

# GRAFOS

**1 – Aspectos gerais**

**2 – Grafos orientados**

**3 – Problemas clássicos sobre grafos orientados**

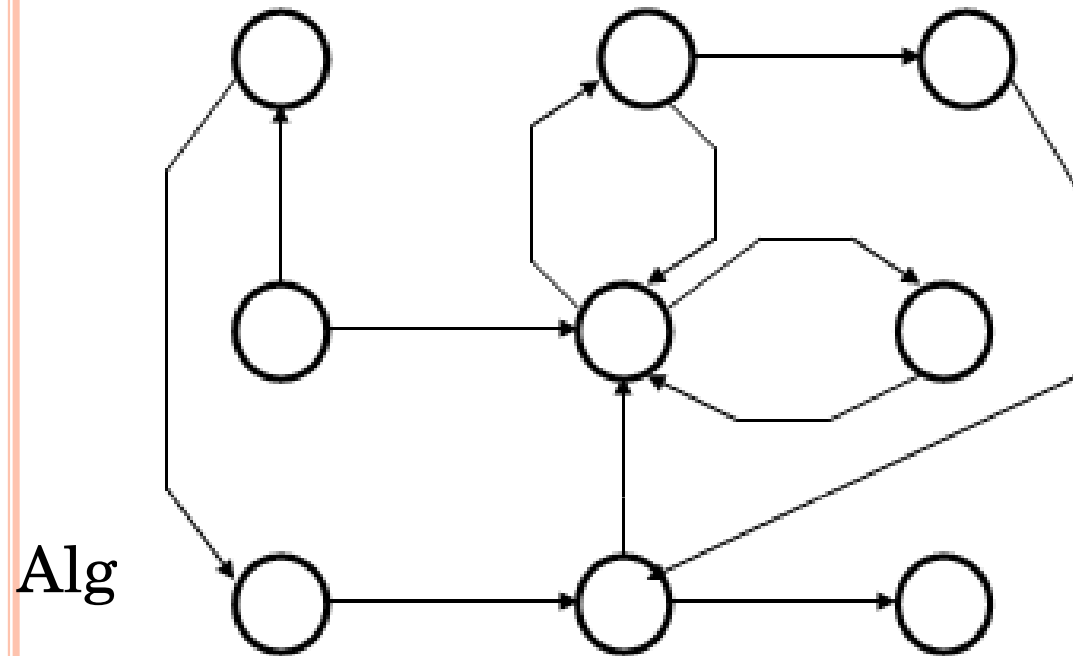
**4 – Grafos não-orientados**

**5 – Problemas clássicos sobre grafos não-orientados**

# 1 – ASPECTOS GERAIS

- **Grafo:** modelo de armazenamento de informações usado para representar uma dada relação entre alguns pares do universo de elementos armazenados
- **Vértice ou nó:** unidade destinada a guardar todas as informações de um elemento desse universo
- **Arco:** representa a existência da referida relação entre dois elementos desse universo

## Forma natural de representação de grafos:



**Legenda:**

Círculo = Vértice

Seta = Arco

**V:** conjunto de vértices

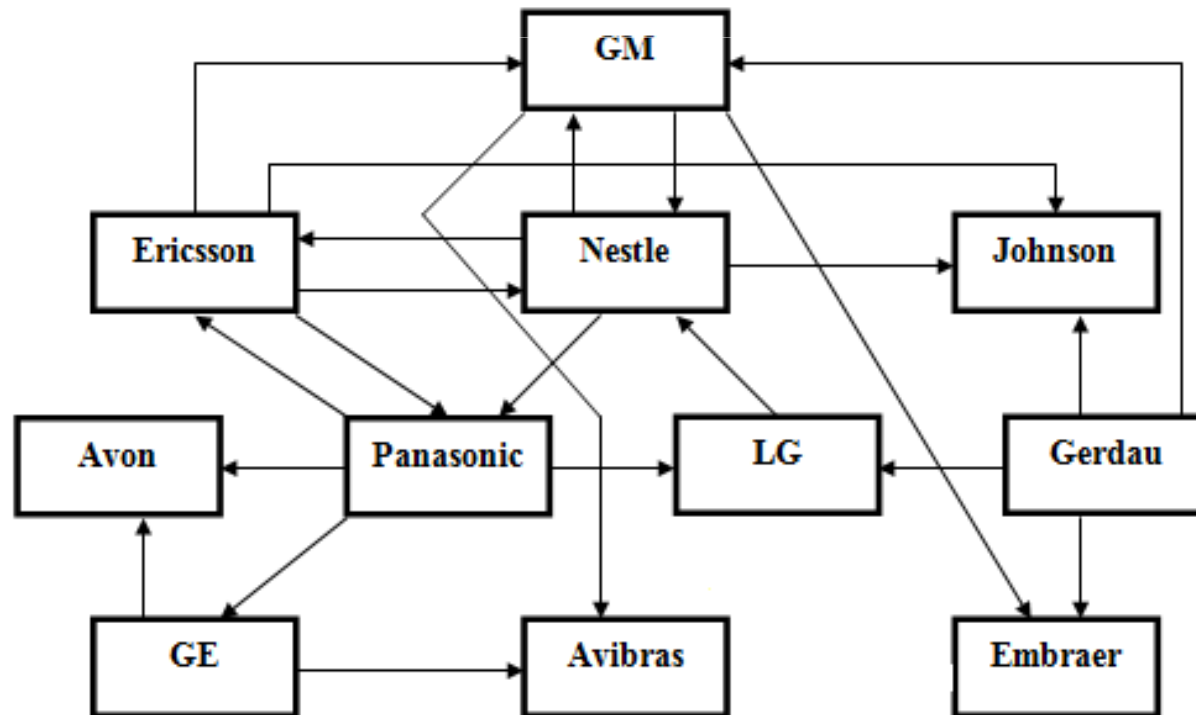
**A:** conjunto de arcos ligando vértices de **V**

## Exemplos:

a) Fornecimento de produtos entre fábricas

**Objetos:** fábricas

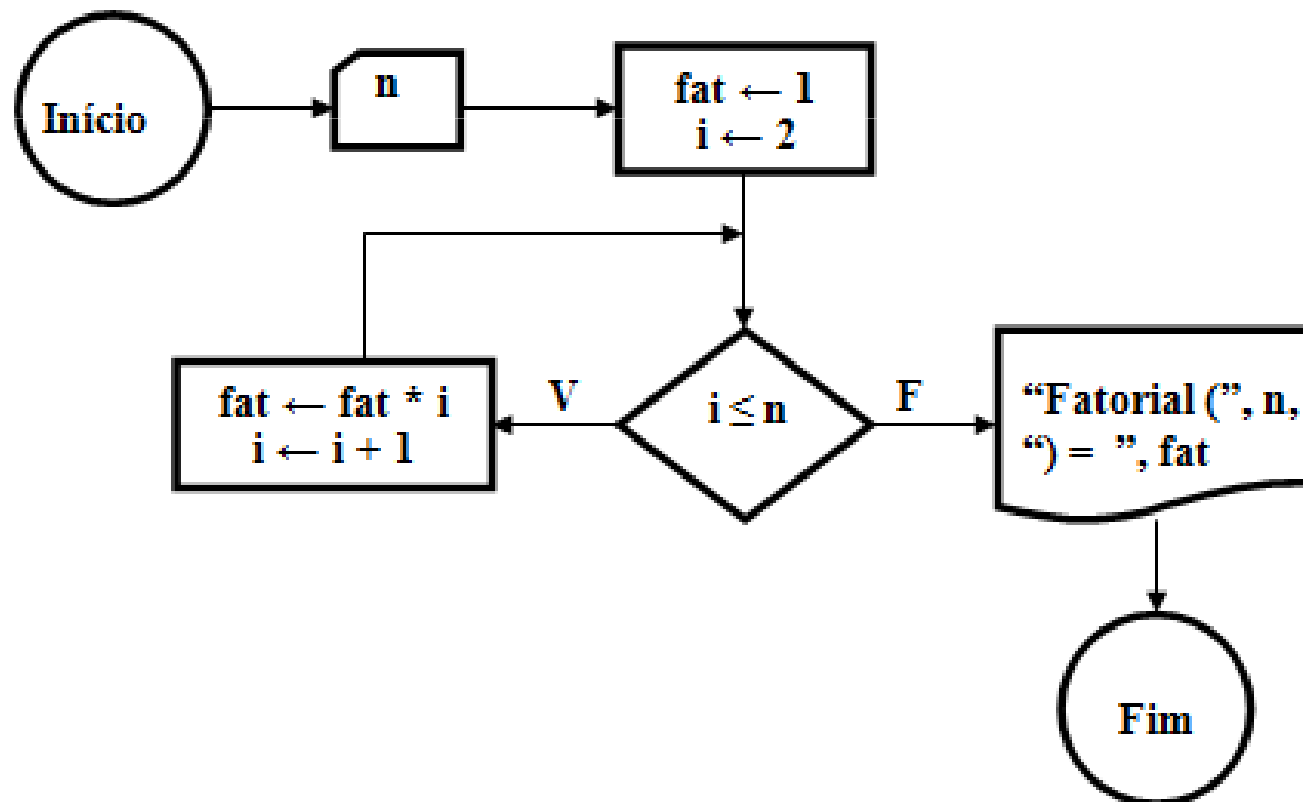
**Relação** entre fábricas:  $F_i$  fornece produtos para  $F_j$



## b) Fluxogramas

**Objetos:** blocos de um fluxograma

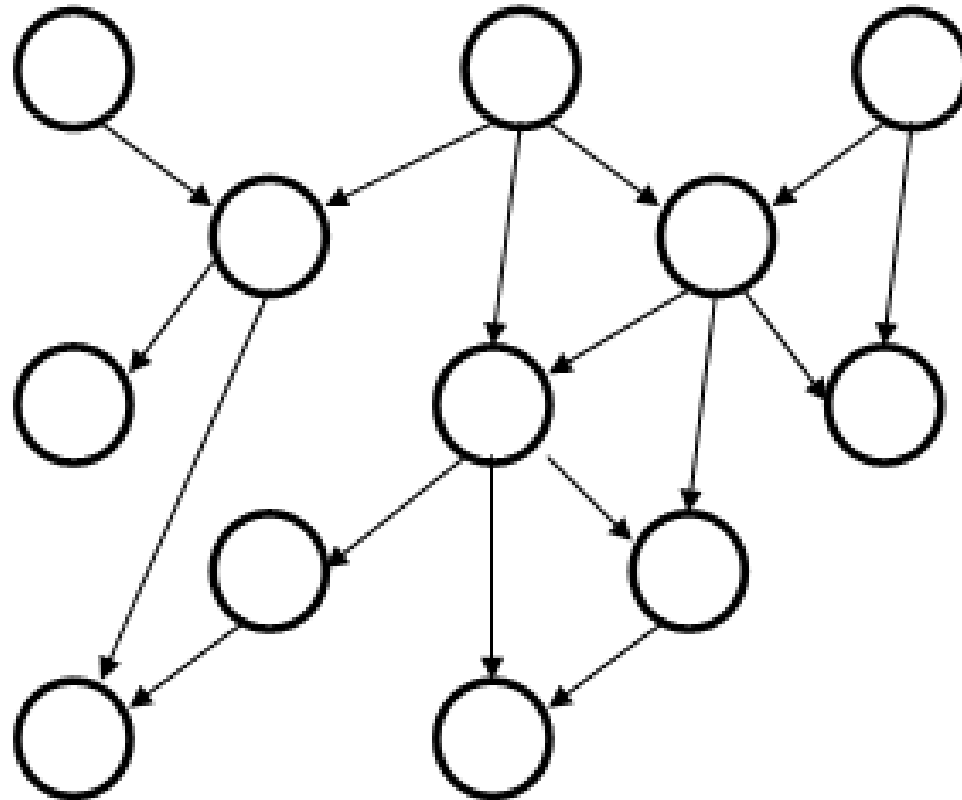
**Relação** entre blocos: o fluxo de controle vai de  $B_i$  para  $B_j$



## c) Redes PERT-CPM

**Objetos:** Tarefas de um projeto

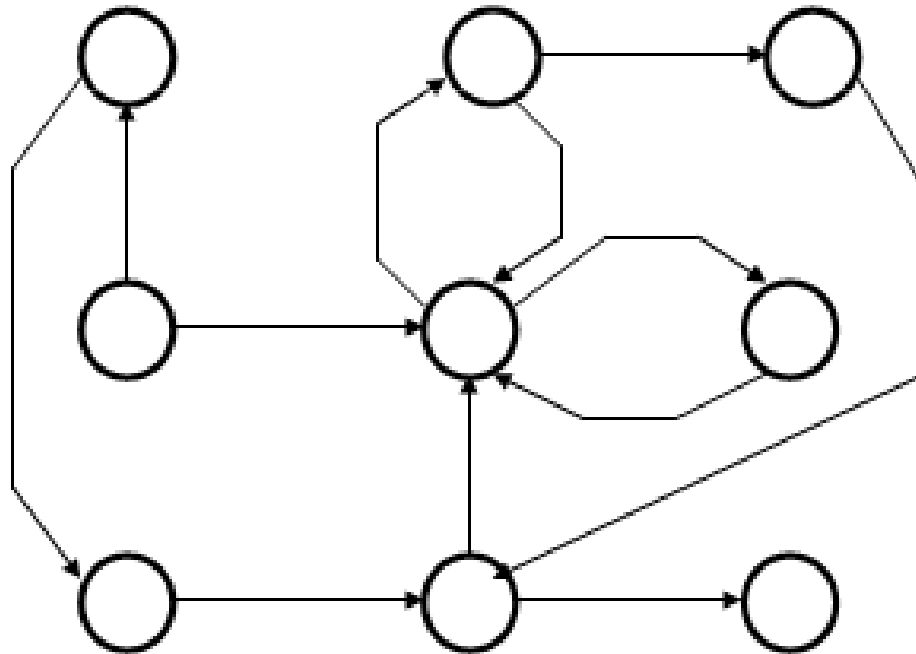
**Relação** entre tarefas: a tarefa  $T_i$  precisa terminar para que a tarefa  $T_j$  possa começar ( $T_j$  depende de  $T_i$ )



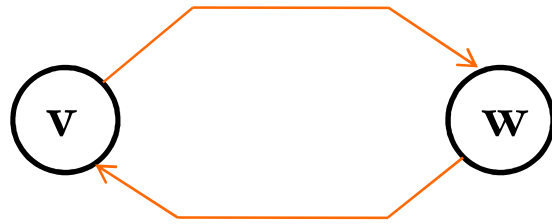
## d) Conhecimento entre pessoas

**Objetos:** Pessoas

**Relação** entre pessoas: a pessoa  $P_i$  conhece a pessoa  $P_j$  de vista e de nome



**Relação simétrica:** a existência da relação de um vértice  $v$  para outro  $w$  implica na existência da relação de  $w$  para  $v$



Na forma natural, costuma-se substituir os dois arcos orientados (setas), por um só arco não-orientado (linha)





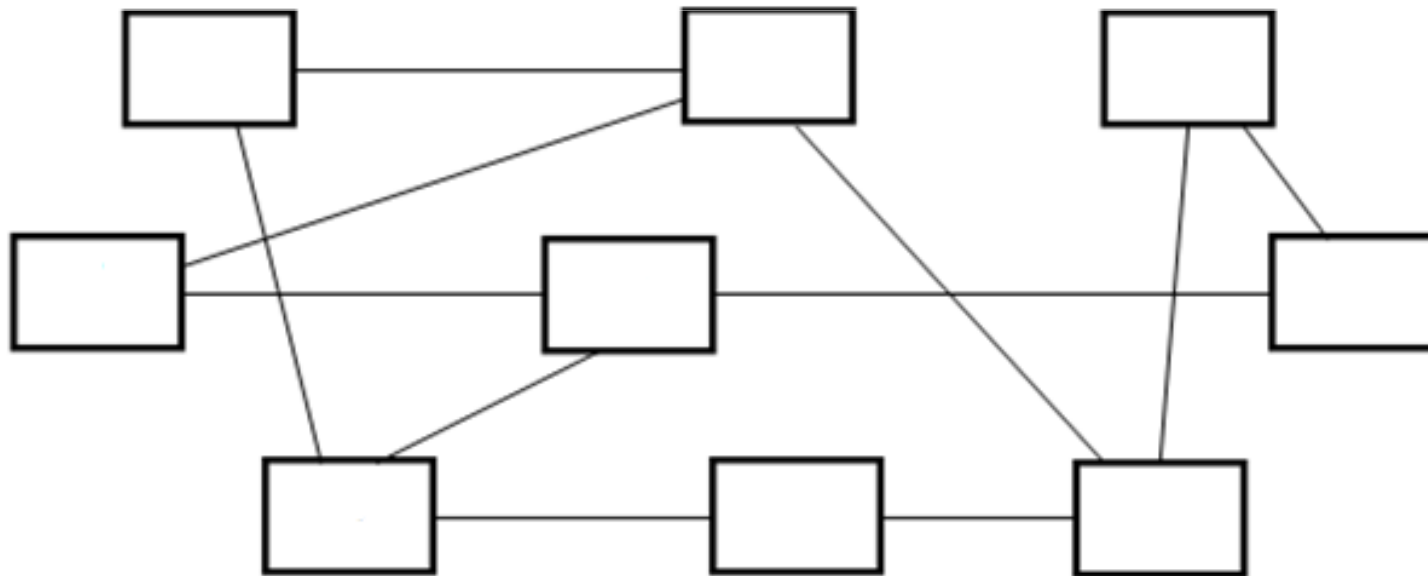
- **Grafo orientado:** sua relação é não-simétrica; vide os quatro exemplos anteriores
  - Fornecimento de produtos entre fábricas
  - Fluxogramas
  - Redes PERT-CPM
  - Conhecimento entre pessoas
- **Grafo não-orientado:** sua relação é simétrica

## Exemplos de grafos não-orientados:

a) Mapa rodoviário de uma região

**Objetos:** cidades

**Relação** entre cidades: existe uma rodovia ligando  $C_i$  e  $C_j$

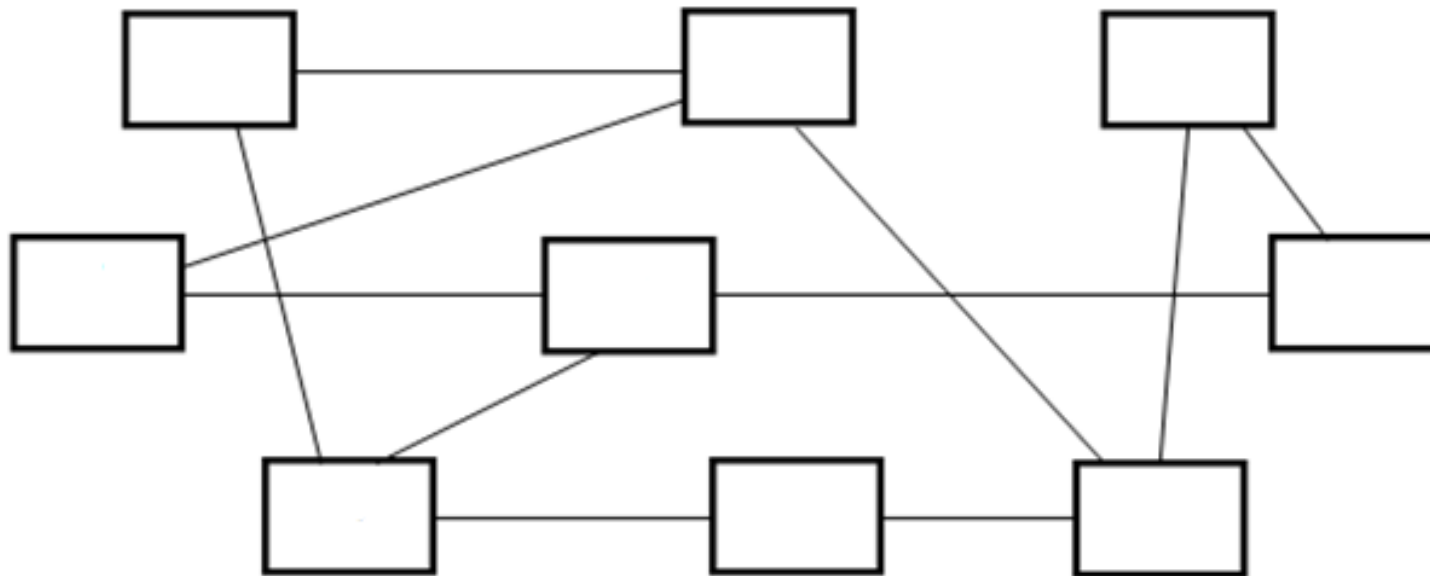


## Exemplos de grafos não-orientados:

b) Interseção entre conjuntos

**Objetos:** conjuntos

**Relação** entre conjuntos: conjuntos  $C_i$  e  $C_j$  têm elementos em comum



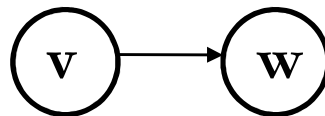
# 2 – GRAFOS ORIENTADOS

## 2.1 – Definições

- Um grafo orientado também é chamado de grafo **dirigido**, ou abreviadamente de **digrafo**
- Num grafo orientado, os arcos são **setas (linhas ou arestas orientadas)**

- Um arco é definido por um par orientado de vértices  $(\mathbf{v}, \mathbf{w})$ , onde  $\mathbf{v}$  é a **fonte** e  $\mathbf{w}$  é o **destino** do arco

$$(\mathbf{v}, \mathbf{w}) = \mathbf{v} \rightarrow \mathbf{w}$$



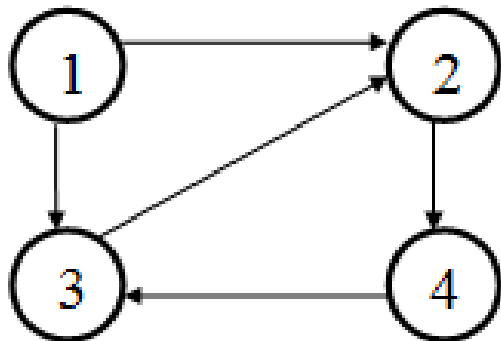
- Diz-se que o arco  $\mathbf{v} \rightarrow \mathbf{w}$  é **de v para w** e que **w é adjacente a v**

- **Caminho** em um digrafo é uma seqüência de vértices

$v_1, v_2, \dots, v_n$ , tais que

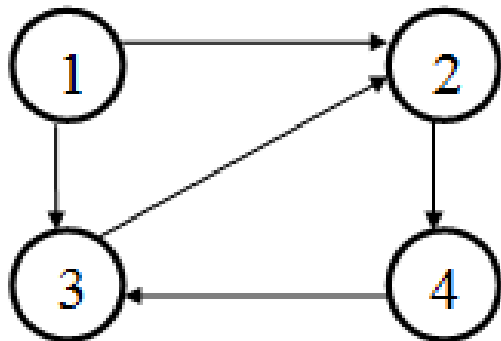
$v_1 \rightarrow v_2, v_2 \rightarrow v_3, \dots, v_{n-1} \rightarrow v_n$  são arcos;

- Tal caminho é dito que vai de  $v_1$  para  $v_n$ , passa por  $v_2, v_3, \dots, v_{n-1}$  e termina em  $v_n$ .
- **Comprimento** de um caminho é o número de arcos desse caminho. Exemplo:



1, 2, 4: caminho de comprimento 2, de 1 a 4

- **Caminho simples:** caminho no qual todos os vértices são distintos, com a exceção possível do 1º e do último vértice
- **Ciclo:** caminho no qual o 1º e o último vértice são iguais.
- **Ciclo simples:** ciclo que também é um caminho simples.  
Exemplo:



3, 2, 4, 3: ciclo simples  
de comprimento 3

## **Informações** em um grafo (orientado ou não):

- Podem estar contidas nos vértices e nos arcos.

