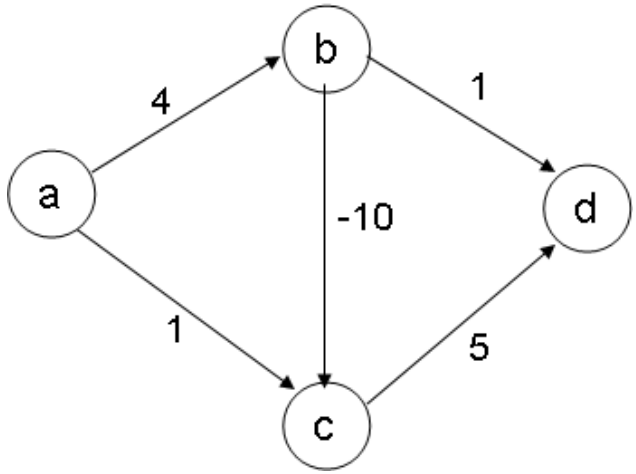
- Exemplo:



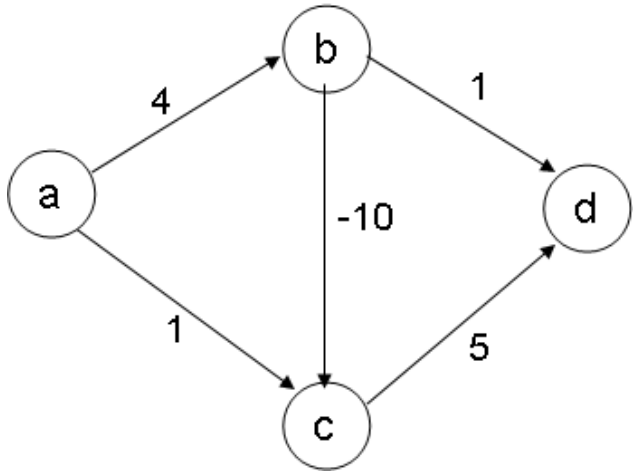
– Matriz de adjacências:

	a	b	c	d
a	0	1	1	0
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

– Exemplo de ordem de consideração das arestas: *ab ac bc bd cd*.

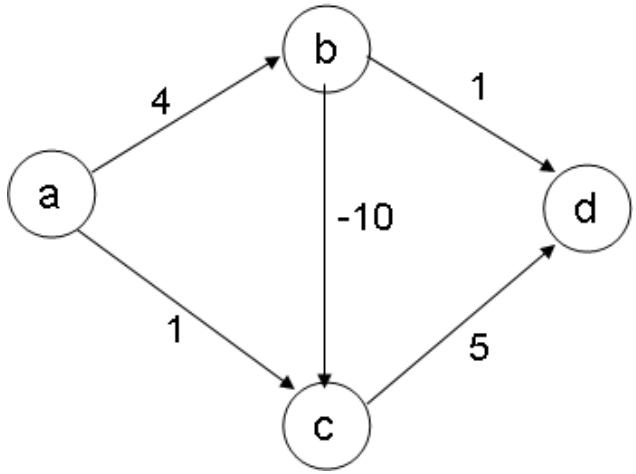


• Considerando-se as arestas *ab ac bc bd cd*, nesta ordem, as etapas do algoritmo até a idempotência são:

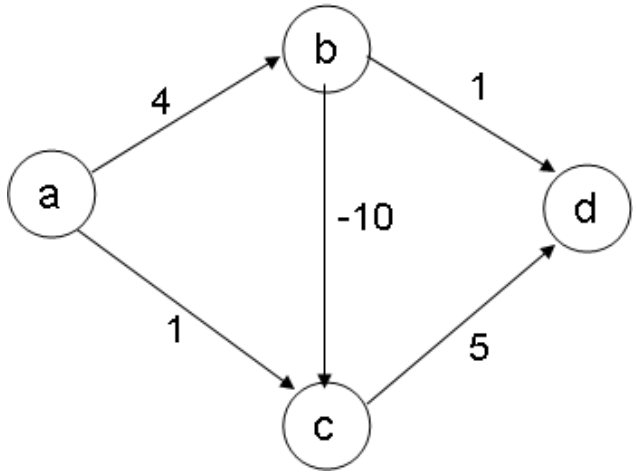


• Considerando-se as arestas *ab ac bc bd cd*, nesta ordem, as etapas do algoritmo até a idempotência são:

	iteração		
	iníco	1	2
a	0		
b	∞	4	
c	∞	1, -6	
d	∞	5, -1	



Mudando-se a ordem de consideração das arestas para *ab cd ac bc bd*, as etapas são:



Mudando-se a ordem de consideração das arestas para *ab cd ac bc bd*, as etapas são:

	iteração		
	iníco	1	2
a	0		
b	∞	4	
c	∞	1, -6	
d	∞	∞, 5	-1

Referências:

– Aaron M. Tenenbaum et al. Estruturas de Dados Usando C, Makron Books, 1995.

– Nívio Ziviani. Projeto de Algoritmos com Implementações em Pascal e C, Thomson Learning, 2004.

– Adam Drozdek. Estruturas de Dados e Algoritmos em C++, Thomson Learning, 2002.