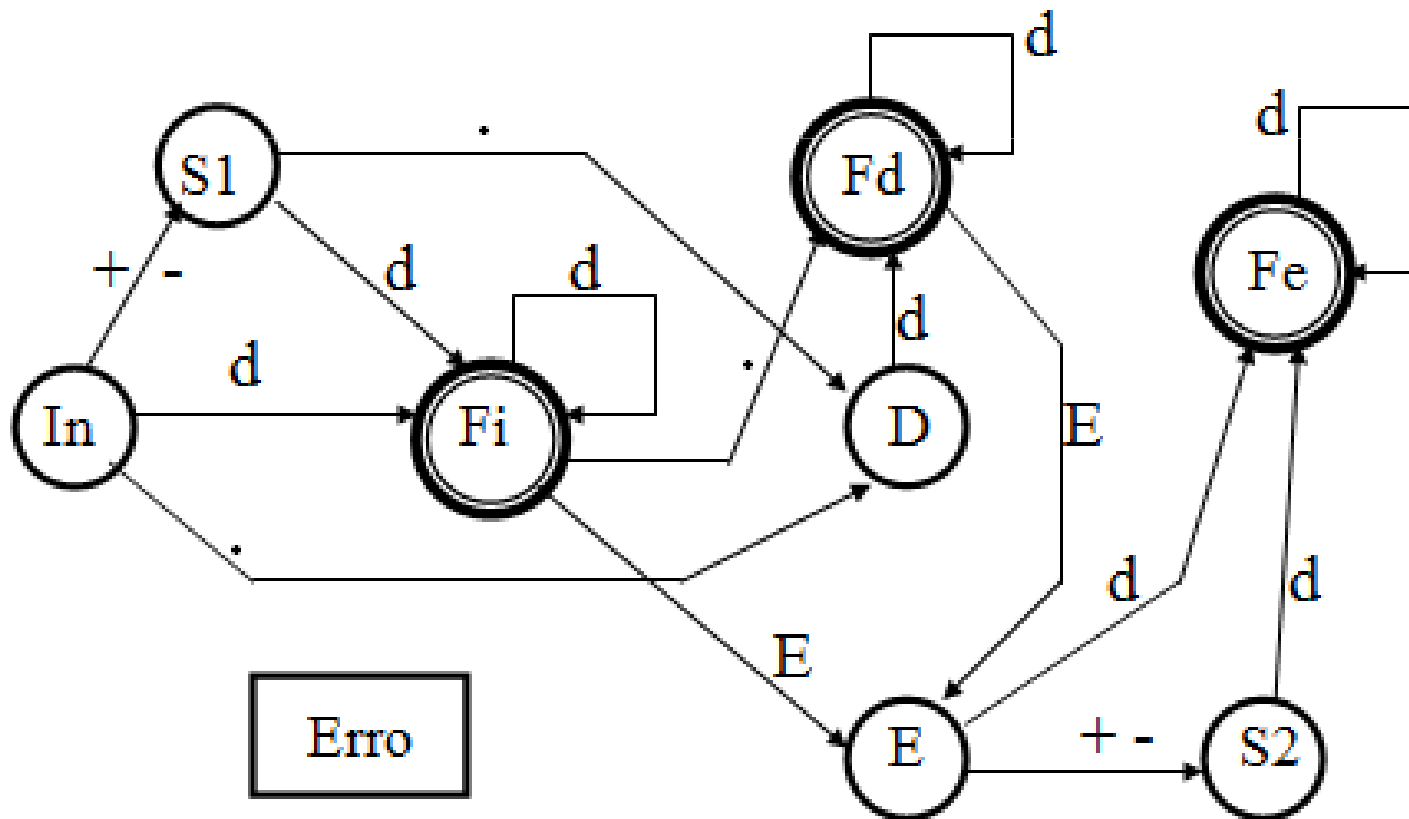
## **Exemplo:** fornecimento de produtos entre fábricas

- **Nos vértices:** nome da fábrica; localização; número de empregados, etc...
- **Nos arcos:** qual o produto; quantidade de produtos fornecidos; custo de cada produto, etc.



## Exemplo: Autômato finito reconhecedor de constantes numéricas na Linguagem C

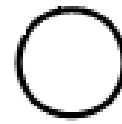
- Reconhece constantes como: 13, +13, -13, 13., 1.25, .25, -.25, -32.43, 13E-15, 13.E-15, -13.25E+72, .75E5



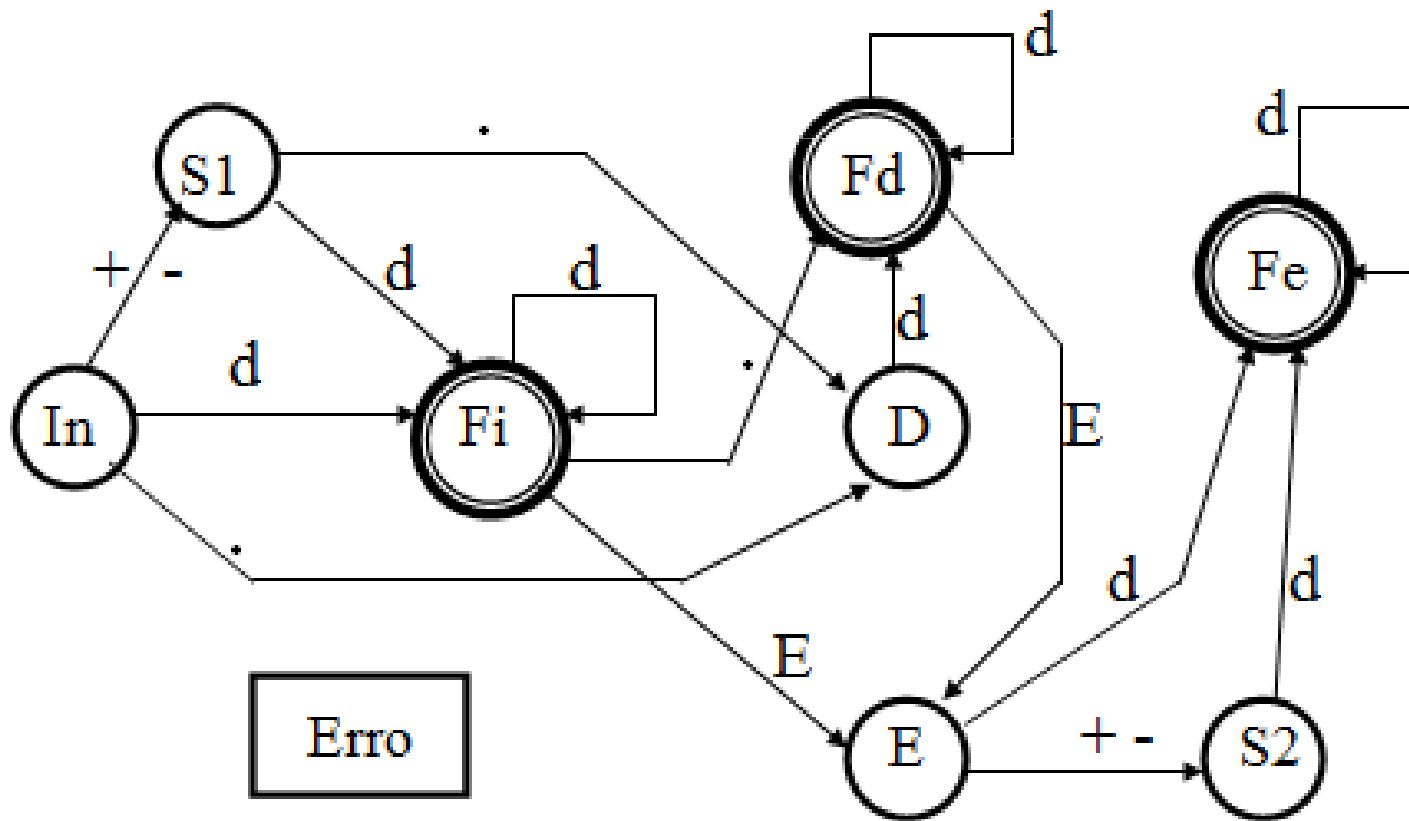
○ **Vértices:** círculos representando estados do autômato



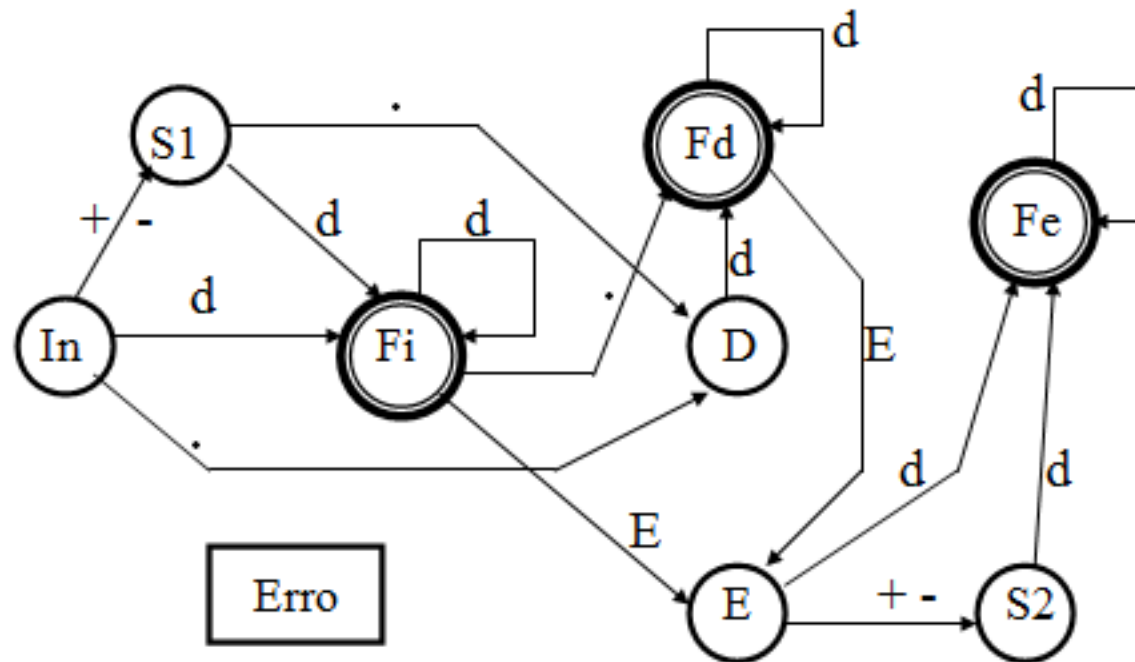
Estado final



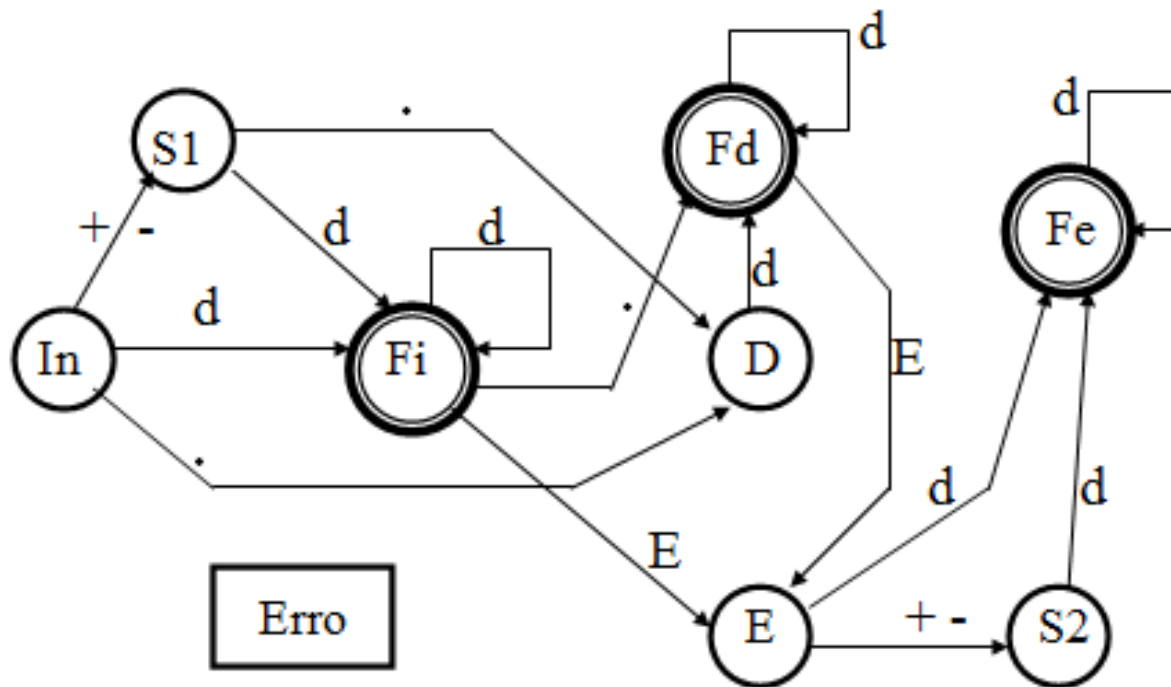
Estado não-final



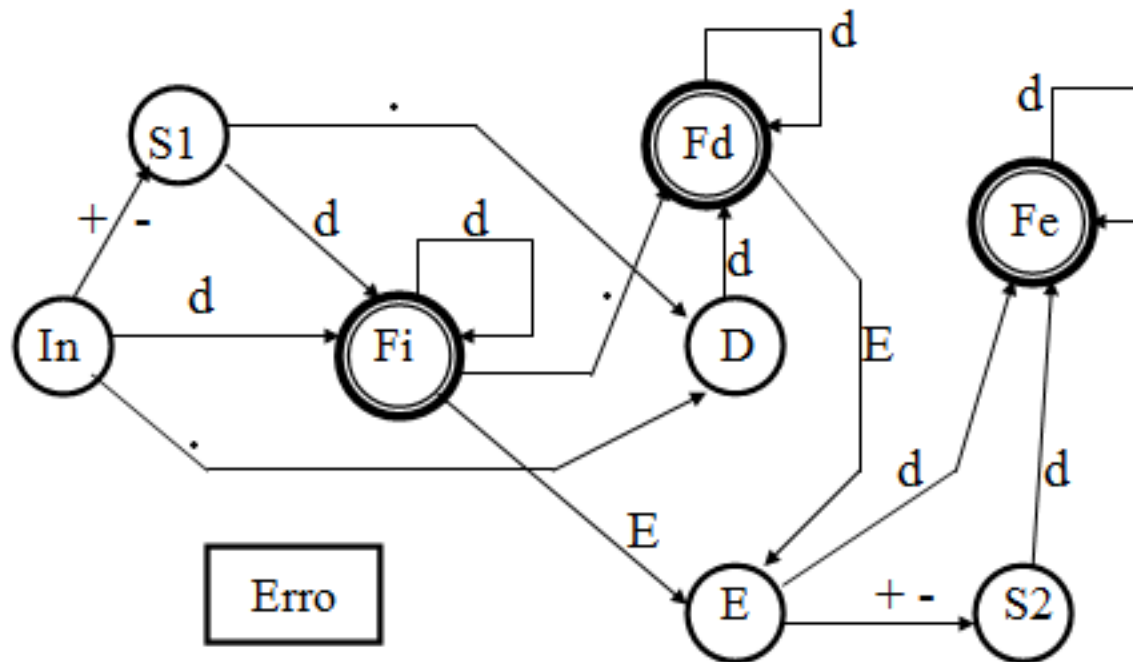
- Informações nos **vértices**: nome, se é estado inicial, central ou final
- Informação nos **arcos**: caractere(s) necessário(s) para transitar de um estado para outro.



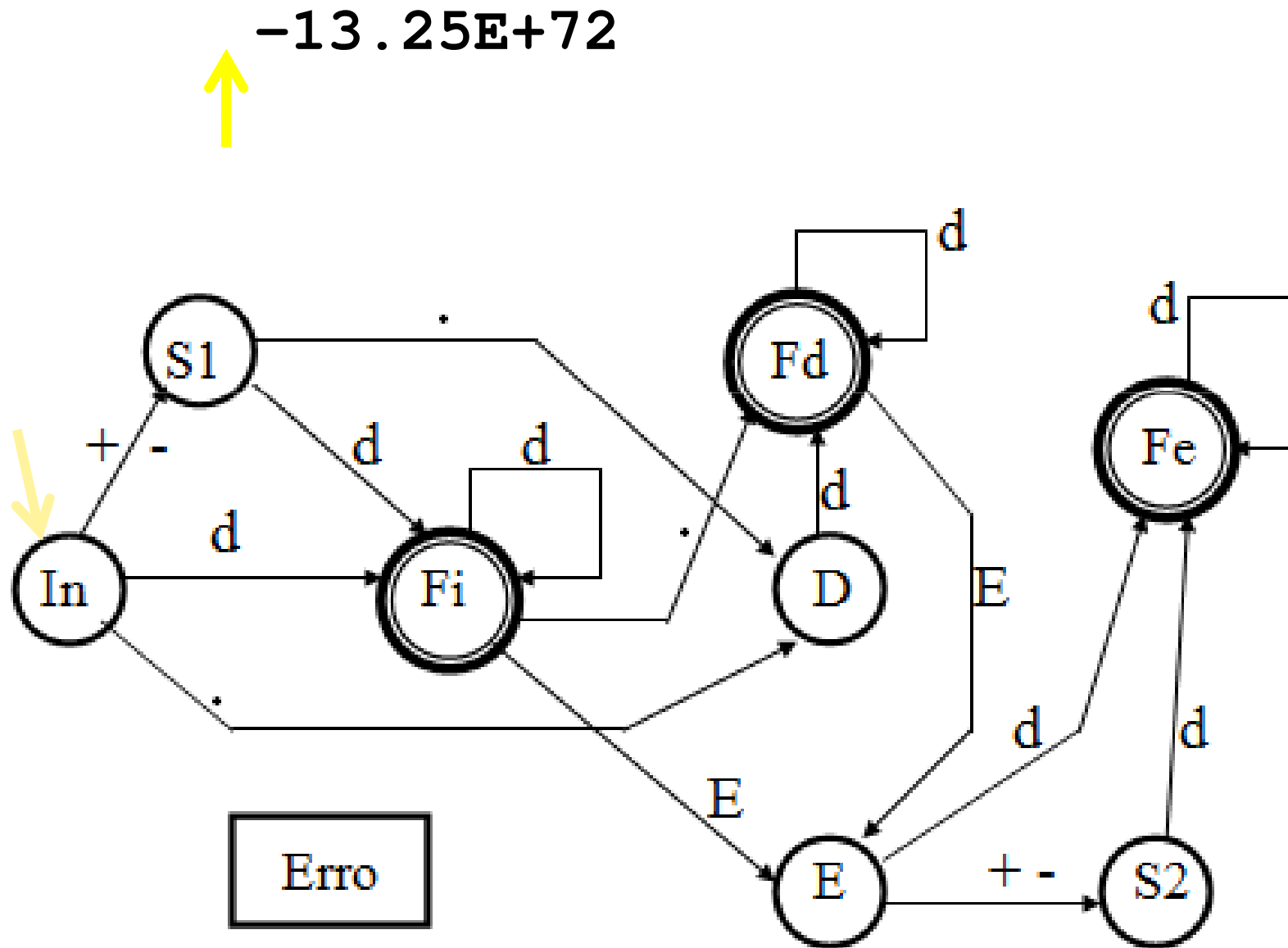
- **Arcos:** setas representando transições de estados
- Alfabeto para transições:  
 $\Sigma = \{d, +, -, ., E\}$  (d é um dígito qualquer)
- Existem transições de todos os estados para o estado **Erro** mas não estão desenhadas, para maior clareza



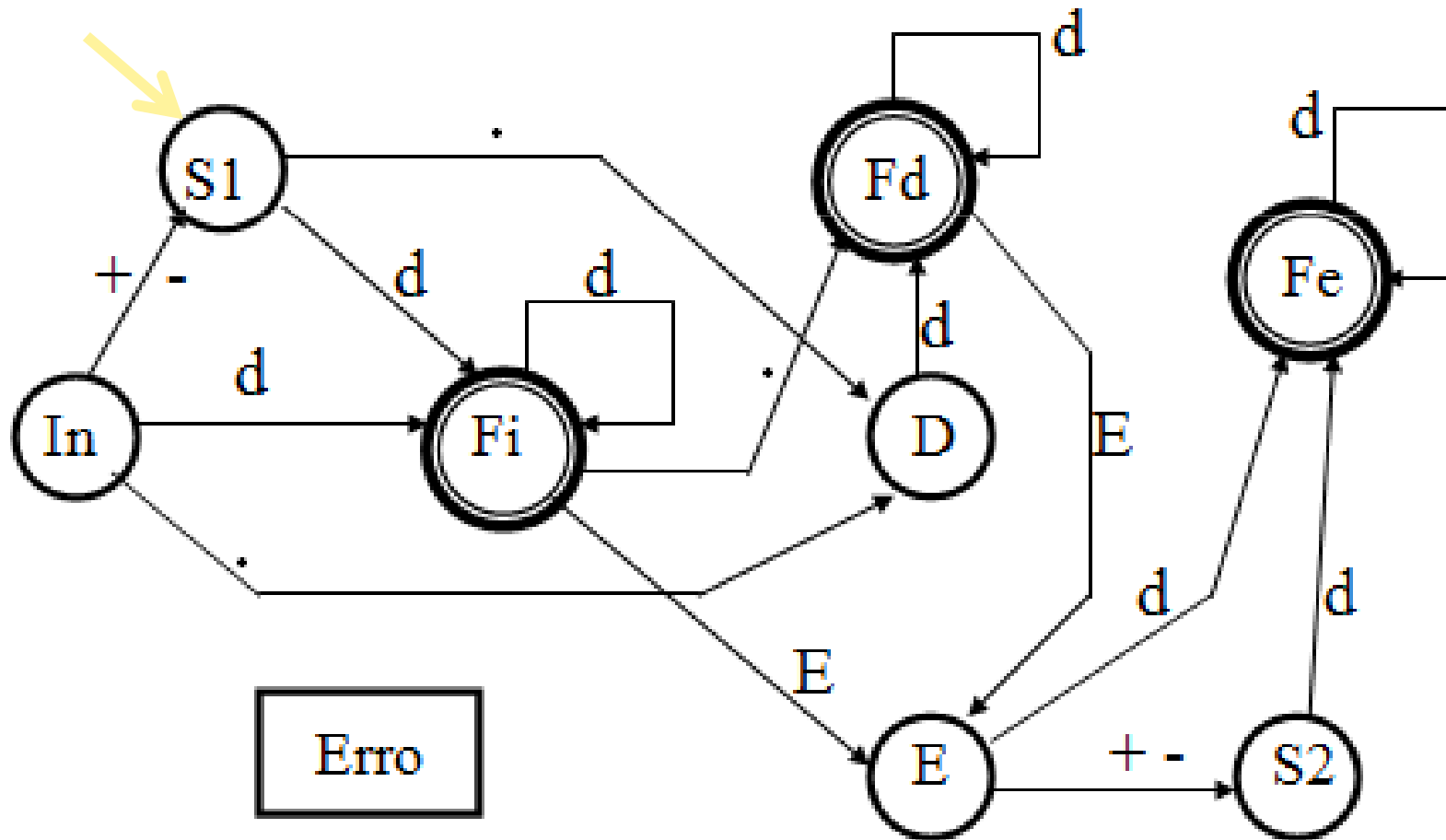
- Cada caractere da constante numérica provoca uma mudança de estado
- No início, o estado é o **In** (inicial)
- Se, no final da constante, o estado for do tipo **final**, ela é reconhecida; caso contrário, é rejeitada



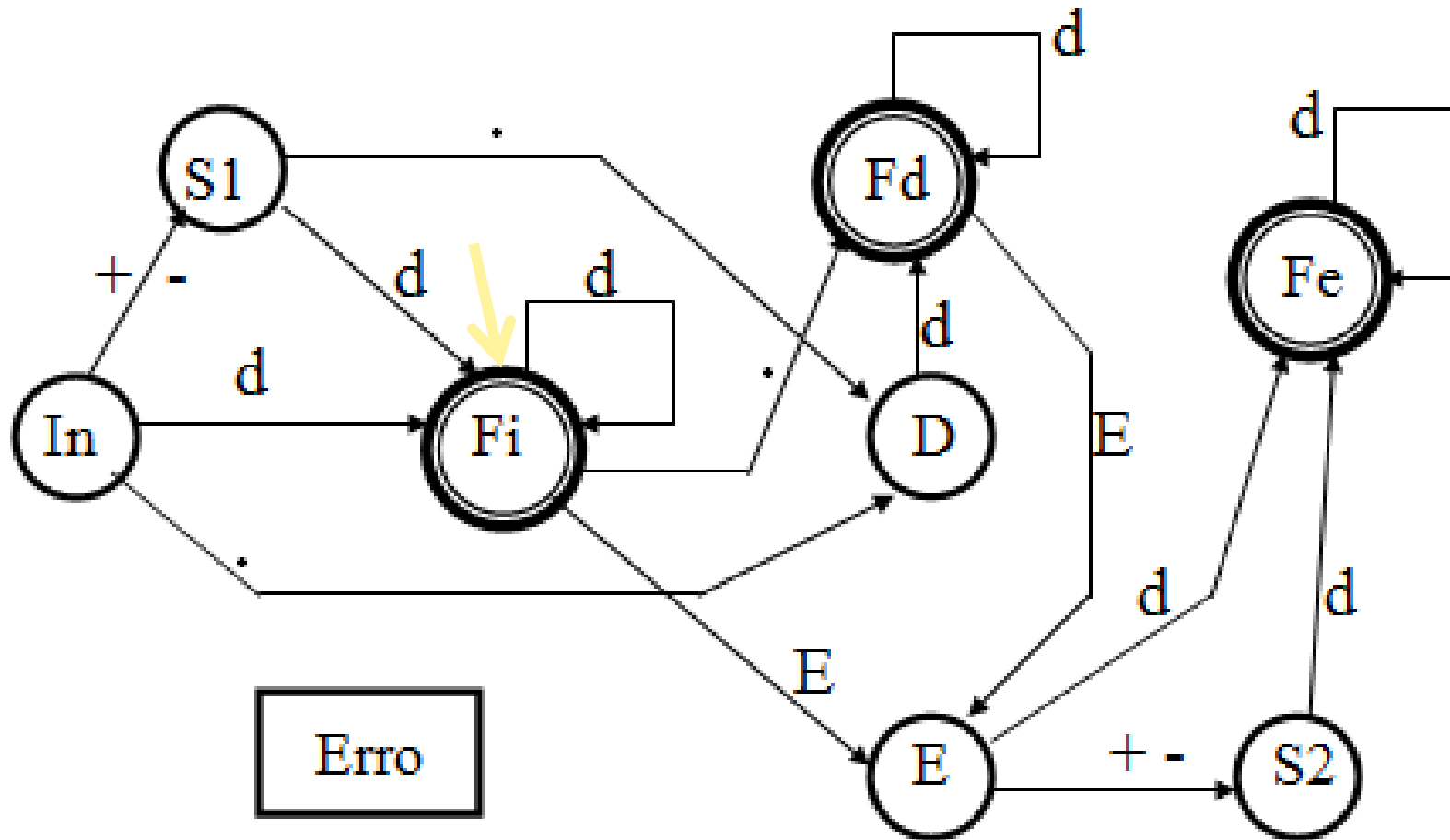
○ Simulação do teste da constante  $-13.25E+72$



$-13.25E+72$

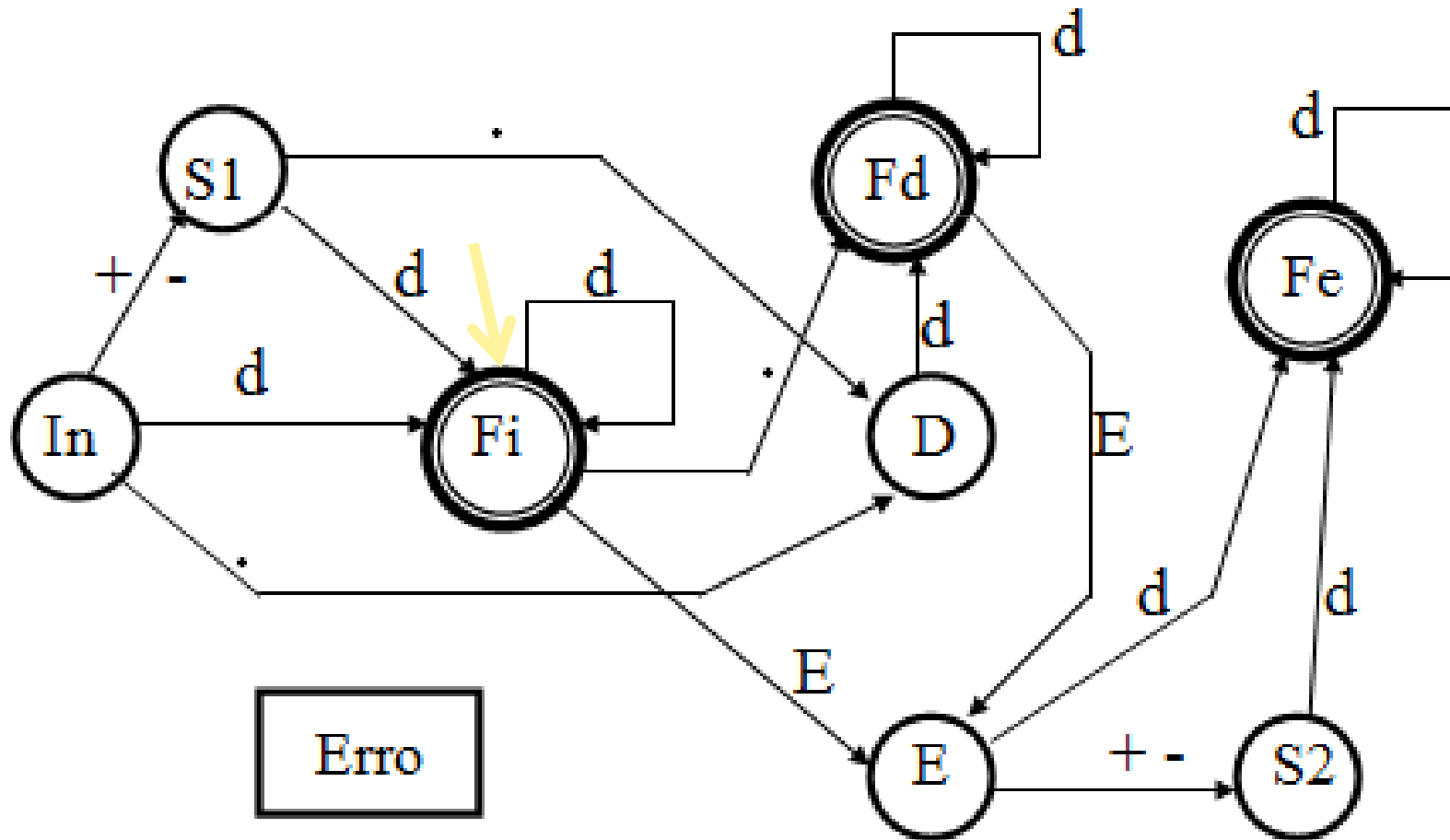


-13.25E+72

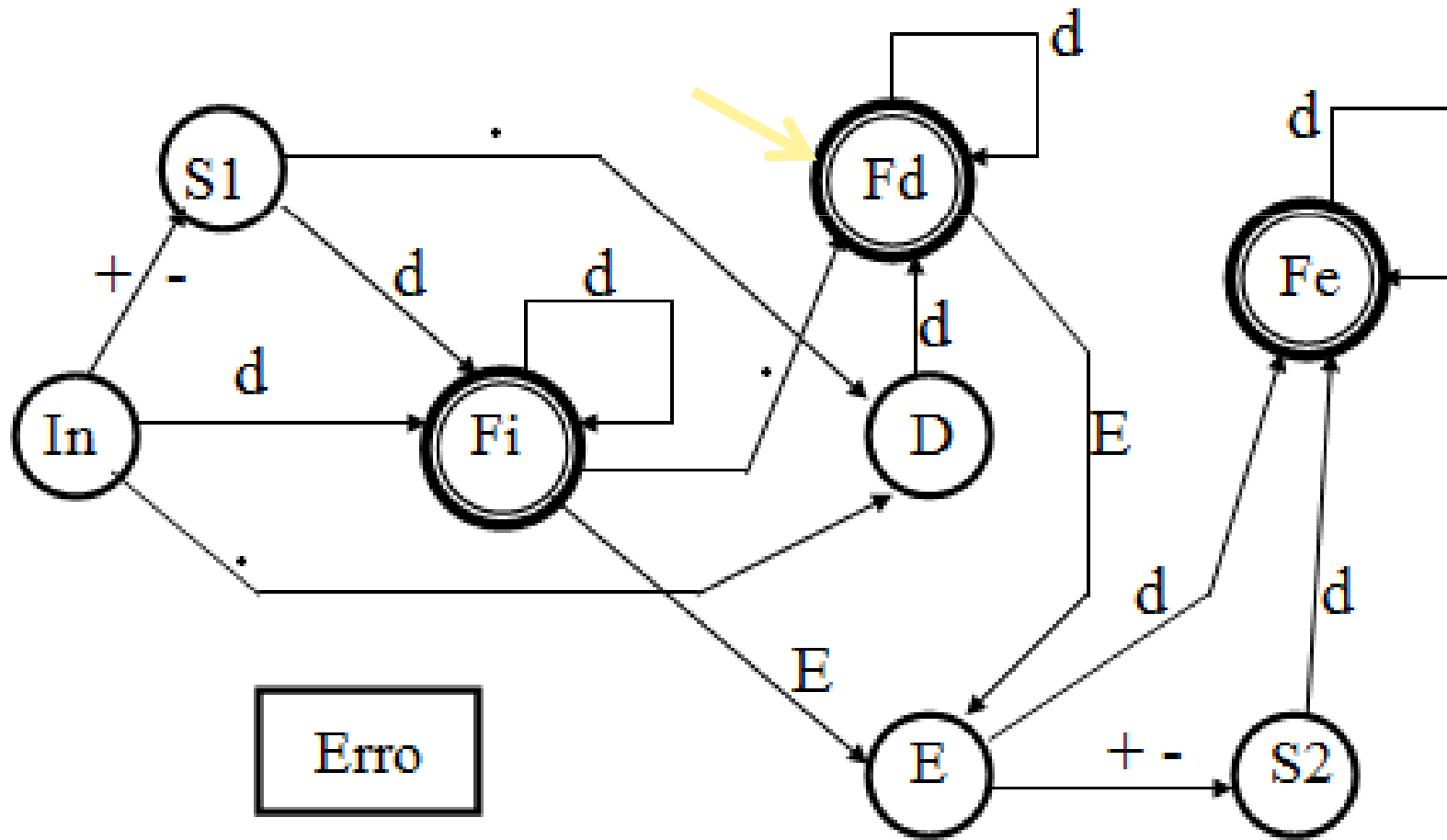




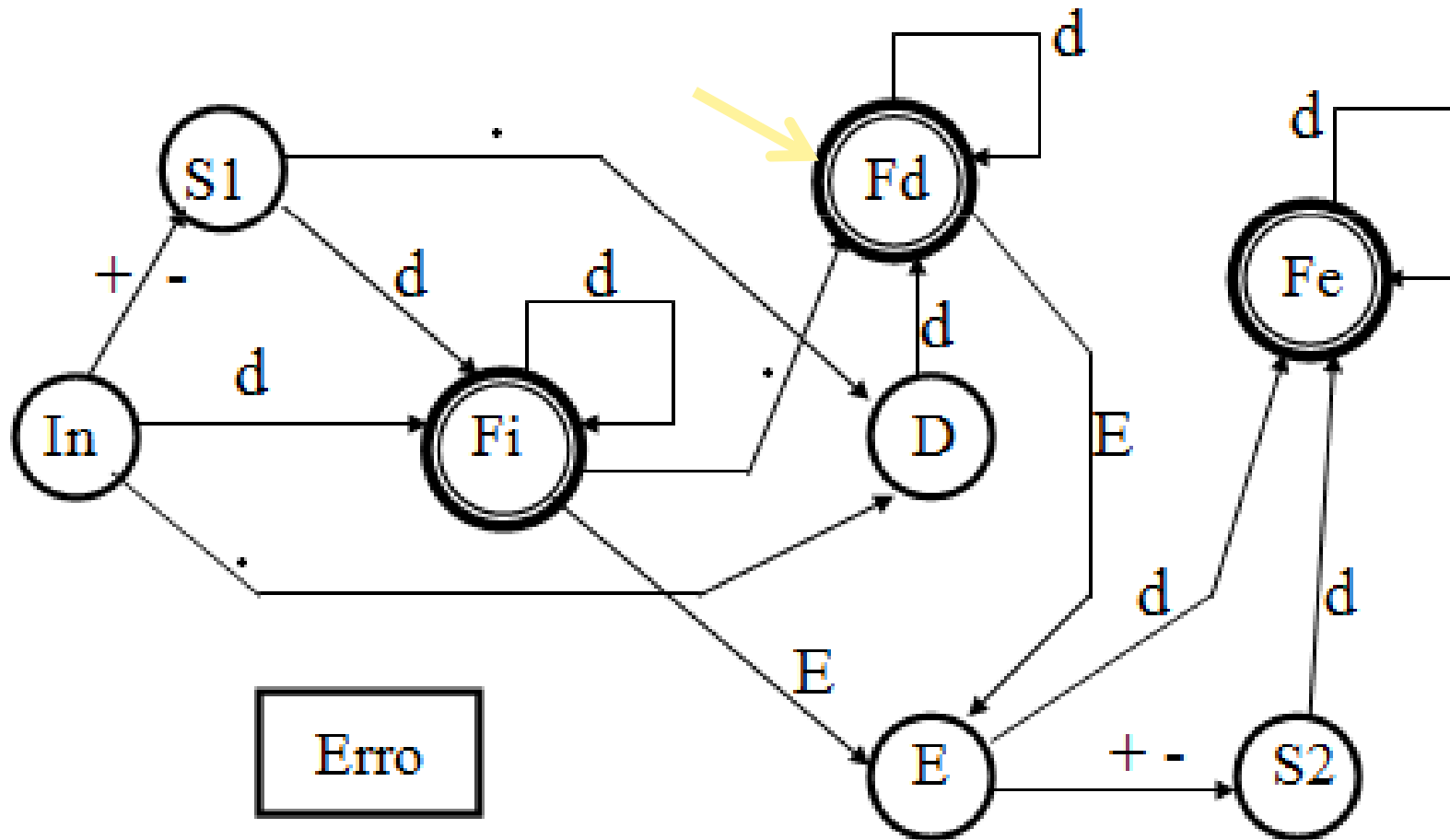
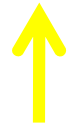
$-13.25E+72$



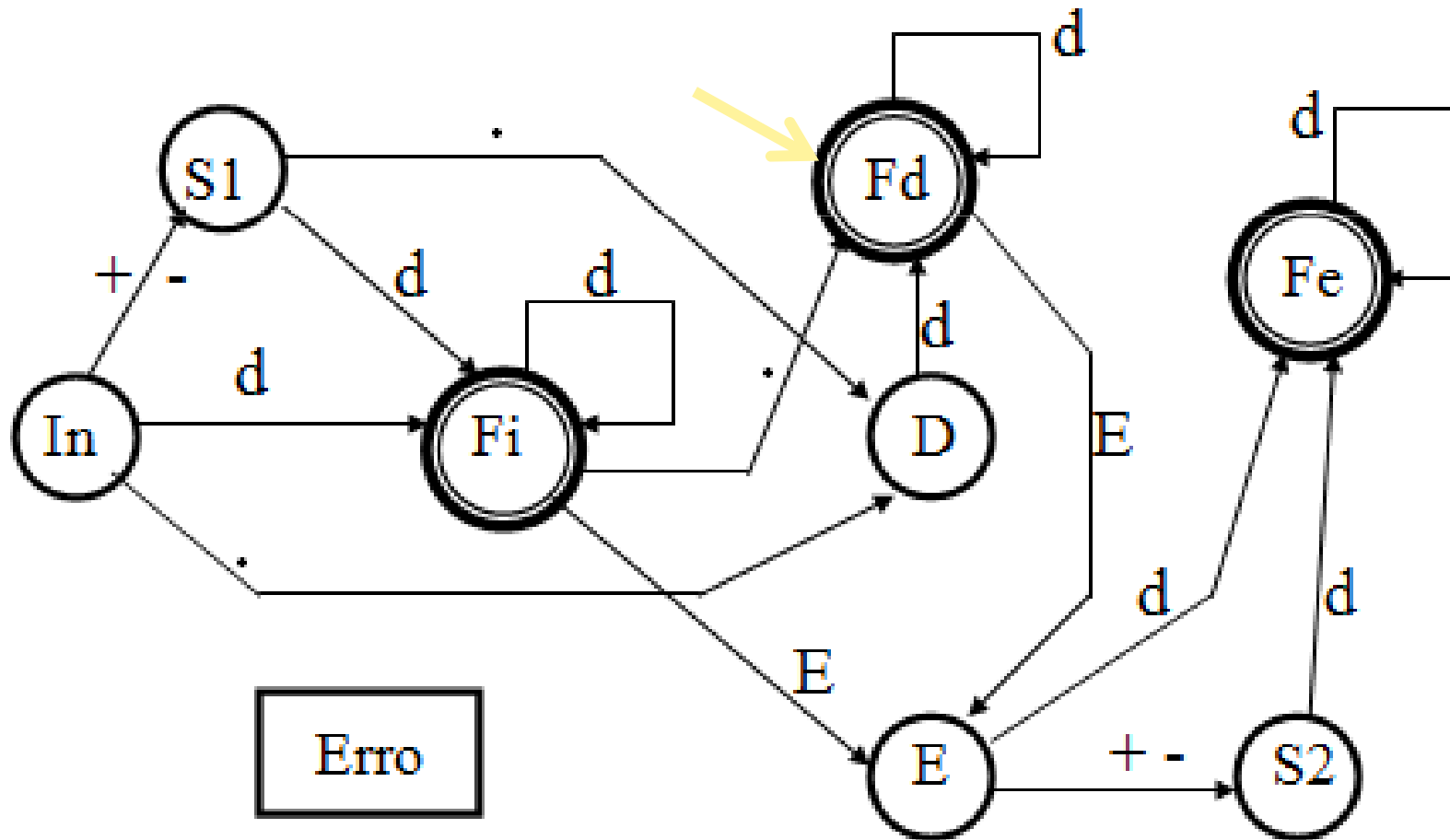
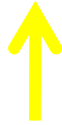
$-13.25E+72$



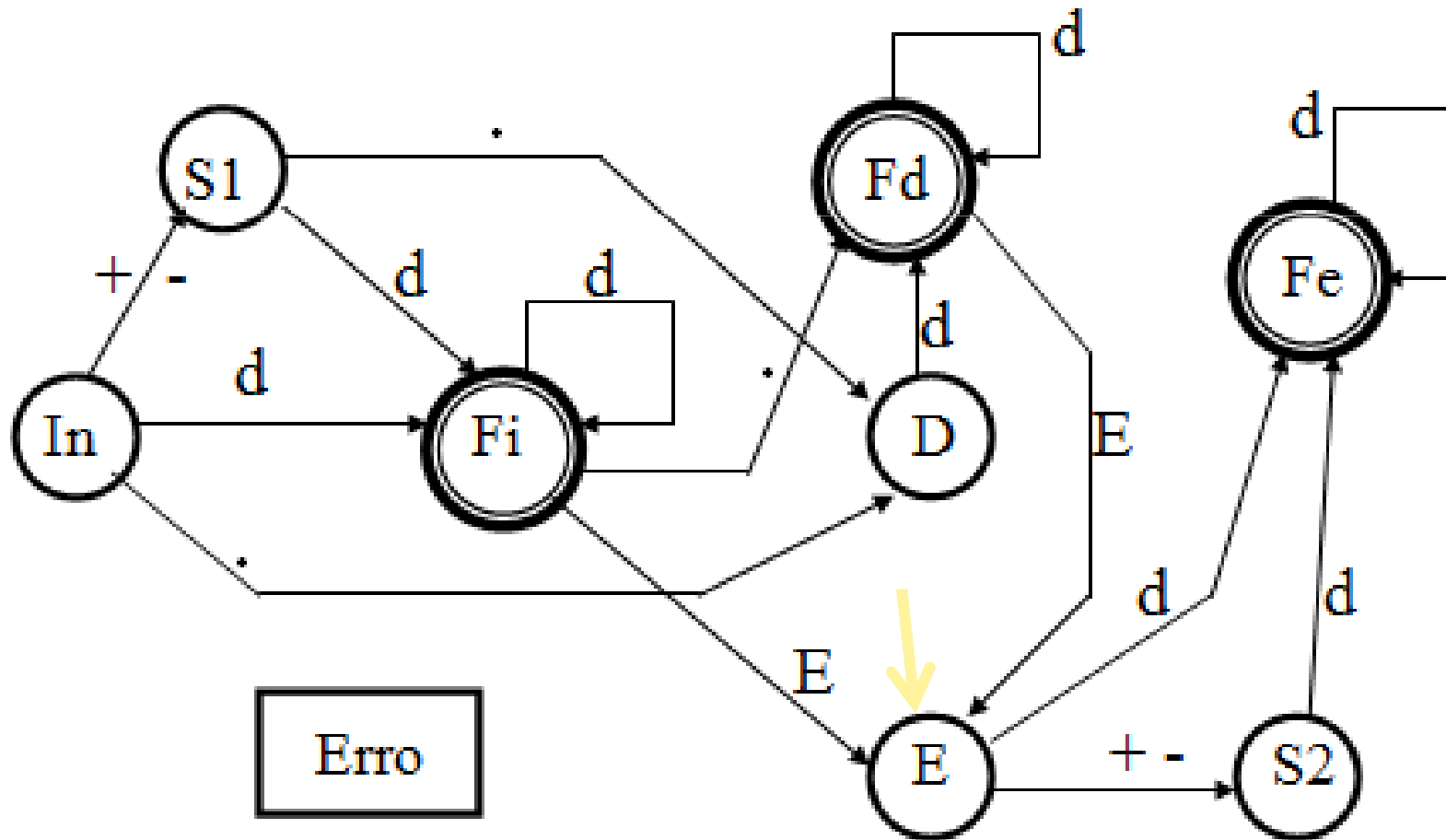
-13.25E+72



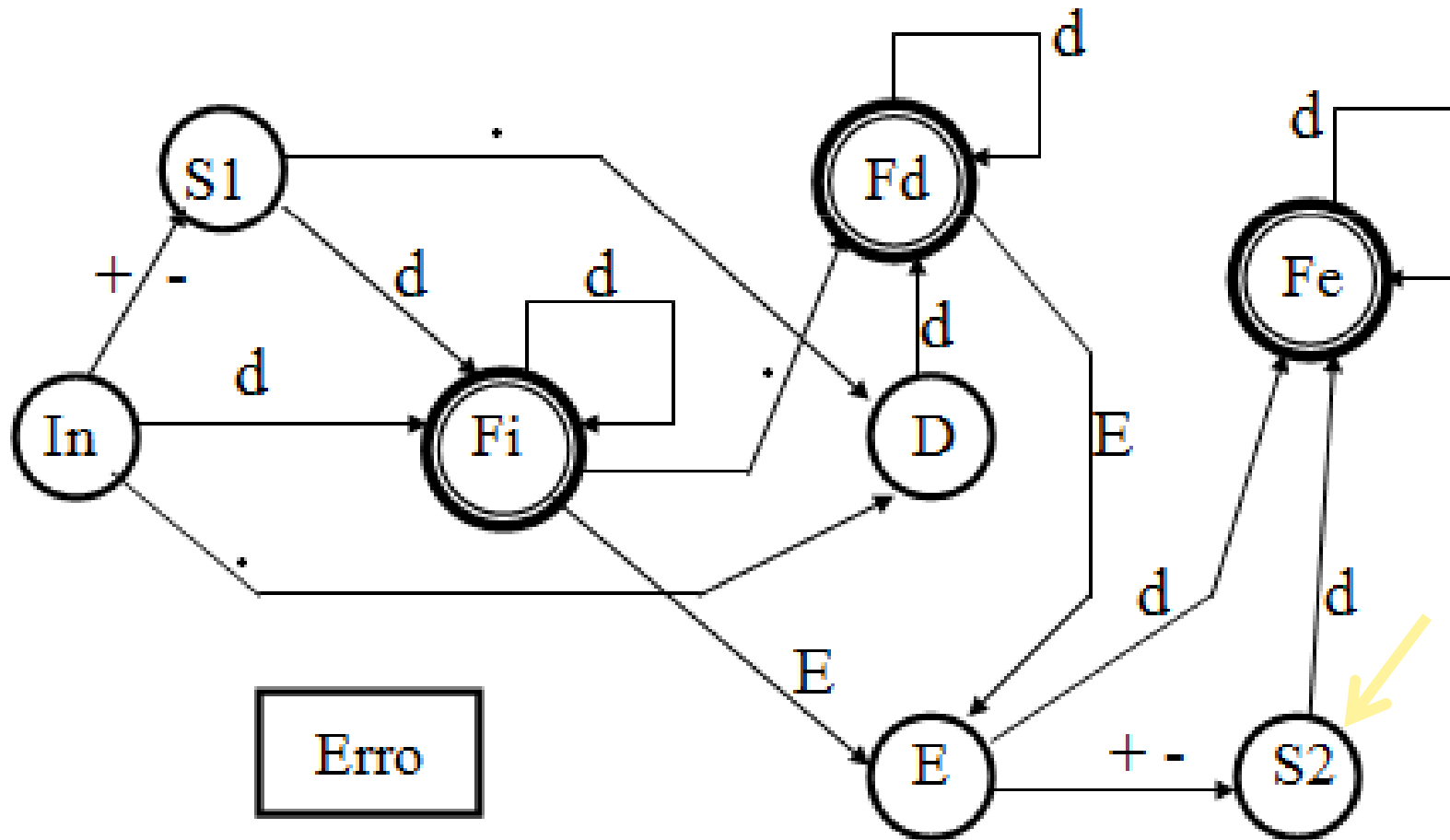
$-13.25E+72$



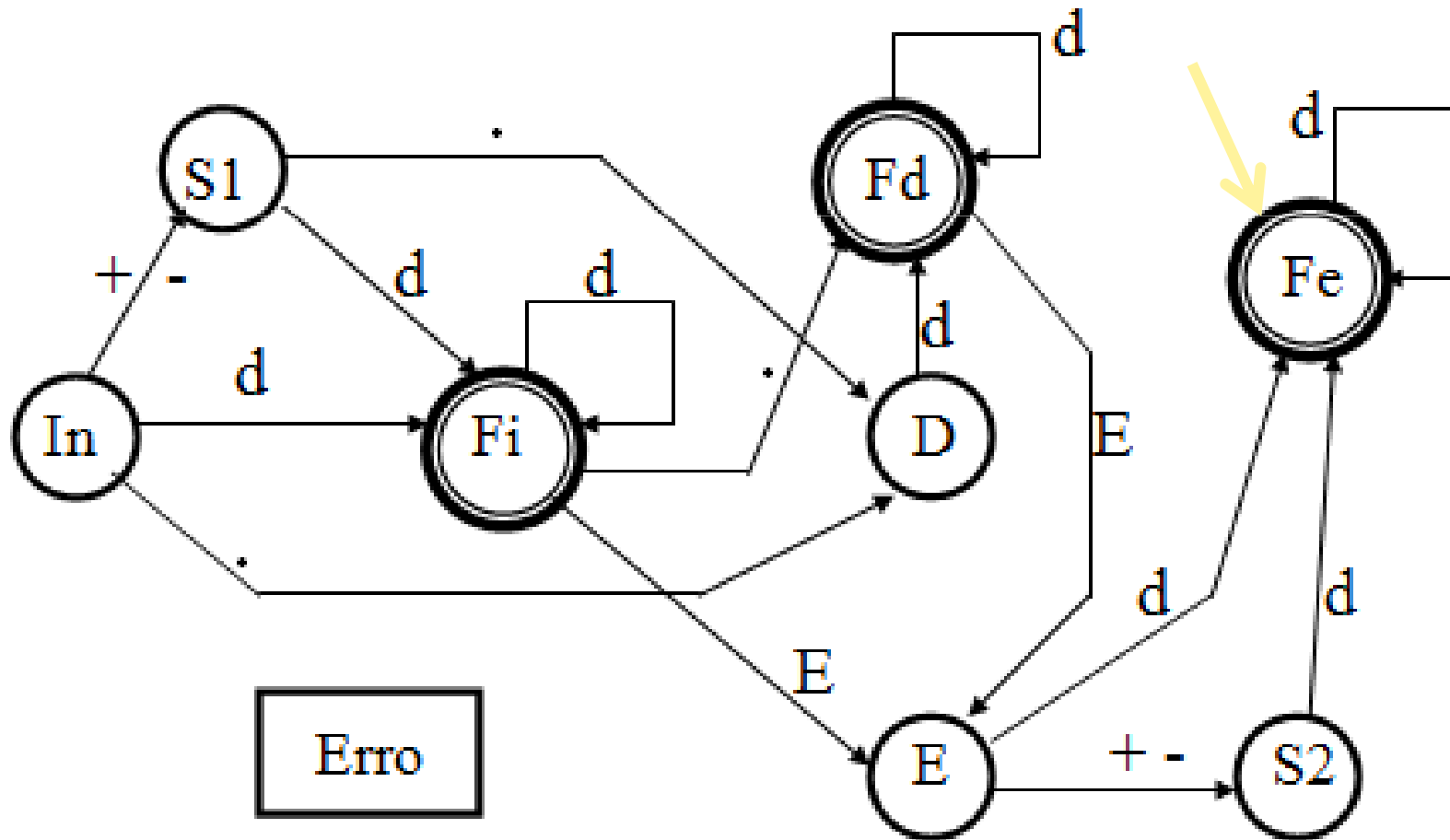
$-13.25E+72$



$-13.25E+72$

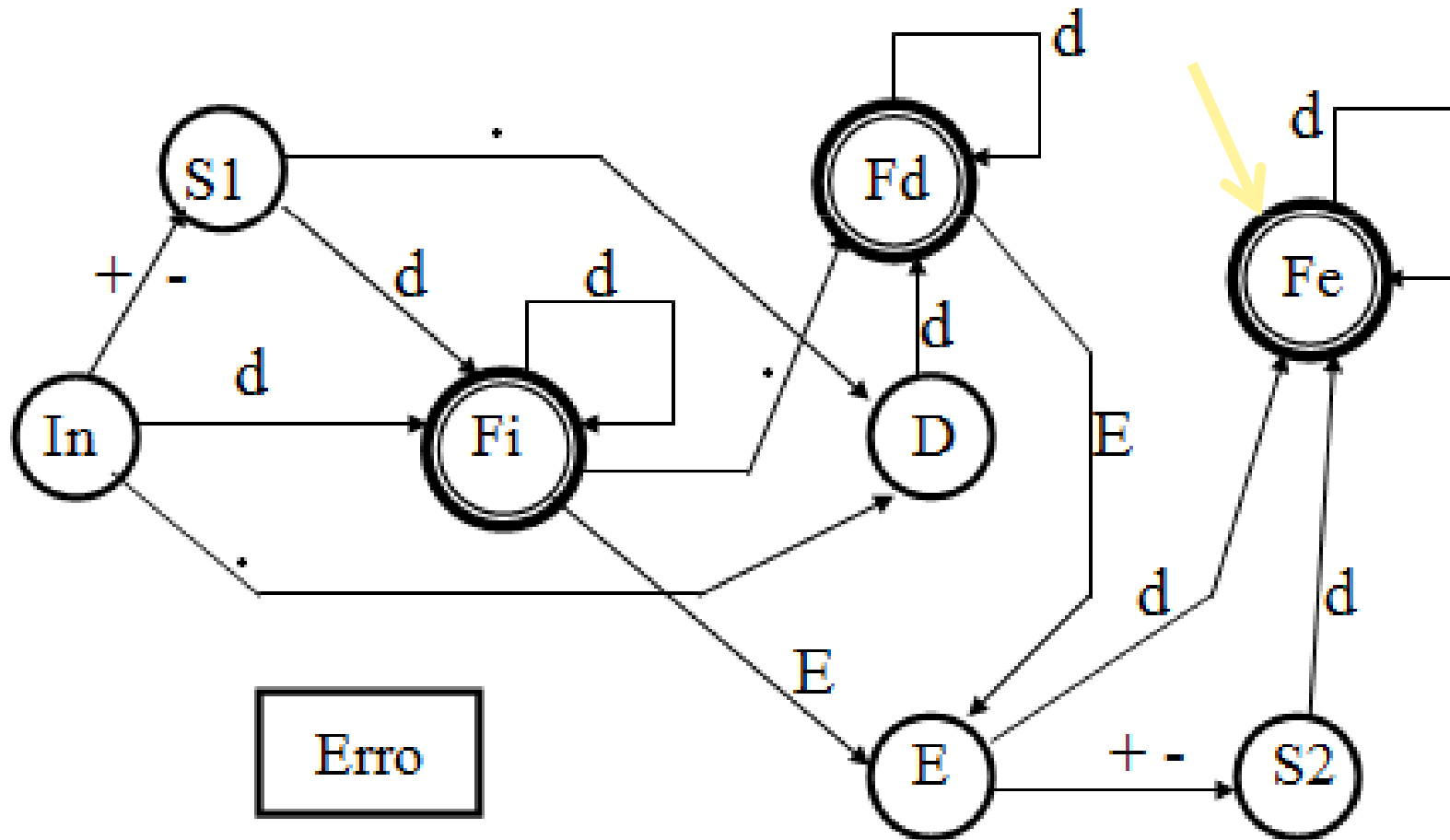


-13.25E+72



O estado é final  
A constante é reconhecida

-13.25E+72

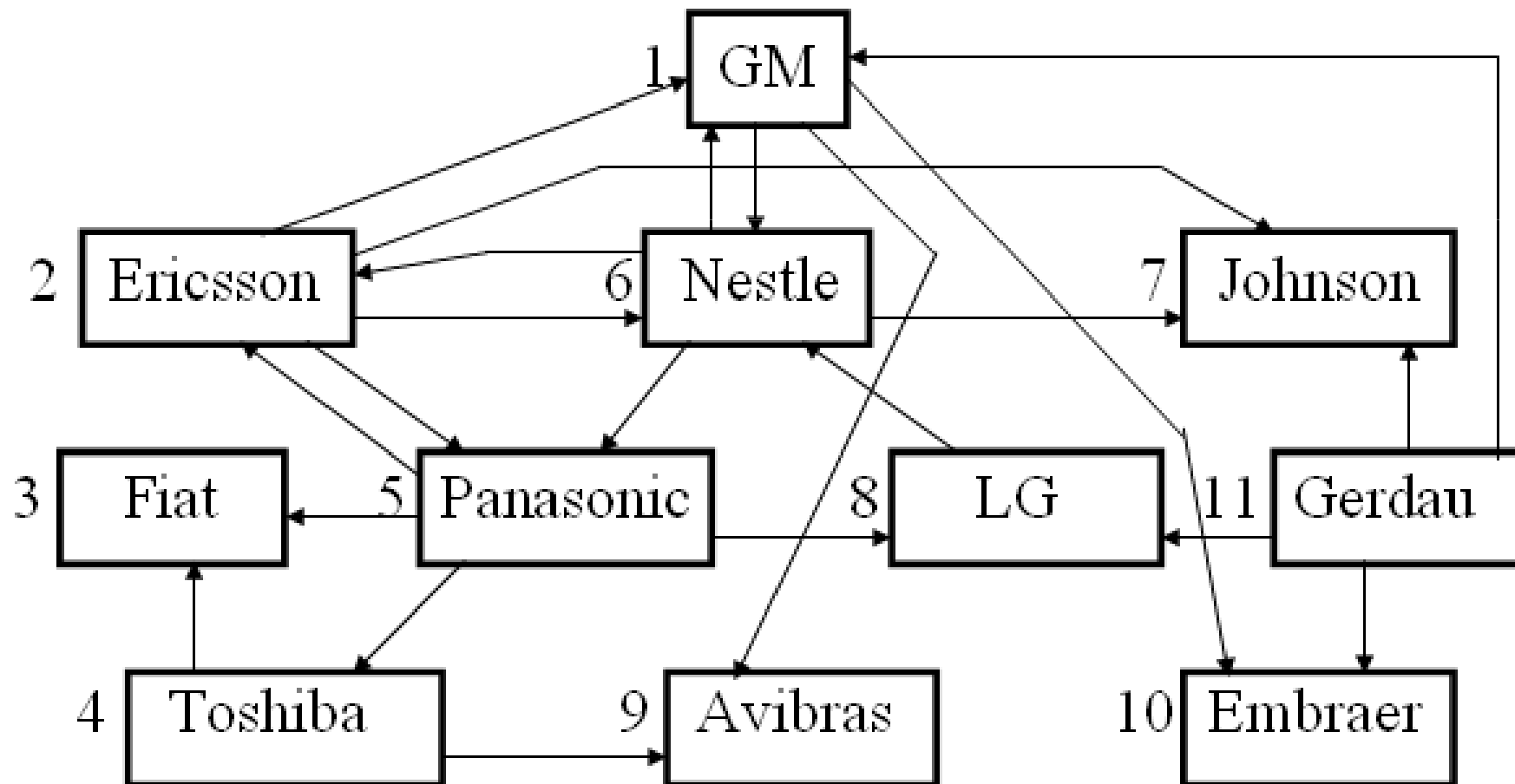


Erro



## 2.2 – Estruturas de dados para digrafos

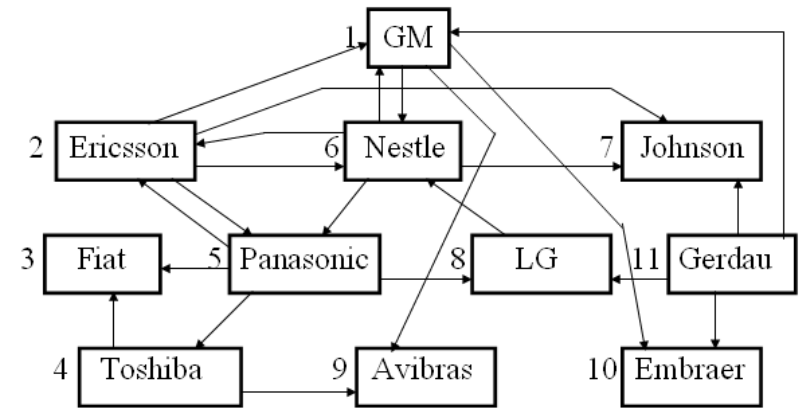
- Seja um grafo de fornecimento de produtos entre fábricas



## a) Matriz de adjacências

Os elementos da matriz são do tipo lógico

Se elemento  $[i, j] = V$ , fábrica  $i$  fornece produto para fábrica  $j$



|    | Fábrica   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|----|-----------|---|---|---|---|---|---|---|---|---|----|----|
| 1  | GM        |   |   |   |   |   | V |   |   | V | V  |    |
| 2  | Ericsson  | V |   |   |   | V | V | V |   |   |    |    |
| 3  | Fiat      |   |   |   |   |   |   |   |   |   |    |    |
| 4  | Toshiba   |   |   | V |   |   |   |   |   | V |    |    |
| 5  | Panasonic |   | V | V | V |   |   |   | V |   |    |    |
| 6  | Nestle    | V | V |   |   | V |   | V |   |   |    |    |
| 7  | Johnson   |   |   |   |   |   |   |   |   |   |    |    |
| 8  | LG        |   |   |   |   |   | V |   |   |   |    |    |
| 9  | Avibras   |   |   |   |   |   |   |   |   |   |    |    |
| 10 | Embraer   |   |   |   |   |   |   |   |   |   |    |    |
| 11 | Gerdau    | V |   |   |   |   |   | V | V |   | V  |    |

# Declarações para matriz de adjacências

```
struct celulaVertice {
    informacaoVertice InfoVert;
    logic Adjacente [maxvert];
};

struct digrafo {
    celulaVertice Vetor [maxvert];
    int nvert;
};

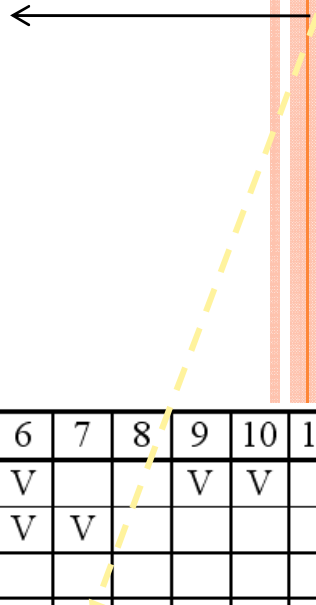
digrafo Fornecimento;
```

|    | Fábrica   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|----|-----------|---|---|---|---|---|---|---|---|---|----|----|
| 1  | GM        |   |   |   |   |   | V |   |   | V | V  |    |
| 2  | Ericsson  | V |   |   |   | V | V | V |   |   |    |    |
| 3  | Fiat      |   |   |   |   |   |   |   |   |   |    |    |
| 4  | Toshiba   |   |   | V |   |   |   |   |   | V |    |    |
| 5  | Panasonic |   | V | V | V |   |   |   | V |   |    |    |
| 6  | Nestle    | V | V |   |   | V |   | V |   |   |    |    |
| 7  | Johnson   |   |   |   |   |   |   |   |   |   |    |    |
| 8  | LG        |   |   |   |   |   | V |   |   |   |    |    |
| 9  | Avibras   |   |   |   |   |   |   |   |   |   |    |    |
| 10 | Embraer   |   |   |   |   |   |   |   |   |   |    |    |
| 11 | Gerdau    | V |   |   |   |   |   | V | V |   | V  |    |

Se houver informações nos arcos:

```
struct celulaVertice {  
    informacaoVertice InfoVert;  
    informacaoarco Adjacente [maxvert];  
};  
  
struct digrafo {  
    celulaVertice Vetor [maxvert];  
    int nvert;  
};  
  
digrafo Fornecimento;
```

**informacaoarco** pode ser até  
uma complexa estrutura



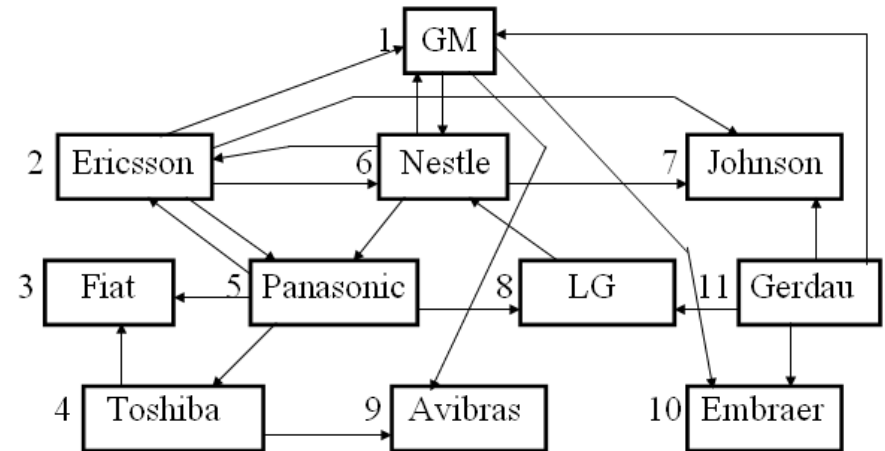
|    | Fábrica   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|----|-----------|---|---|---|---|---|---|---|---|---|----|----|
| 1  | GM        |   |   |   |   |   | V |   |   | V | V  |    |
| 2  | Ericsson  | V |   |   |   | V | V | V |   |   |    |    |
| 3  | Fiat      |   |   |   |   |   |   |   |   |   |    |    |
| 4  | Toshiba   |   |   | V |   |   |   |   |   | V |    |    |
| 5  | Panasonic |   | V | V | V |   |   |   | V |   |    |    |
| 6  | Nestle    | V | V |   |   | V |   | V |   |   |    |    |
| 7  | Johnson   |   |   |   |   |   |   |   |   |   |    |    |
| 8  | LG        |   |   |   |   |   | V |   |   |   |    |    |
| 9  | Avibras   |   |   |   |   |   |   |   |   |   |    |    |
| 10 | Embraer   |   |   |   |   |   |   |   |   |   |    |    |
| 11 | Gerda     | V |   |   |   |   |   | V | V |   | V  |    |

## Características dessa estrutura:

- É rápido saber se  $v \rightarrow w$  é um arco do grafo
- Gasta muita memória com grafos esparsos
- O preenchimento do grafo é demorado.

|    | Fábrica   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|----|-----------|---|---|---|---|---|---|---|---|---|----|----|
| 1  | GM        |   |   |   |   |   | V |   |   | V | V  |    |
| 2  | Ericsson  | V |   |   |   | V | V | V |   |   |    |    |
| 3  | Fiat      |   |   |   |   |   |   |   |   |   |    |    |
| 4  | Toshiba   |   |   | V |   |   |   |   |   | V |    |    |
| 5  | Panasonic |   | V | V | V |   |   |   | V |   |    |    |
| 6  | Nestle    | V | V |   |   | V |   | V |   |   |    |    |
| 7  | Johnson   |   |   |   |   |   |   |   |   |   |    |    |
| 8  | LG        |   |   |   |   |   | V |   |   |   |    |    |
| 9  | Avibras   |   |   |   |   |   |   |   |   |   |    |    |
| 10 | Embraer   |   |   |   |   |   |   |   |   |   |    |    |
| 11 | Gerda     | V |   |   |   |   |   | V | V |   | V  |    |

## b) Listas de adjacências



|    |           |   |   |   |   |   |    |   |    |   |
|----|-----------|---|---|---|---|---|----|---|----|---|
| 1  | GM        | → | 6 | → | 9 | → | 10 | • |    |   |
| 2  | Ericsson  | → | 1 | → | 5 | → | 6  | → | 7  | • |
| 3  | Fiat      | • |   |   |   |   |    |   |    |   |
| 4  | Toshiba   | → | 3 | → | 9 | • |    |   |    |   |
| 5  | Panasonic | → | 2 | → | 3 | → | 4  | → | 8  | • |
| 6  | Nestle    | → | 1 | → | 2 | → | 5  | → | 7  | • |
| 7  | Johnson   | • |   |   |   |   |    |   |    |   |
| 8  | LG        | → | 6 | • |   |   |    |   |    |   |
| 9  | Avibras   | • |   |   |   |   |    |   |    |   |
| 10 | Embraer   | • |   |   |   |   |    |   |    |   |
| 11 | Gerdau    | → | 1 | → | 7 | → | 8  | → | 10 | • |

# Declarações para listas de adjacências

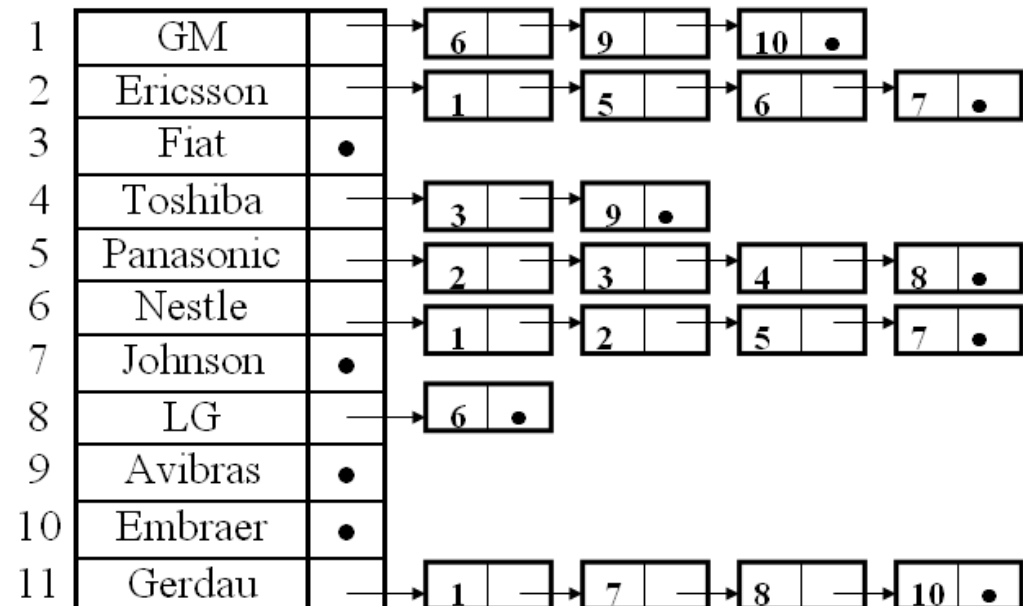
```
struct celulaadj {int vertice; celulaadj *prox;};
```

```
struct celulavertice {  
    informacaovertice InfoVert; CélulaAdj *ListAdj;};
```

```
struct digrafo {  
    celulavertice EspacoVertices [maxvert];  
    int nvert;
```

```
};
```

```
digrafo Fornecimento;
```



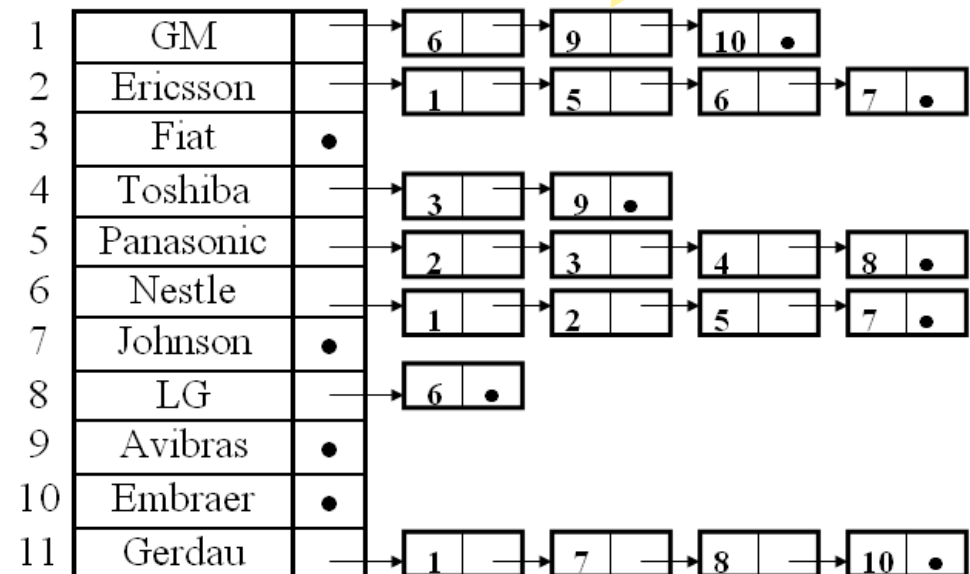
## Se houver informações nos arcos:

```
struct celulaadj {int vertice; informacaoarco InfoArco;  
    celulaadj *prox;};
```

```
struct celulavertice {  
    informacaovertice InfoVert; CélulaAdj *ListAdj;};
```

```
struct digrafo {  
    celulavertice Vertices [maxvert];  
    int nvert;  
};
```

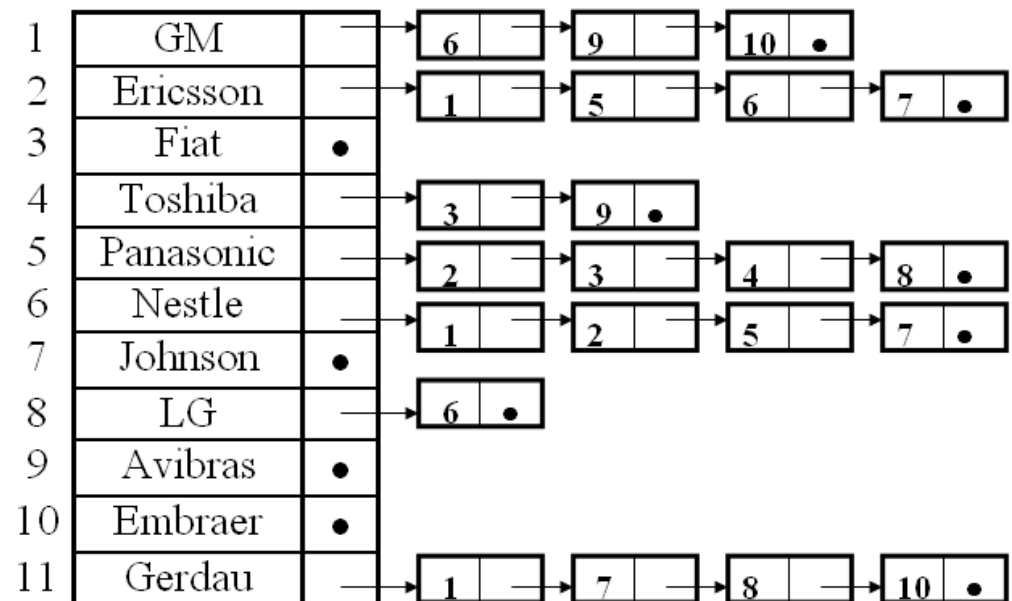
```
digrafo Fornecimento;
```





## Características dessa estrutura:

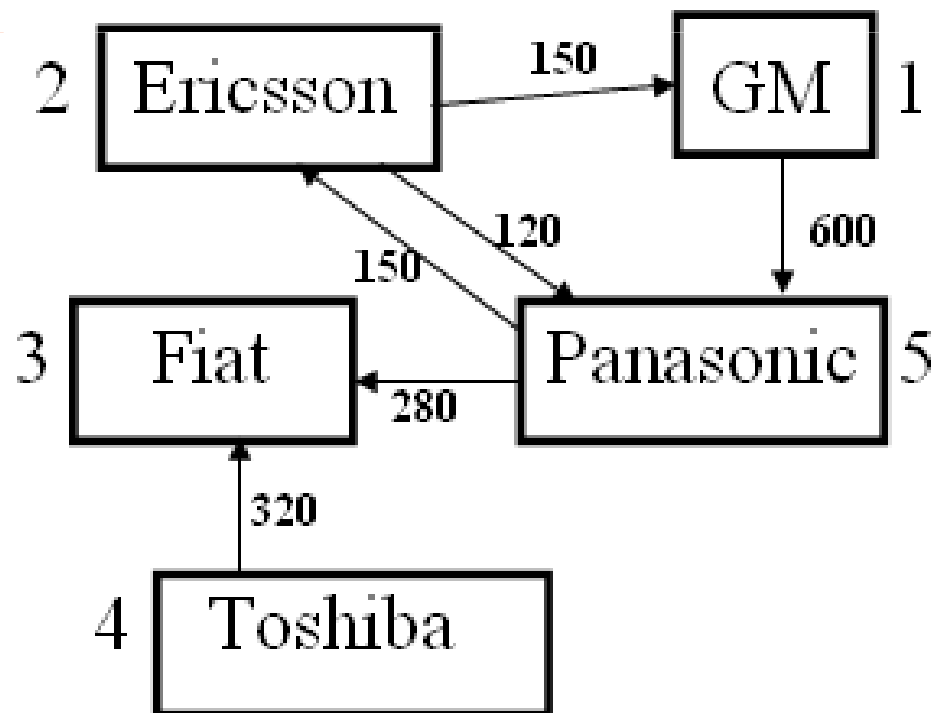
- Demora um pouco mais para saber se  $v \rightarrow w$  é um arco do grafo
- Gasta menos memória para grafos esparsos
- A programação para preencher o grafo é mais complexa



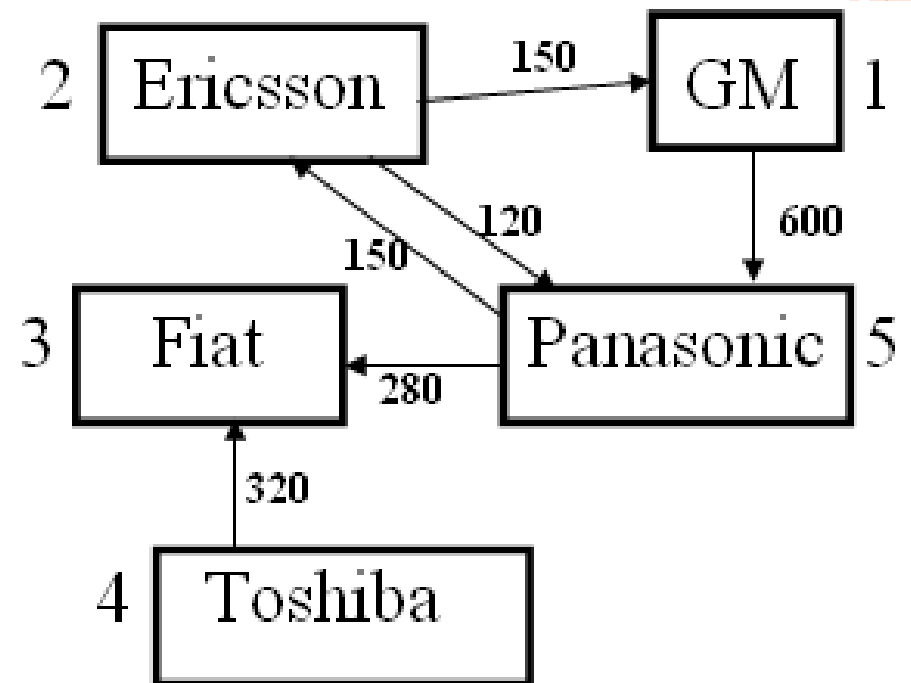
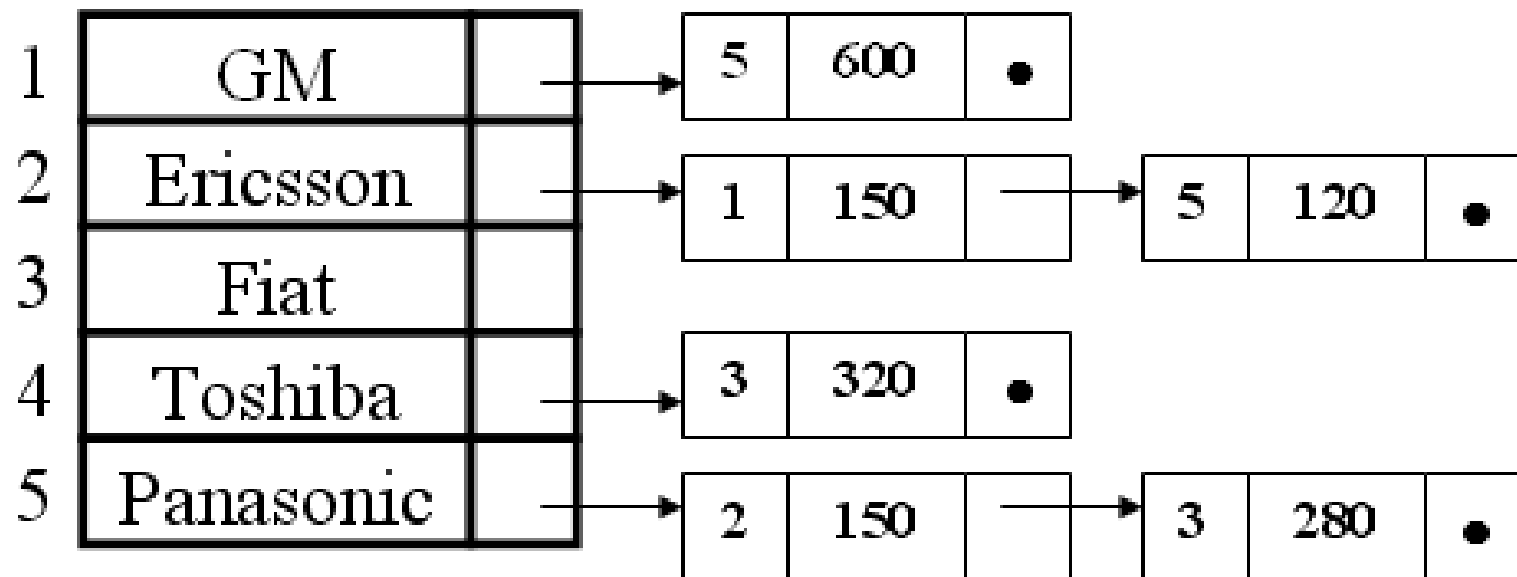
## 2.3 – Armazenamento de um digrafo

- Seja um digrafo de fornecimento de produtos entre fábricas

Nos arcos: custo do fornecimento/mês em R\$1000,00



Adotada a estrutura com listas de adjacências



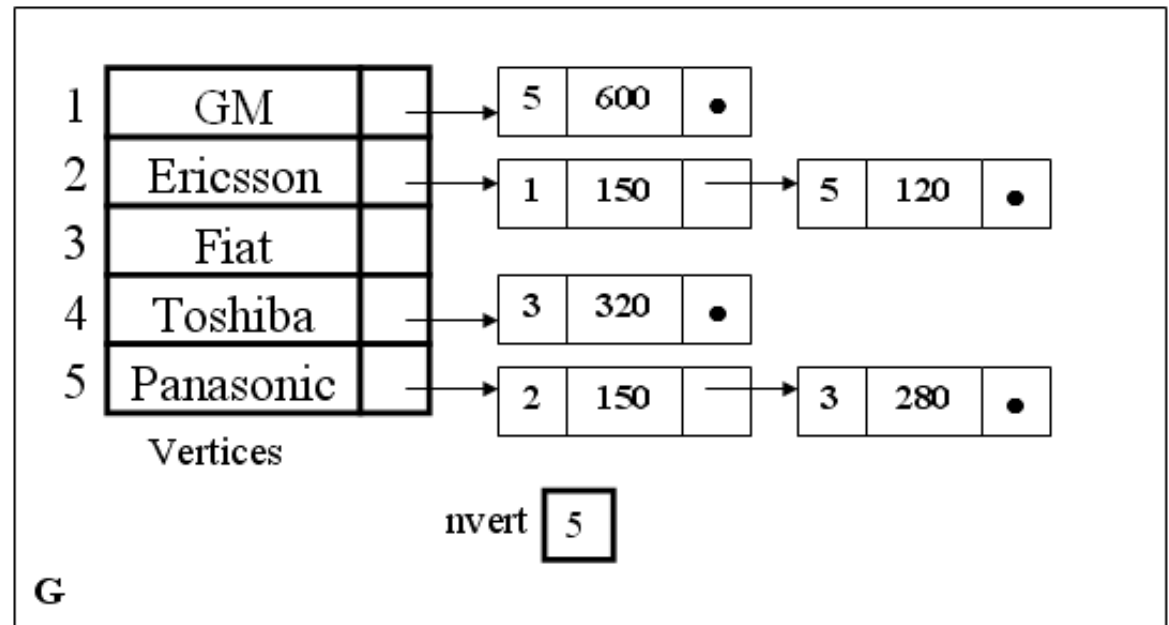
## Estrutura de dados:

```
struct celulaadj {
    int vertice, custo;
    celulaadj *prox;
};

struct celulavertice {
    char nome[20];
    celulaadj *listadj;
};

struct digrafo {
    int nvert;
    celulavertice *Vertices;
};

digrafo G;
```



```

/*          Funcao LerGrafo: le e armazena um digrafo na
variavel global G          */

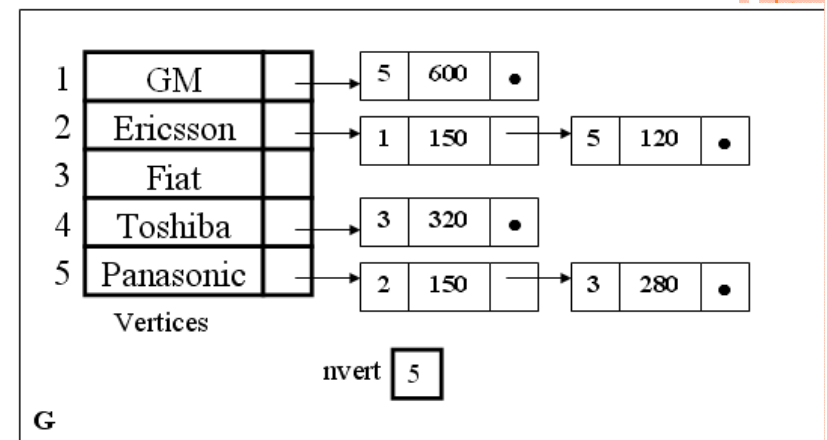
void LerGrafo () {
    int i, j, n;
    celulaadj *p;

    /*  Alocacao do vetor de vertices  */

    printf("Leitura do Grafo:");
    printf("Numero de fabricas:");
    scanf("%d", &G.nvert);
    G.Vertices = malloc ((G.nvert+1) *
        sizeof(celulavertice));
}

```

Não usa o elemento índice zero

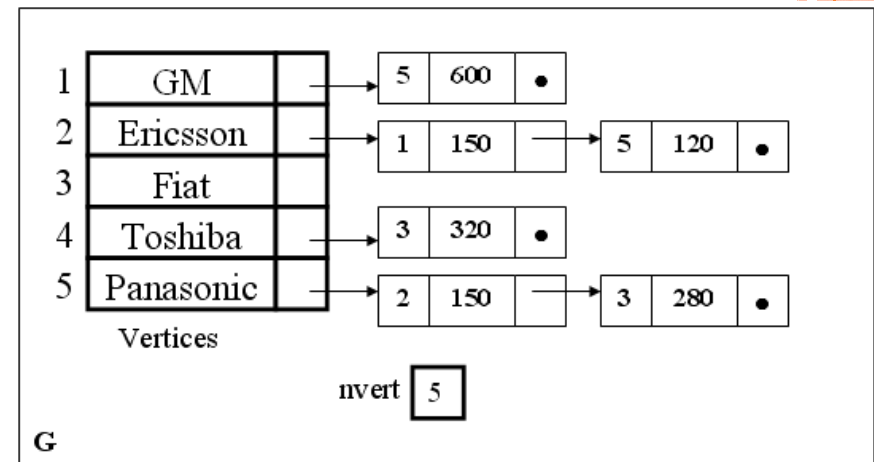


```

/*  Leitura do nome das fabricas  */

for (i = 1; i <= G.nvert; i++) {
    printf("Nome da fabrica", i, ": ");
    scanf("%s", G.Vertices[i].nome);
    G.Vertices[i].listadj = NULL;
}

```



```

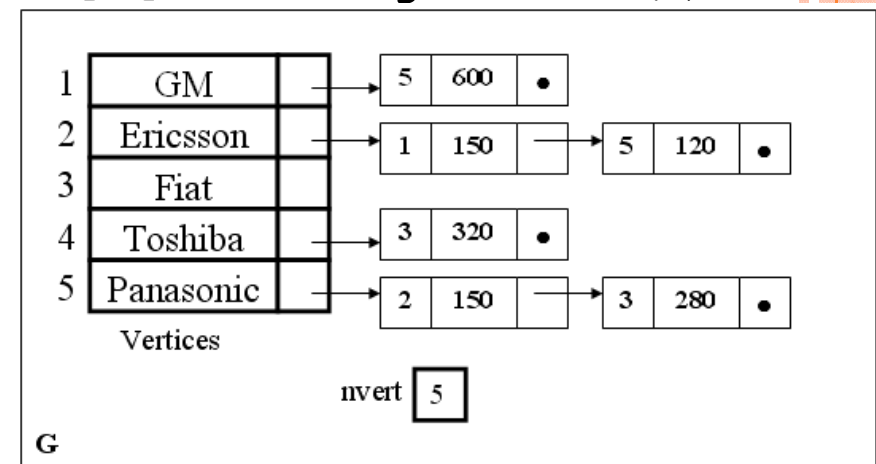
/*  Leitura dos fornecimentos e seus custos  */

for (i = 1; i <= G.nvert; i++) {
    printf("Numero de fornecimentos de %s:",
           G.Vertices[i].nome);

    scanf("%d", &n);
    if (n > 0) {
        for (j = 1; j <= n; j++) {
            printf ("%d.o) n.o da fabrica e custo: ", j);
            p = G.Vertices[i].listadj;
            G.Vertices[i].listadj =
                malloc (sizeof (celulaadj));
            G.Vertices[i].listadj->prox = p;
            scanf ("%d", &G.Vertices[i].listadj->vertice);
            scanf ("%d", &G.Vertices[i].listadj->custo);
        }
    }
}
}

```

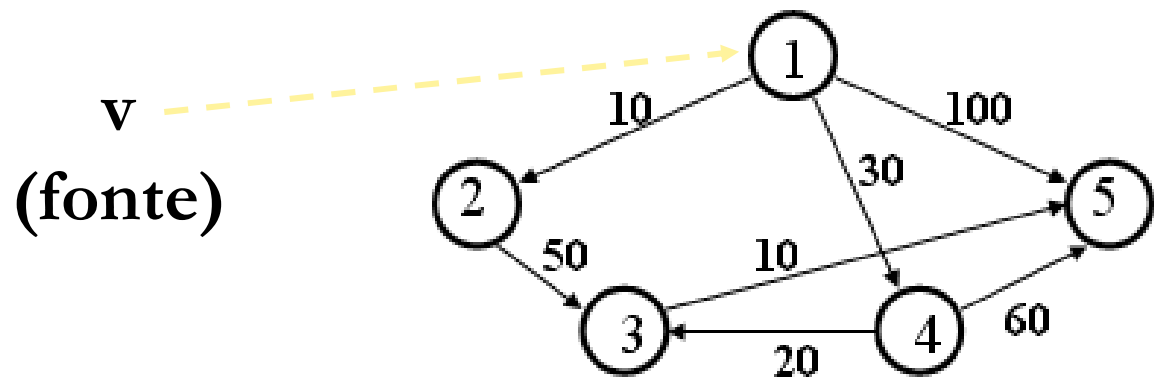
As células de adjacências são inseridas no início de cada lista



# 3 – PROBLEMAS CLÁSSICOS SOBRE GRAFOS ORIENTADOS

## 3.1 – Caminhos de menores custos

- Seja um digrafo  $G = \{V, A\}$  em que cada arco de  $A$  possui uma informação inteira positiva chamada **custo**
- Seja  $v$  um vértice de  $V$  que recebe o nome de **fonte**





- Determinar o custo do caminho **mais barato** da fonte para todos os outros vértices de  $V$
- **Custo de um caminho:** soma dos custos dos arcos desse caminho

Caminhos de 1 a 3:

1-2-3 (custo 60)

**1-4-3 (custo 50)**

Caminhos de 1 a 5:

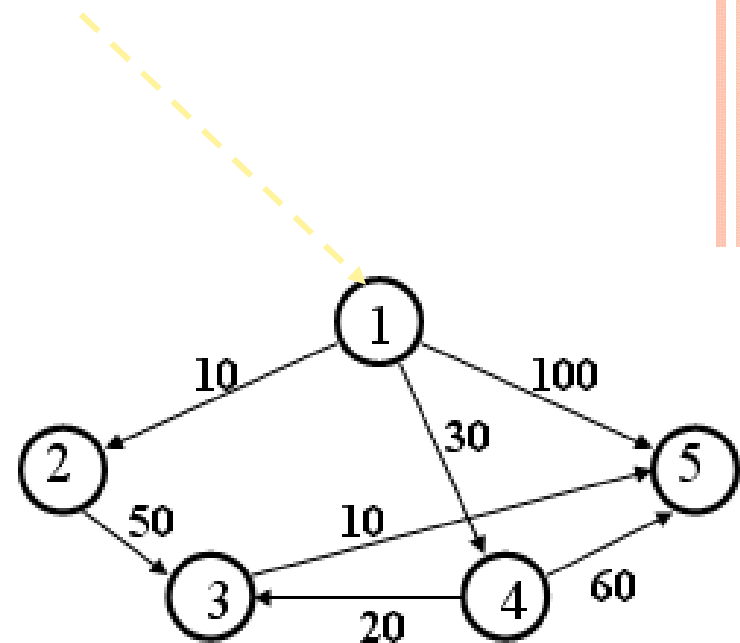
1-5 (custo 100)

1-2-3-5 (custo 70)

1-4-5 (custo 90)

**1-4-3-5 (custo 60)**

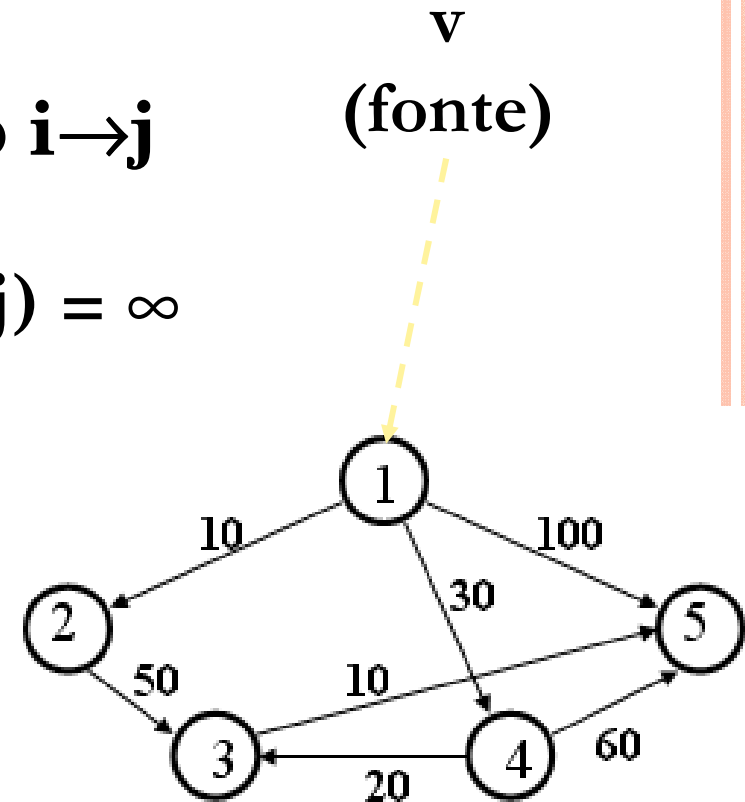
$v$   
(fonte)



## Solução: Algoritmo de Dijkstra

- O grafo é  $G = (V, A)$
- O conjunto de vértices é  $V = \{1, 2, \dots, n\}$
- A fonte é o vértice 1
- $C(i, j)$ : custo associado ao arco  $i \rightarrow j$
- Se o arco  $i \rightarrow j$  não existe,  $C(i, j) = \infty$

Esse algoritmo e o conceito de custos podem ser aplicados em grafos não-orientados

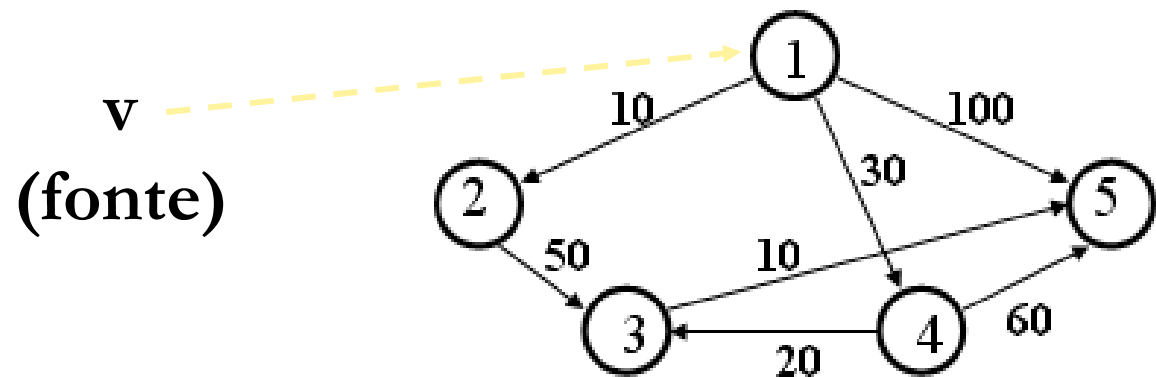


**Vetor D:** destinado a guardar os caminhos mais baratos e seus custos; no início  $D[i]$  guarda:

Caminho  $[1-i]$  e  $C(1, i)$

**Vetor D**

|                   | D[2] | D[3]     | D[4] | D[5] |
|-------------------|------|----------|------|------|
| <b>Caminho</b>    | 1-2  | 1-3      | 1-4  | 1-5  |
| <b>Custo</b>      | 10   | $\infty$ | 30   | 100  |
| <b>Definitivo</b> | não  | não      | não  | não  |



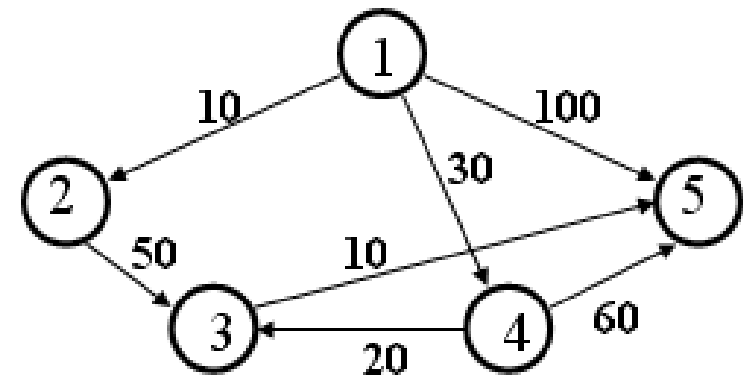
# Simulação:

## Vetor D

|            | D[2] | D[3]     | D[4] | D[5] |
|------------|------|----------|------|------|
| Caminho    | 1-2  | 1-3      | 1-4  | 1-5  |
| Custo      | 10   | $\infty$ | 30   | 100  |
| Definitivo | sim  | não      | não  | não  |

Menor custo: 10

D[2] é definitivo



## Vetor D

|            | D[2] | D[3]     | D[4] | D[5] |
|------------|------|----------|------|------|
| Caminho    | 1-2  | 1-3      | 1-4  | 1-5  |
| Custo      | 10   | $\infty$ | 30   | 100  |
| Definitivo | sim  | não      | não  | não  |

Novos caminhos:

$$\text{Custo (1-3)} = \infty$$

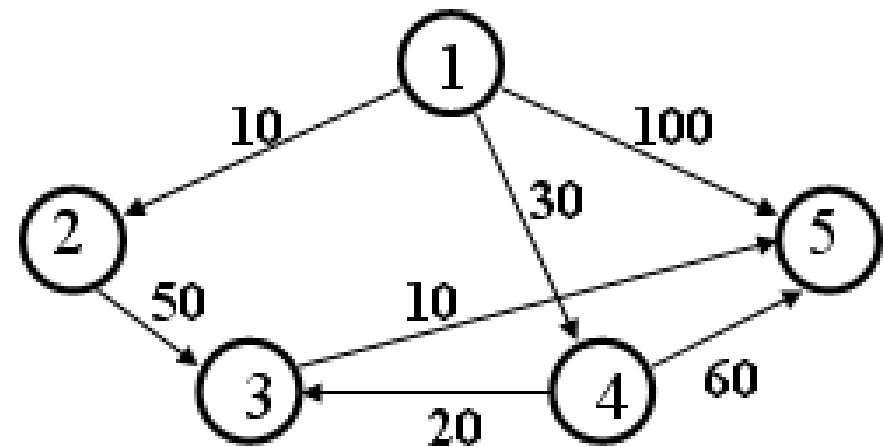
$$\begin{aligned}\text{Custo (1-2-3)} &= \text{Custo (1-2)} + C(2, 3) \\ &= 10 + 50 = 60 \text{ (menor)}\end{aligned}$$

$$\text{Custo (1, 4)} = 30 \text{ (menor)}$$

$$\begin{aligned}\text{Custo (1-2-4)} &= \text{Custo (1-2)} + C(2, 4) \\ &= 10 + \infty = \infty\end{aligned}$$

$$\text{Custo (1, 5)} = 100 \text{ (menor)}$$

$$\begin{aligned}\text{Custo (1-2-5)} &= \text{Custo (1-2)} + C(2, 5) \\ &= 10 + \infty = \infty\end{aligned}$$



## Vetor D

|                   | <b>D[2]</b> | D[3]     | D[4] | D[5] |
|-------------------|-------------|----------|------|------|
| <b>Caminho</b>    | <b>1-2</b>  | 1-3      | 1-4  | 1-5  |
| <b>Custo</b>      | <b>10</b>   | $\infty$ | 30   | 100  |
| <b>Definitivo</b> | <b>sim</b>  | não      | não  | não  |

Menor entre os não definitivos:

Caminho de 1 para 4. onde

$$\text{Custo}(1-2-4) > \text{Custo}(1-4)$$

Outros caminhos possíveis, onde C é um caminho qualquer:

$$\text{Custo}(1-2-C-4) = \text{Custo}(1,2) + \text{Custo}(2-C-4) \geq \text{Custo}(1,2,4) > \text{Custo}(1,4)$$

$$\text{Custo}(1-3-C-4) = \text{Custo}(1,3) + \text{Custo}(3-C-4) > \text{Custo}(1,4)$$

$$\text{Custo}(1-5-C-4) = \text{Custo}(1,5) + \text{Custo}(5-C-4) > \text{Custo}(1,4)$$

Logo o caminho de menor custo possível entre 1 e 4 é: 1-4

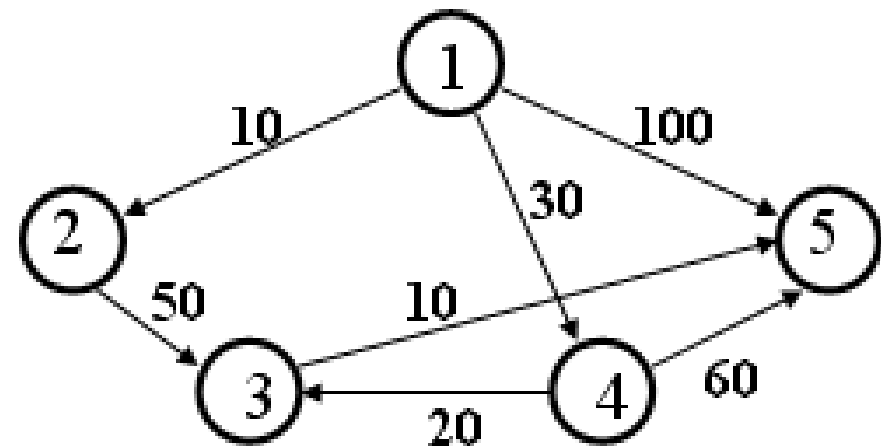


## Vetor D

|                   | <b>D[2]</b> | D[3]  | <b>D[4]</b> | D[5] |
|-------------------|-------------|-------|-------------|------|
| <b>Caminho</b>    | <b>1-2</b>  | 1-2-3 | <b>1-4</b>  | 1-5  |
| <b>Custo</b>      | <b>10</b>   | 60    | <b>30</b>   | 100  |
| <b>Definitivo</b> | <b>sim</b>  | não   | <b>sim</b>  | não  |

Menor custo: 30

D[4] é definitivo



## Vetor D

|                   | <b>D[2]</b> | D[3]  | <b>D[4]</b> | D[5] |
|-------------------|-------------|-------|-------------|------|
| <b>Caminho</b>    | <b>1-2</b>  | 1-2-3 | <b>1-4</b>  | 1-5  |
| <b>Custo</b>      | <b>10</b>   | 60    | <b>30</b>   | 100  |
| <b>Definitivo</b> | <b>sim</b>  | não   | <b>sim</b>  | não  |

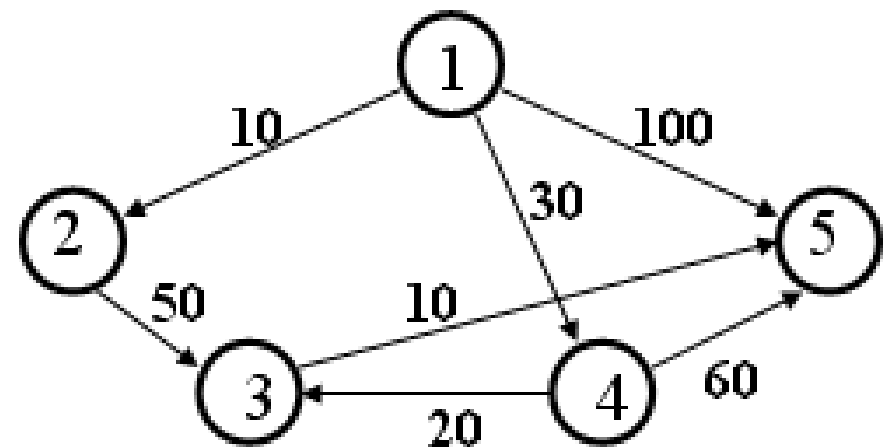
Novos caminhos:

$$\text{Custo (1-2-3)} = 60$$

$$\begin{aligned}\text{Custo (1-4-3)} &= \text{Custo (1-4)} + C(4, 3) \\ &= 30 + 20 = 50 \text{ (menor)}\end{aligned}$$

$$\text{Custo (1, 5)} = 100$$

$$\begin{aligned}\text{Custo (1-4-5)} &= \text{Custo (1-4)} + C(4, 5) \\ &= 30 + 60 = 90 \text{ (menor)}\end{aligned}$$



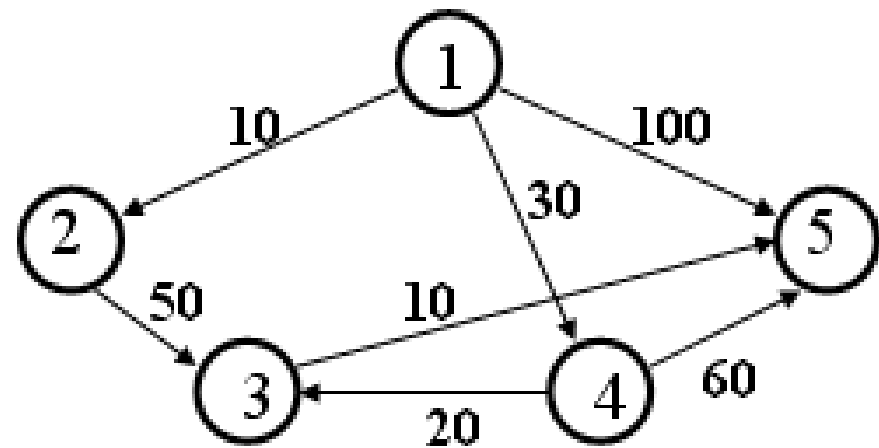


## Vetor D

|            | D[2] | D[3]  | D[4] | D[5]  |
|------------|------|-------|------|-------|
| Caminho    | 1-2  | 1-4-3 | 1-4  | 1-4-5 |
| Custo      | 10   | 50    | 30   | 90    |
| Definitivo | sim  | sim   | sim  | não   |

Menor custo: 50

D[3] é definitivo



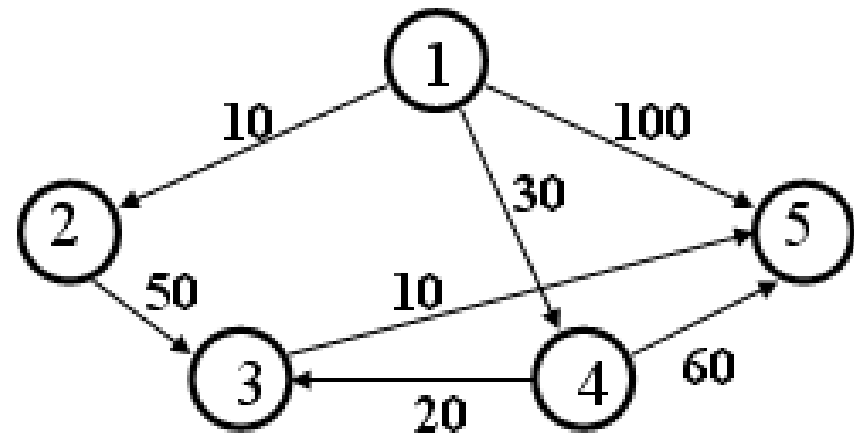
## Vetor D

|            | D[2] | D[3]  | D[4] | D[5]  |
|------------|------|-------|------|-------|
| Caminho    | 1-2  | 1-4-3 | 1-4  | 1-4-5 |
| Custo      | 10   | 50    | 30   | 90    |
| Definitivo | sim  | sim   | sim  | não   |

Novos caminhos:

Custo (1-4-5) = 90

Custo (1-4-3-5) = Custo (1-4-3) + C (3, 5)  
= 50 + 10 = 60 (menor)

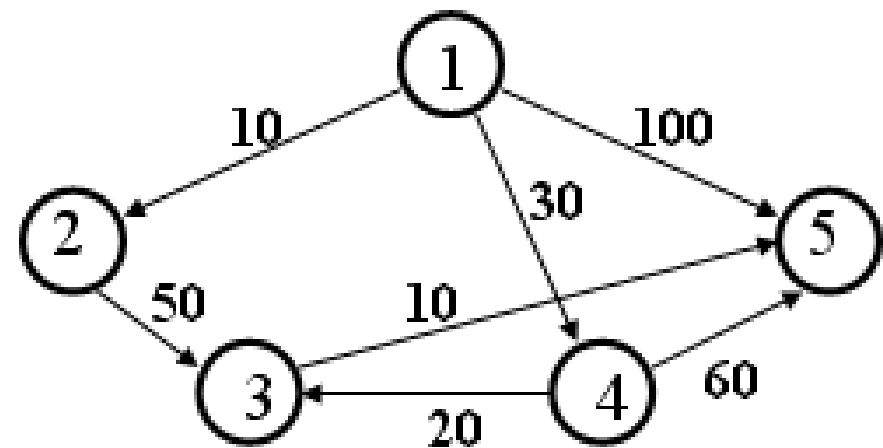


## Vetor D

|            | D[2] | D[3]  | D[4] | D[5]    |
|------------|------|-------|------|---------|
| Caminho    | 1-2  | 1-4-3 | 1-4  | 1-4-3-5 |
| Custo      | 10   | 50    | 30   | 60      |
| Definitivo | sim  | sim   | sim  | sim     |

Menor custo: 60

D[5] é definitivo



## Vetor D

|            | D[2] | D[3]  | D[4] | D[5]    |
|------------|------|-------|------|---------|
| Caminho    | 1-2  | 1-4-3 | 1-4  | 1-4-3-5 |
| Custo      | 10   | 50    | 30   | 60      |
| Definitivo | Sim  | sim   | sim  | sim     |

### Declarações para o vetor D:

```
struct infovert {  
    int custo;  
    char definitivo;  
    caminho cam;  
};  
  
infovert *D;
```

### Caminho é lista de vértices:

```
typedef noh *posicao;  
struct noh {  
    vertice elem;  
    posicao prox;  
};  
struct caminho {  
    posicao inic, fim;  
};
```

## Vetor D

|            | D[2] | D[3]  | D[4] | D[5]    |
|------------|------|-------|------|---------|
| Caminho    | 1-2  | 1-4-3 | 1-4  | 1-4-3-5 |
| Custo      | 10   | 50    | 30   | 60      |
| Definitivo | Sim  | sim   | sim  | sim     |

Caminho é lista de vértices:

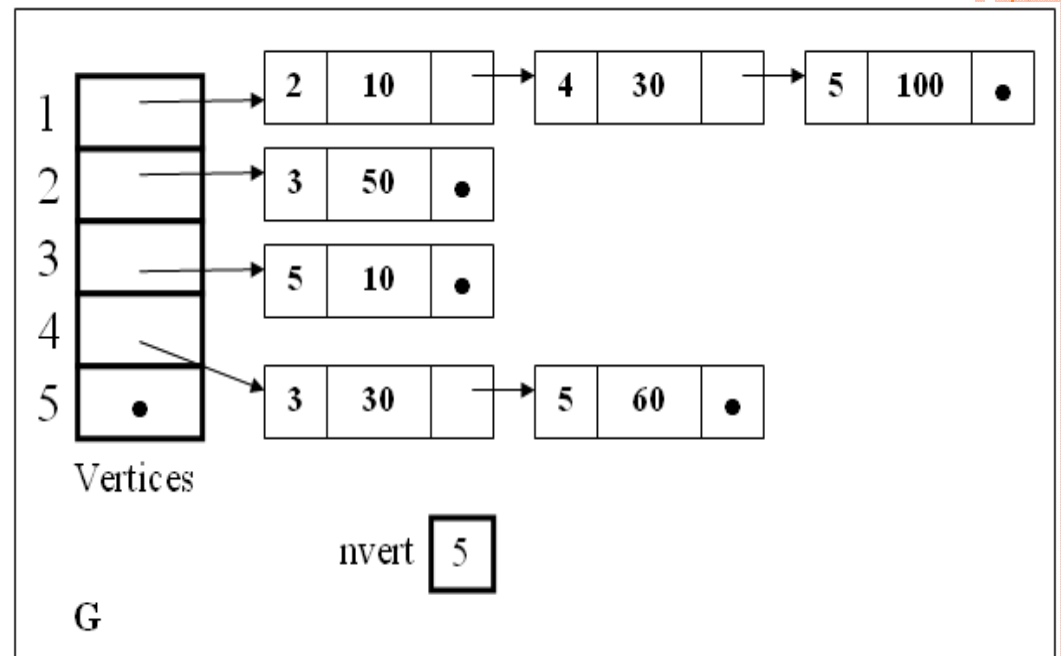
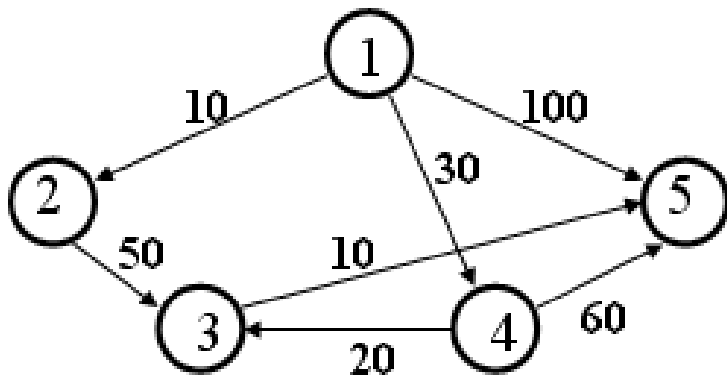
```
typedef noh *posicao;
struct noh {
    vertice elem;
    posicao prox;
};
struct caminho {
    posicao inic, fim;
};
```

Caminho – operadores:

```
void InicCaminho (caminho *cam);
posicao Primeira (caminho cam);
posicao Proximo (posicao p);
posicao Fim (caminho cam);
void InserirNoFinal (int v, caminho *c)
int CustoArco(int, int, digrafo);
void EscreverCaminho (caminho);
void CopiarCaminho (caminho *destino,
                    caminho origem);
void EsvaziarCaminho (caminho *);
```

# Algoritmo de Dijkstra:

```
void Dijkstra (Grafo *G) {  
    infovert *D; int i, novocusto;  
    celulaadj *p; vertice w, v;  
    Inicializar vetor D;  
    Calcular vetor D definitivo;  
    Escrever vetor D;  
}
```



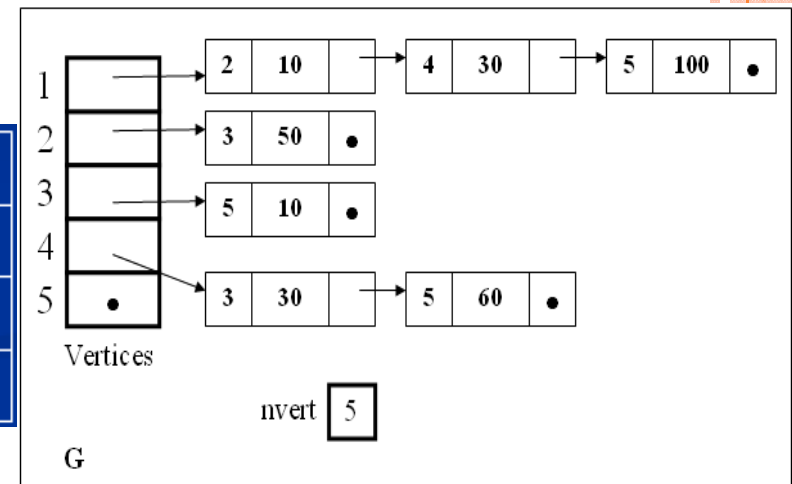
## Inicializar vetor D:

```

D = malloc ((G->nvert + 1) * sizeof(infovert));
for (i = 2; i <= G->nvert; i++) {
    D[i].custo = infinito;
    D[i].definitivo = FALSE;
    InicCaminho (&D[i].cam);
    InserirNoFinal (1, &D[i].cam);
    InserirNoFinal (i, &D[i].cam);
}
for (p = G->Vertices[1]; p != NULL; p = p->prox) {
    D[p->vert].custo = p->custo;
}

```

|            | D[2] | D[3]     | D[4] | D[5] |
|------------|------|----------|------|------|
| Caminho    | 1-2  | 1-3      | 1-4  | 1-5  |
| Custo      | 10   | $\infty$ | 30   | 100  |
| Definitivo | não  | não      | não  | não  |

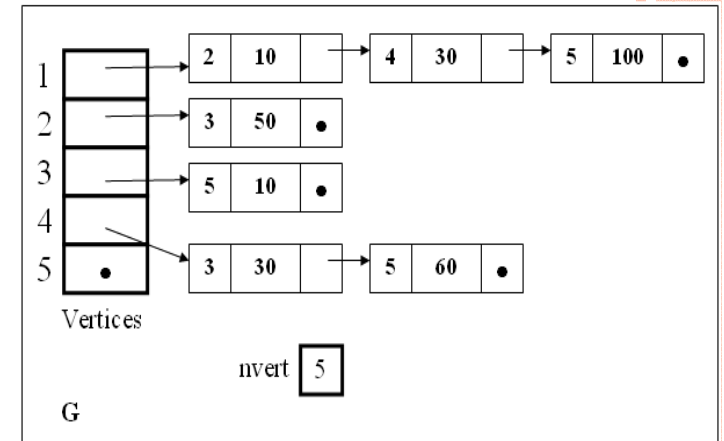


## Calcular vetor D definitivo:

```

for (i = 2; i <= G->nvert; i++) {
    w = Minimo (D, G->nvert);
    D[w].definitivo = TRUE;
    for (v = 2; v <= G->nvert; v++)
        if (D[v].definitivo == FALSE) {
            novocusto = D[w].custo +
                CustoArco (w, v, G);
            if (novocusto < D[v].custo) {
                D[v].custo = novocusto;
                CopiarCaminho
                    (&D[v].cam, D[w].cam);
                InserirNoFinal (v, &D[v].cam);
            }
        }
    }
}

```

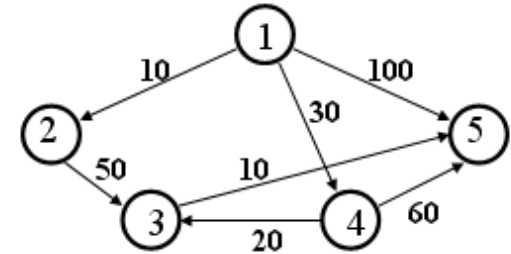


|            | D[2] | D[3] | D[4] | D[5] |
|------------|------|------|------|------|
| Caminho    | 1-2  | 1-3  | 1-4  | 1-5  |
| Custo      | 10   | ∞    | 30   | 100  |
| Definitivo | não  | não  | não  | não  |



## Escrever vetor D:

```
for (i = 2; i <= G->nvert; i++) {
    printf("Caminho (1, %d): ", i);
    printf("Custo = %d Vertices:", D[i].custo);
    EscreverCaminho (D[i].cam);
}
```



|            | D[2] | D[3]  | D[4] | D[5]    |
|------------|------|-------|------|---------|
| Caminho    | 1-2  | 1-4-3 | 1-4  | 1-4-3-5 |
| Custo      | 10   | 50    | 30   | 60      |
| Definitivo | sim  | sim   | sim  | sim     |

Saída para o grafo acima:

Caminhos mais baratos ao vertice 1:

|                 |             |           |   |   |   |   |
|-----------------|-------------|-----------|---|---|---|---|
| Caminho (1, 2): | Custo = 10; | Vertices: | 1 | 2 |   |   |
| Caminho (1, 3): | Custo = 50; | Vertices: | 1 | 4 | 3 |   |
| Caminho (1, 4): | Custo = 30; | Vertices: | 1 | 4 |   |   |
| Caminho (1, 5): | Custo = 60; | Vertices: | 1 | 4 | 3 | 5 |

## Aplicações de caminhos de menores custos:

- Escolha da melhor rota para transitar numa malha rodoviária ou para viagens aéreas
  - Aparelhos GPS, sites de informações sobre mapas, etc.



## 3.2 – Busca em profundidade

### a) Conceito e metodologia

- Busca em profundidade é um método de **travessia de grafos** (orientados ou não)
- É esqueleto para a solução de muitos problemas relacionados com grafos (orientados ou não)



- **Travessia de grafos:** visitaç o sistem tica de todos os n s de um grafo, usando os arcos para transitar

- **Busca em profundidade:** generaliza o de pr , p s e ordem central para  rvores

- **Busca em largura:** generaliza o da ordem de n vel para  rvores (ser  apresentada para os grafos n o-orientados)

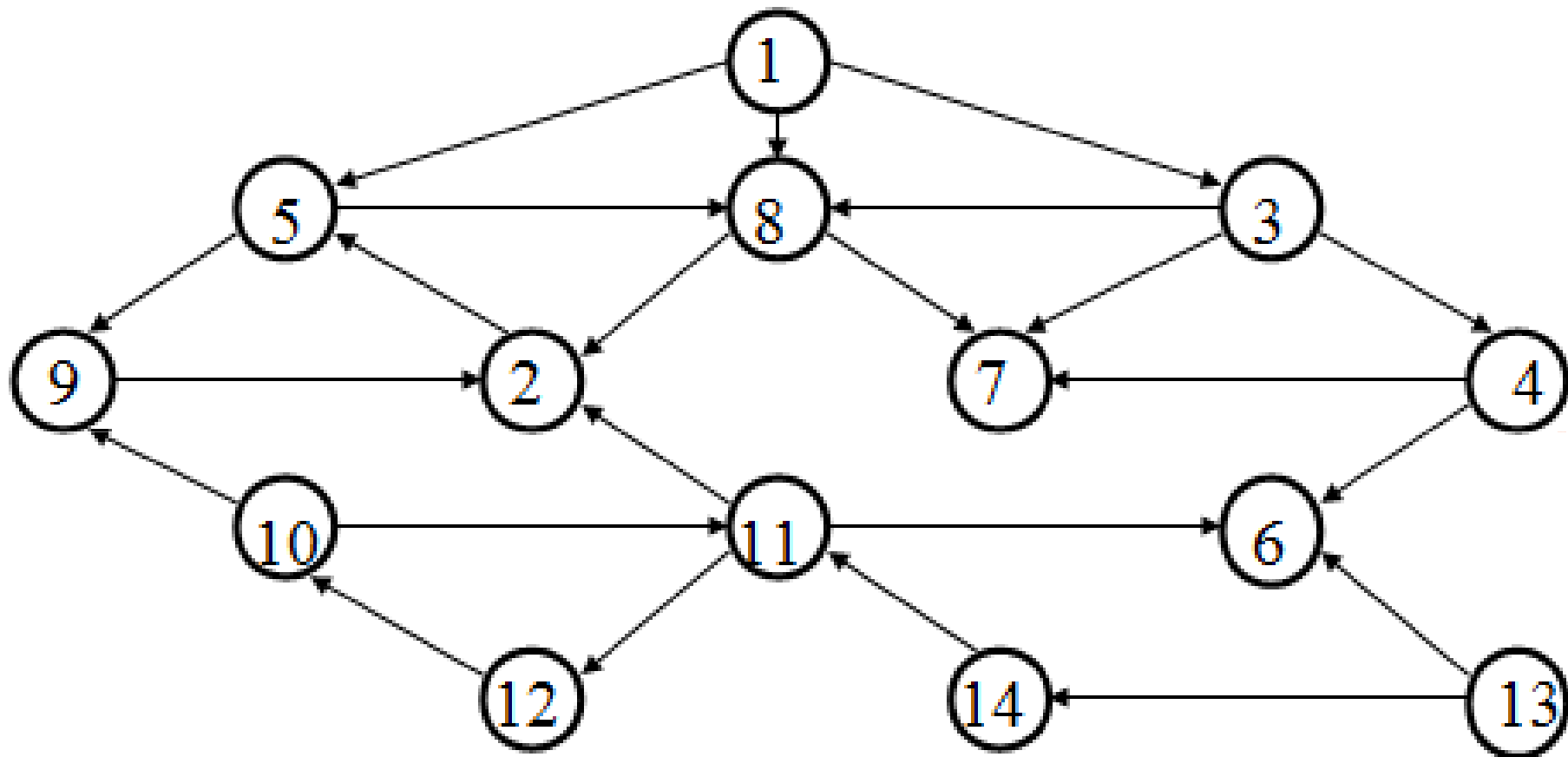


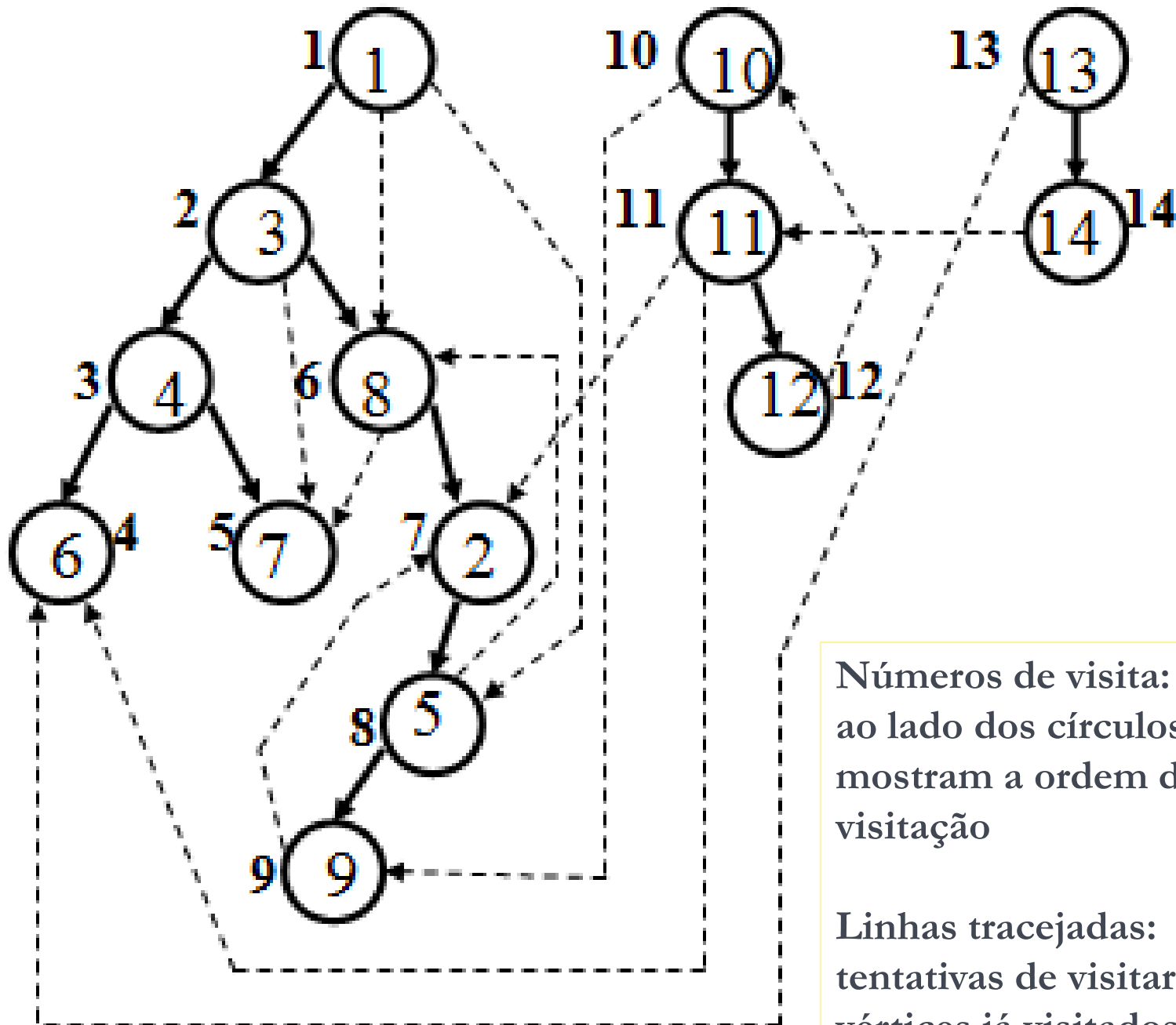
**Método para digrafos:** Seja  $G$  um digrafo

1. Assinalar todos os vértices de  $G$  como **não-visitados**
2. Seja  $v$  um vértice de  $G$  selecionado como vértice inicial; Marcar  $v$  como **visitado**
3. Visitar cada vértice  $w$  adjacente a  $v$ , não visitado
4. Ao visitar um vértice  $w$ , marcá-lo como **visitado** e visitar cada vértice não visitado, adjacente a  $w$  (recursividade)
5. A busca em  $v$  é encerrada quando todos os vértices  $w$  adjacentes a  $v$  tiverem sido visitados
6. Se alguns vértices permanecerem não visitados, selecionar um deles como inicial, e aplicar-lhe os passos de 2 a 6



**Exemplo:** seja o digrafo:

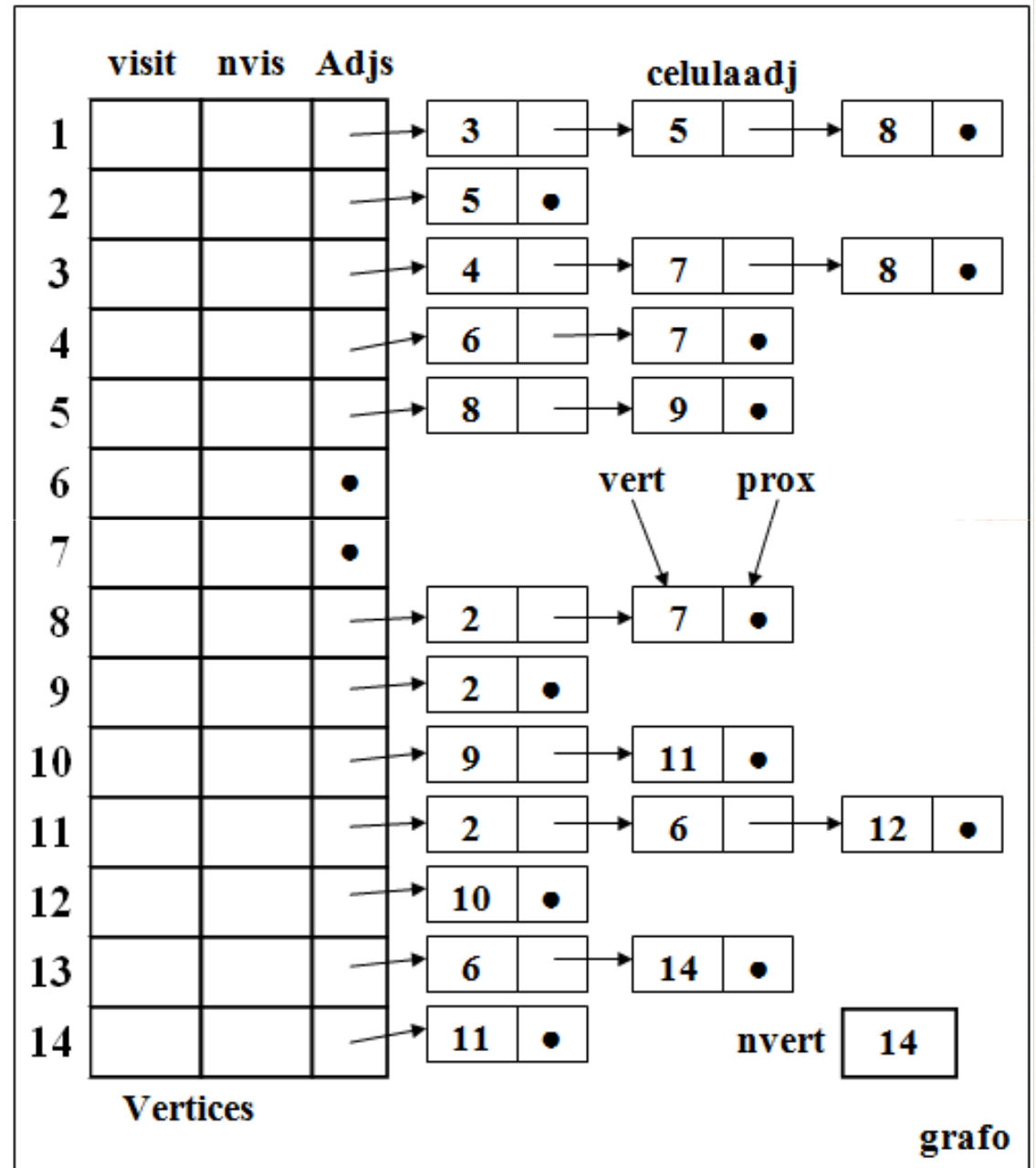
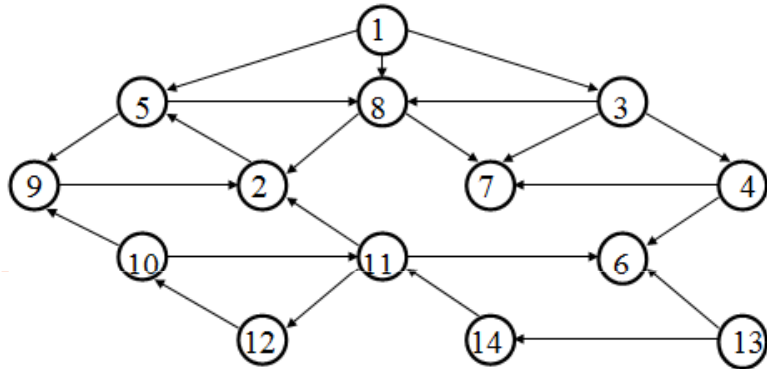




Números de visita:  
ao lado dos círculos,  
mostram a ordem de  
visitação

Linhas tracejadas:  
tentativas de visitar  
vértices já visitados

## b) Programação e estrutura de dados (listas de adjacências)

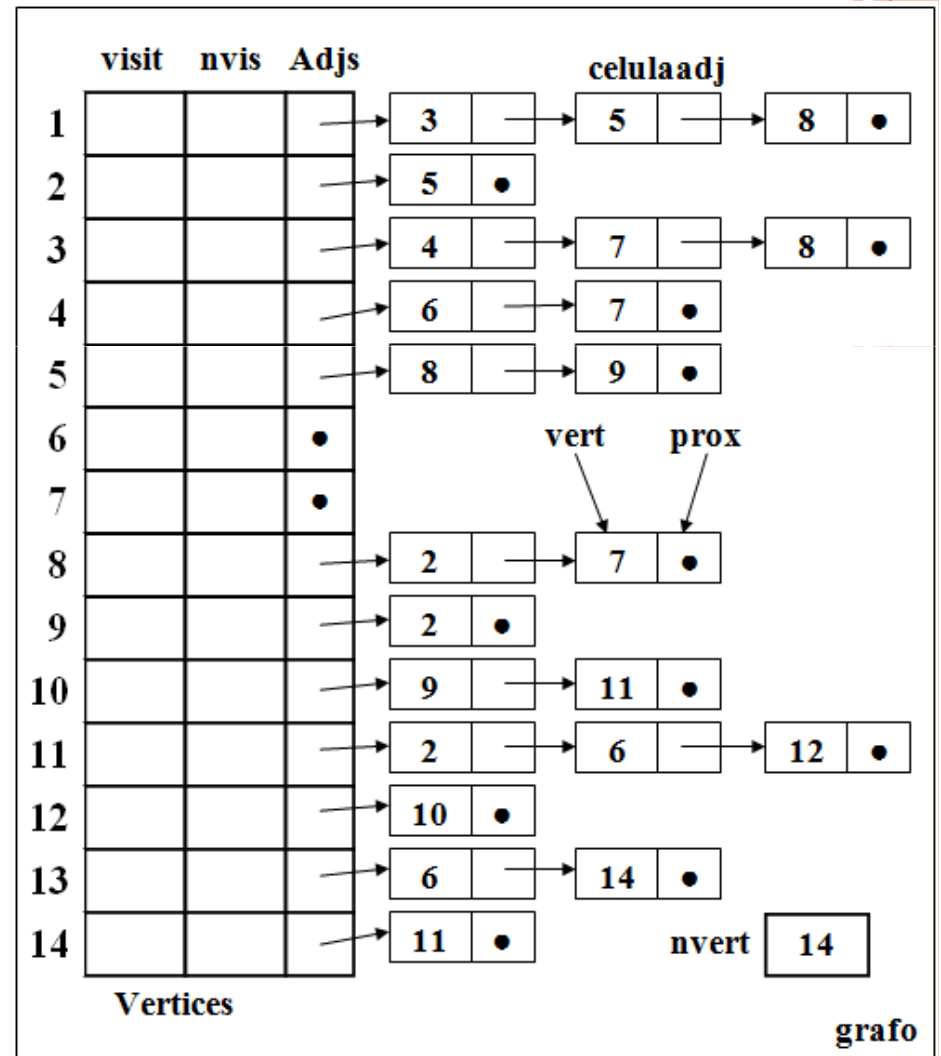




```

struct celulaadj {
    vertice vert; celulaadj *prox;
};
typedef celulaadj *listadj;
struct celulavert {
    logic visit; int nvis;
    listadj adjs;
};
typedef int vertice;
struct Grafo {
    int nvert;
    celulavert *Vertices;
};
Grafo G; int cont;

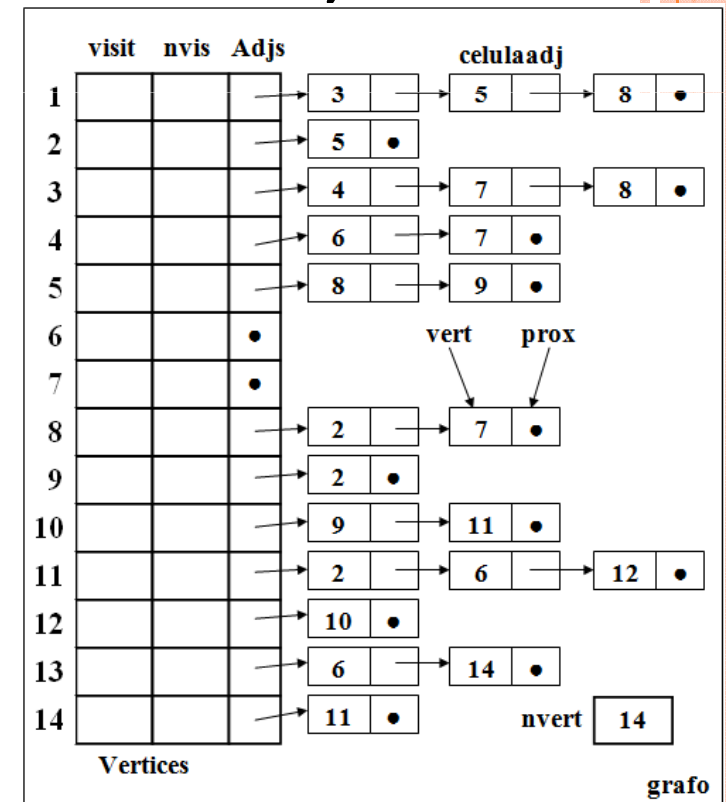
```



```

void Travessia (Grafo *G)  {
    vertice v;
    for (v = 1; v <= G->nvert; v++)
        G->Vertices[v].visit = FALSE;
    cont = 0;
    for (v = 1; v <= G->nvert; v++)
        if (G->Vertices[v].visit == FALSE)
            BuscaProf (v, G);
}

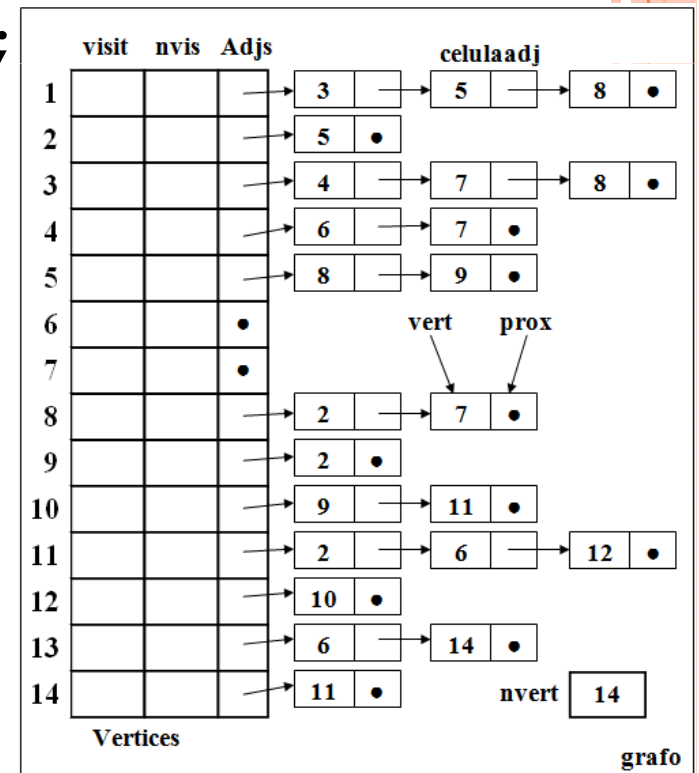
```



```

void BuscaProf (vertice v, Grafo *G) {
    celulaadj *p;
    G->Vertices[v].visit = TRUE;    cont++;
    G->Vertices[v].nvis = cont;
    p = G->Vertices[v].adjs;
    while (p != NULL) {
        if (G->Vertices[p->vert].visit == FALSE)
            BuscaProf (p->vert, G);
        p = p->prox;
    }
}

```

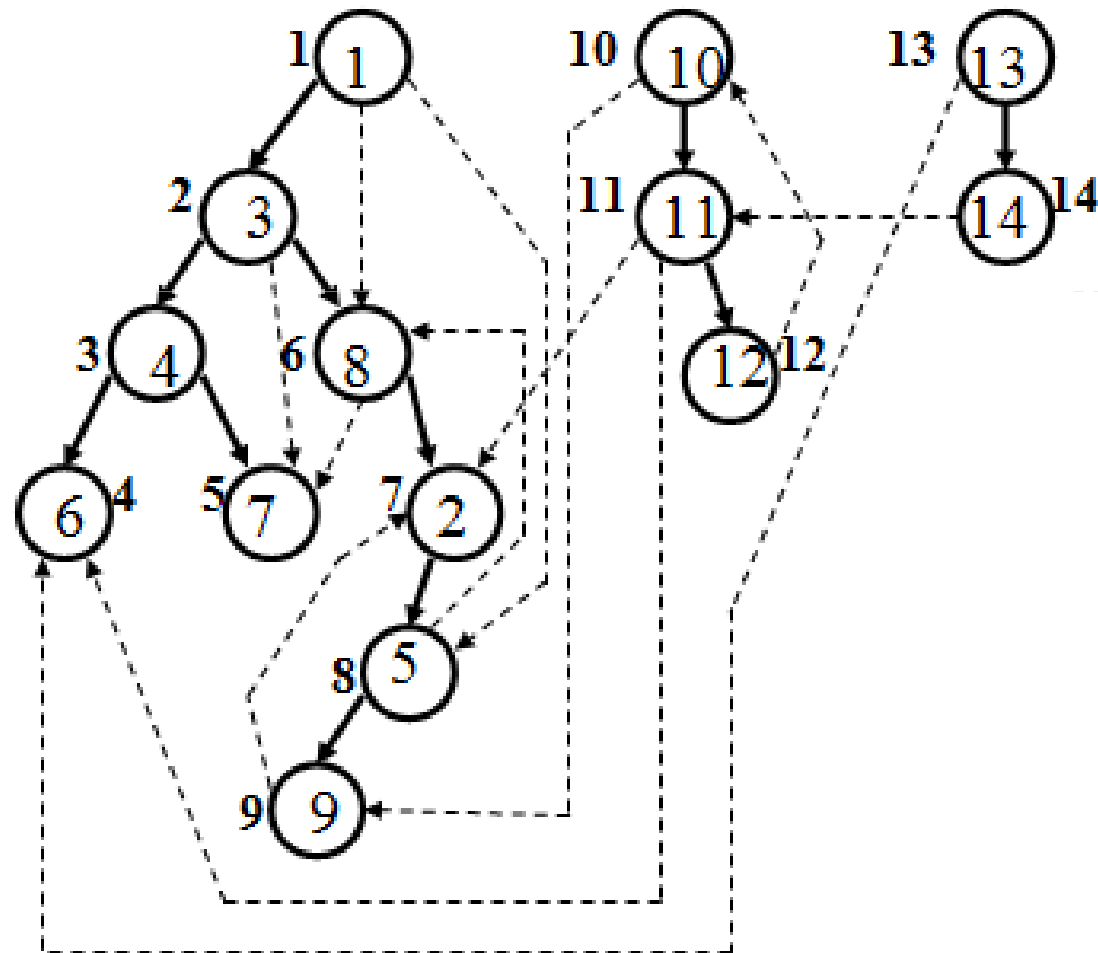


## c) Floresta da busca em profundidade

Os arcos com **linha cheia** levam a vértices não-visitados

São chamados **arcos de árvore**

Formam a **floresta da busca em profundidade**



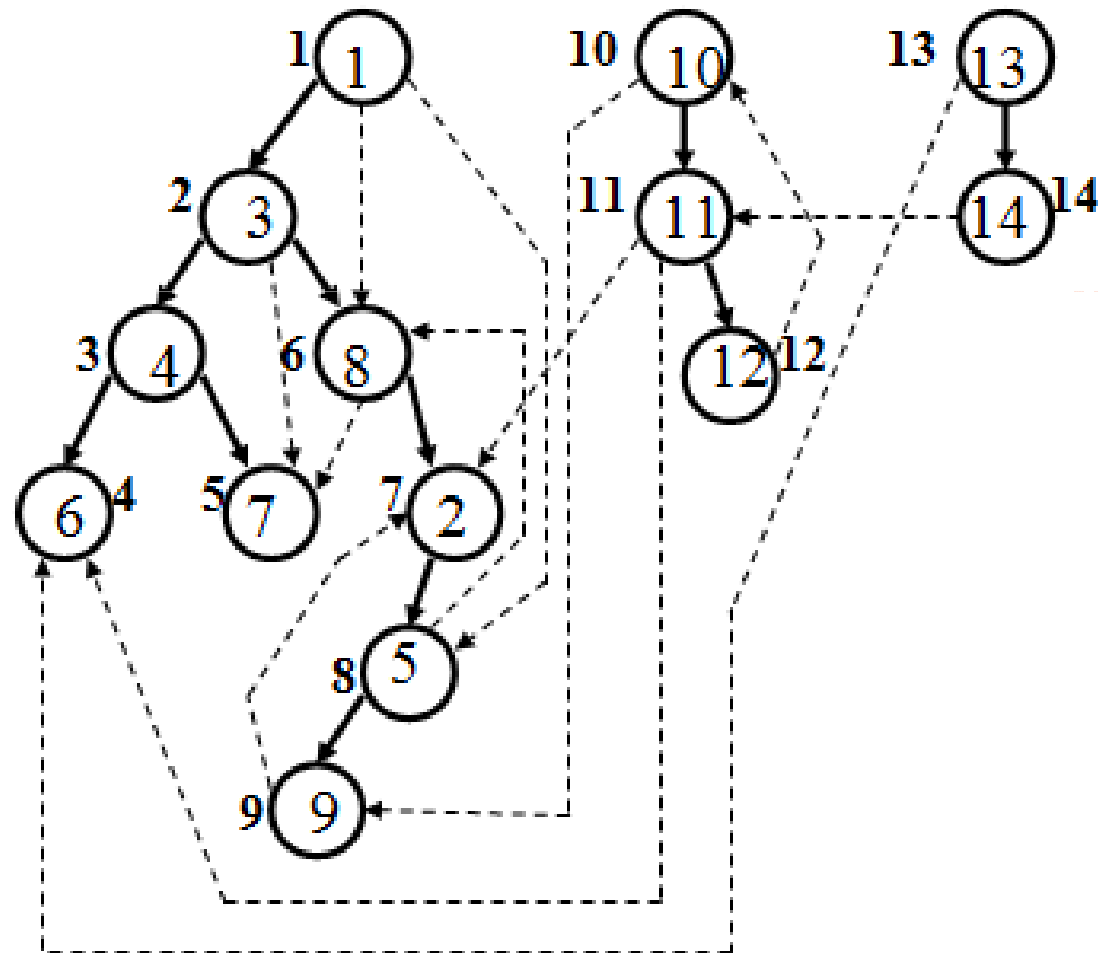
## Outros tipos de arcos (linha tracejada):

Arcos de volta, arcos para frente e arcos cruzantes

**Arcos de volta:** vão de um vértice a seu ancestral na floresta

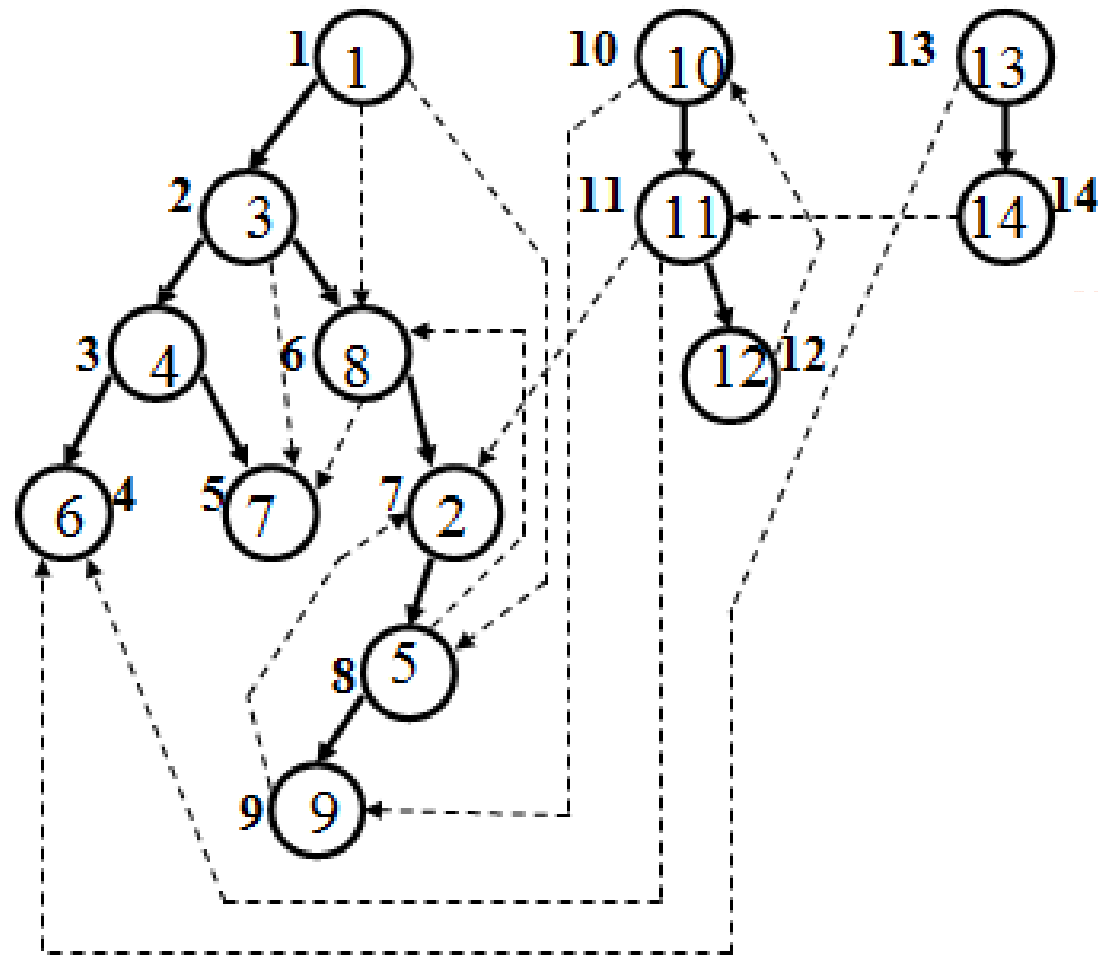
Um arco de um vértice para si mesmo é também um arco de volta

Exemplos:  $5 \rightarrow 8$ ,  $9 \rightarrow 2$ ,  
 $12 \rightarrow 10$



**Arcos para frente:** arcos *não de árvore*, que vão de um vértice a um seu descendente próprio

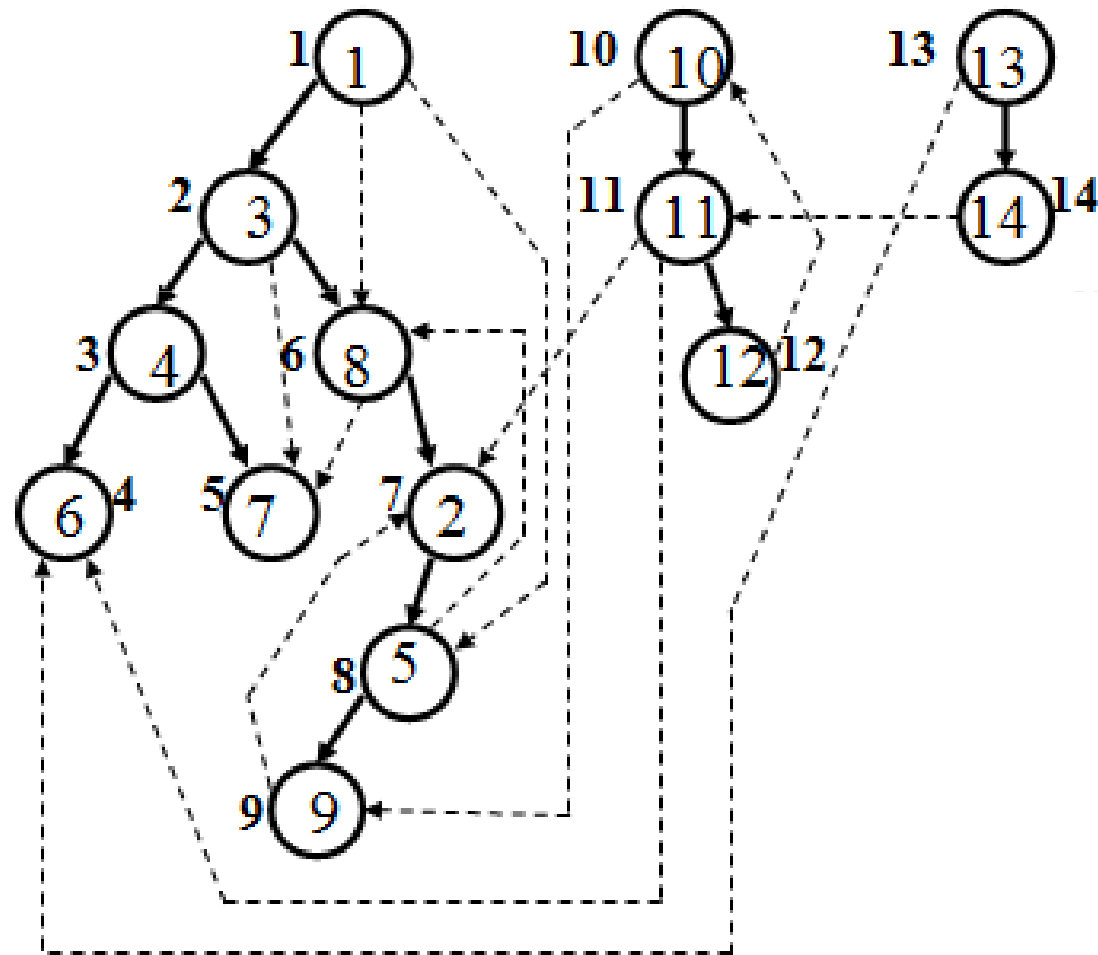
Exemplos:  $1 \rightarrow 8$ ,  $1 \rightarrow 5$ ,  
 $3 \rightarrow 7$



**Arcos cruzantes:** vão de um vértice a outro que não seja seu ancestral nem descendente na floresta

Vão sempre da direita para a esquerda

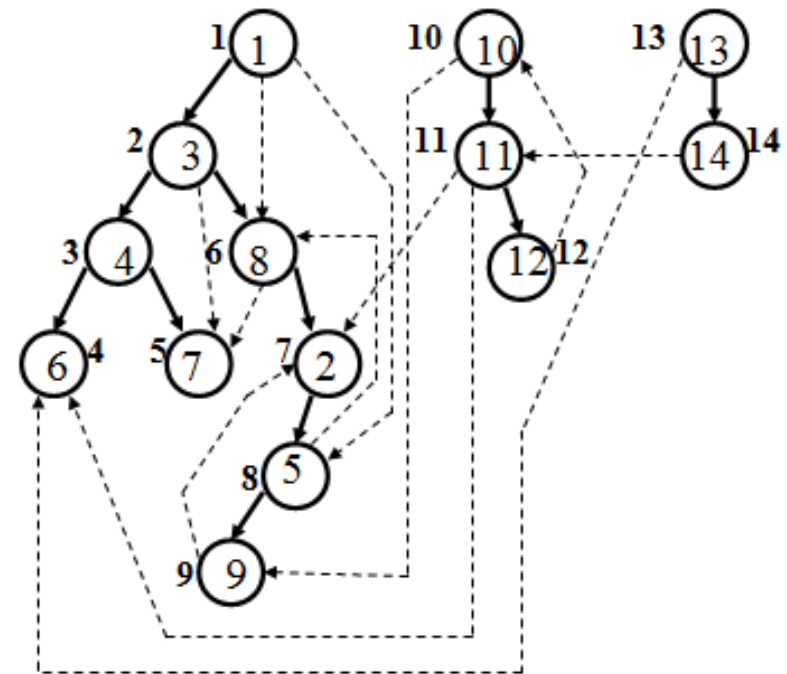
Exemplos:  $8 \rightarrow 7$ ,  $11 \rightarrow 6$ ,  
 $14 \rightarrow 11$ ,  $13 \rightarrow 6$ ,  $11 \rightarrow 2$ ,  
 $10 \rightarrow 9$



Características: se  $v \rightarrow w$  é

- Arco de árvore ou arco para frente:  $nvis(v) < nvis(w)$
- Arco de volta:  $nvis(v) \geq nvis(w)$
- Arco cruzante:  $nvis(v) > nvis(w)$

Tal floresta e tal classificação de arcos são usados para resolver muitos problemas sobre grafos.





## 3.3 – Digrafos acíclicos (dga's)

### a) Aspectos gerais

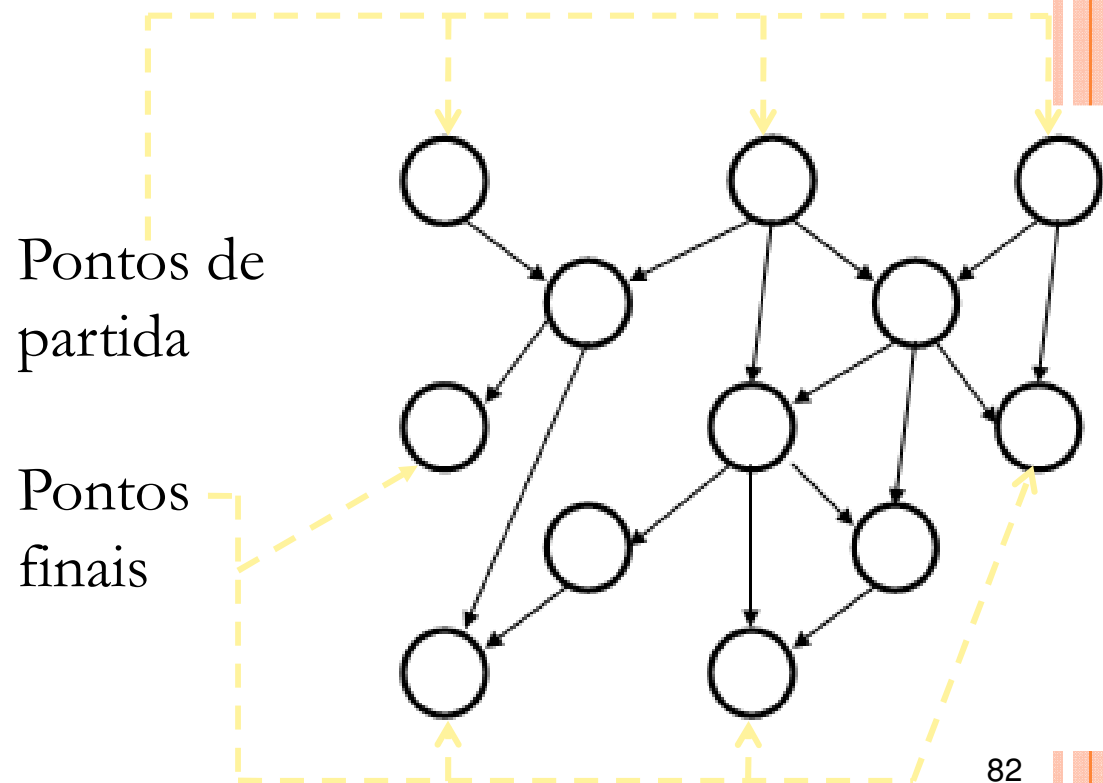
- Dga é um digrafo sem ciclos
- Dga é um caso particular de digrafos e árvore é um caso particular de dga's



- **Ponto de partida** em um dga: vértice não adjacente de nenhum outro vértice.
- **Ponto final** em um dga: vértice sem adjacentes.
- Um dga tem um ou mais pontos de partida e um ou mais pontos finais.

- **Exemplo:**  
Redes PERT-CPM

Uma tarefa não pode começar enquanto aquelas das quais ela depende não terminarem



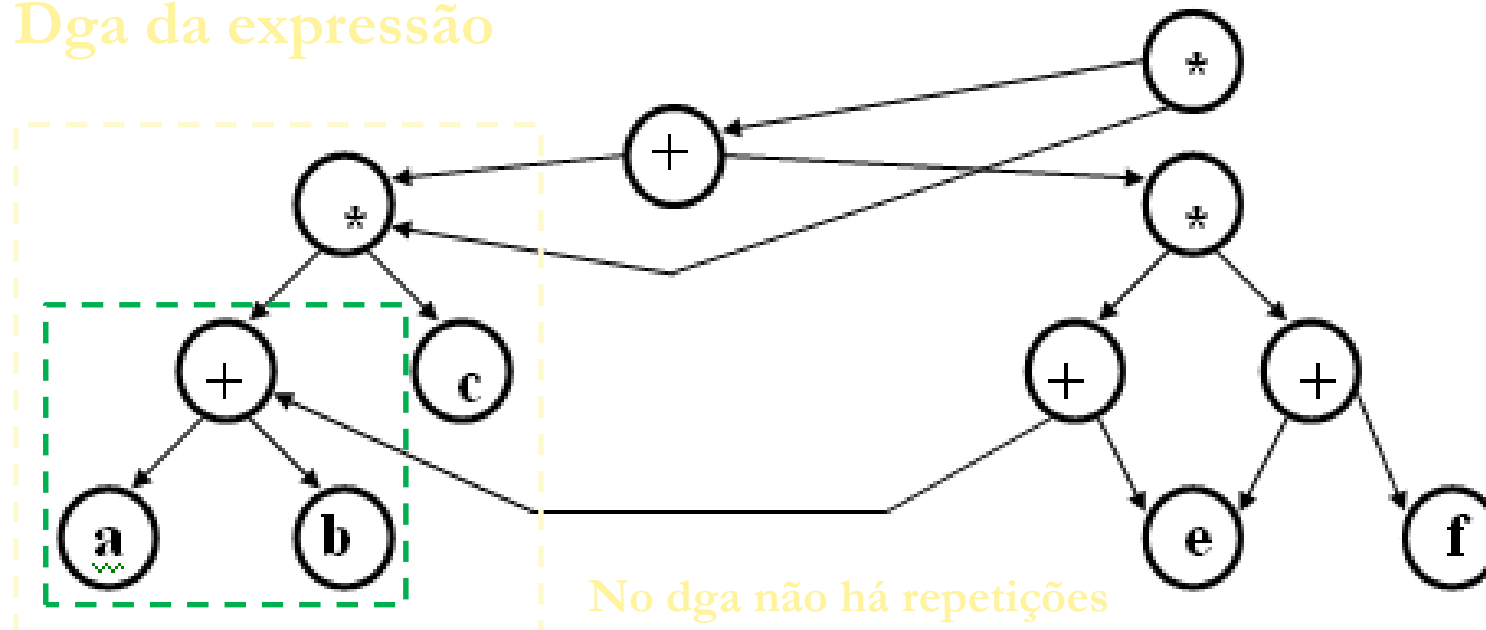
- **Exemplo:** expressões aritméticas com sub-expressões repetidas

Seja a expressão:

$$((a+b)*c + ((a+b)+e) * (e+f)) * ((a+b)*c)$$

Diagram illustrating the expression with annotations for repetitions. Dashed orange boxes and arrows labeled "repetições" indicate that the sub-expressions  $(a+b)*c$  and  $(a+b)$  are repeated in the expression.

Dga da expressão

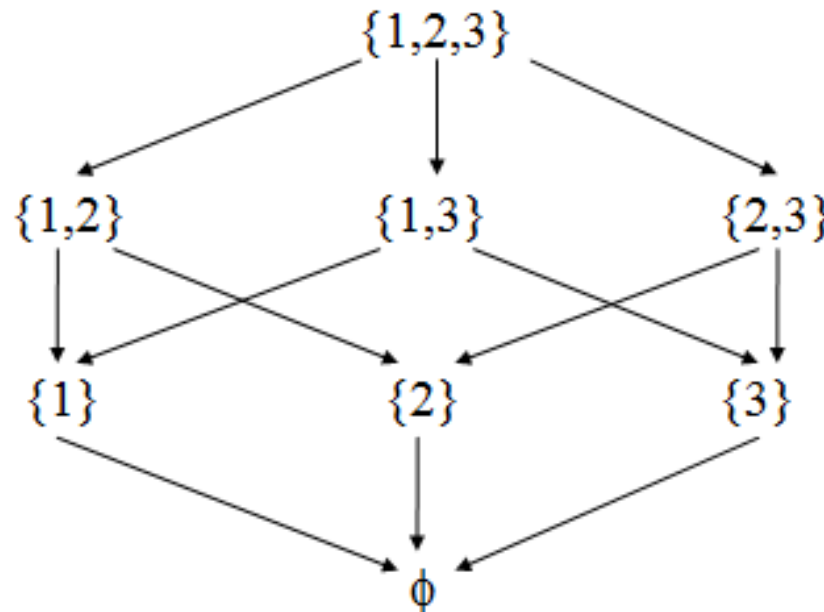


- **Exemplo:** Seja  $\text{Conj} = \{1, 2, 3\}$  e seja  $P(\text{Conj})$  o conjunto potência de  $\text{Conj}$

$$P(\text{Conj}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$

Seja a relação  $\omega = (\text{Contém e tem exatamente 1 elemento a mais que})$ , relação essa definida em  $P(\text{Conj})$ .

$P(\text{Conj})$  e  $\omega$  :  
dga:



pele seguinte

## b) Teste de aciclicidade

- Certas aplicações exigem que o digrafo seja acíclico
- É necessário verificar se o digrafo lido é um dga, para garantir a consistência dos dados de entrada
- Pode-se usar a **busca em profundidade** como esqueleto para sua solução
- Se um **arco de volta** for encontrado durante a busca, então o grafo **não é acíclico**



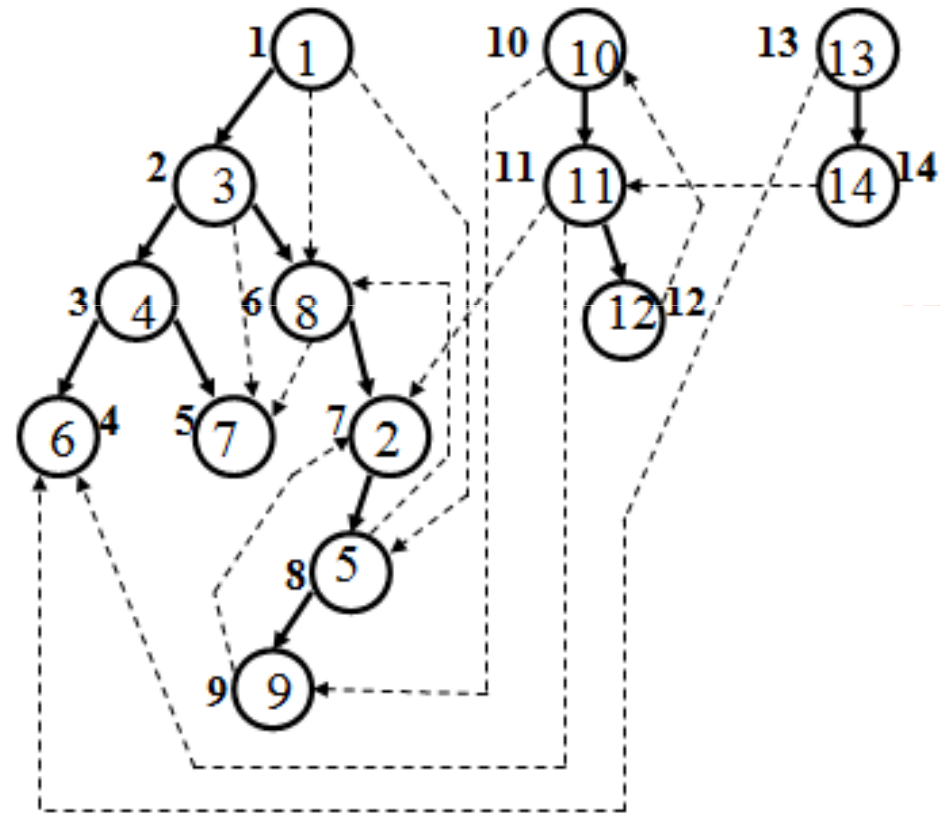
Para fazer este teste, basta:

- Manter uma pilha com todos os ancestrais do vértice que está sendo visitado, incluindo ele próprio
- Partindo dele, para visitar seus adjacentes, se um deles, já visitado, pertencer a tal pilha, existe o **arco de volta** e o grafo não é acíclico
- Quando todos os seus adjacentes já tiverem sido visitados, deve-se retirá-lo da pilha de ancestrais



## Exemplo:

- Ao visitar o vértice **5**, estarão na pilha seus ancestrais **1, 3, 8, 2 e 5**
- Adjacentes de **5**: **8 e 9**
- Na tentativa de visitar o **8**, encontra-se o **arco de volta**
- O digrafo não é um **dga**
- A seguir um algoritmo para o teste de aciclicidade



```

/* Variaveis globais */

int cont; pilha P; logic aciclico;

void Travessia (Grafo *G) {
    vertice v;
InicPilha (&P); aciclico = TRUE:
    for (v = 1; v <= G->nvert; v++)
        G->Vertices[v].visit = FALSE;
    cont = 0;
    for (v = 1; v <= G->nvert; v++)
        if (G->Vertices[v].visit == FALSE)
            BuscaProf (v, G);
}

```

O algoritmo usa como **esqueleto** o algoritmo da **busca em profundidade** (em **negrito** os enxertos)





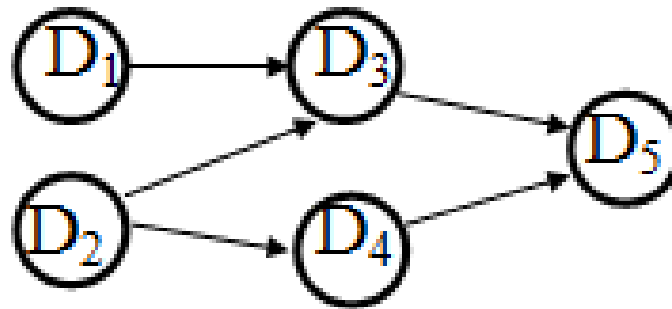
```

void BuscaProf (vertice v, Grafo *G)  {
    celulaadj *p;
    G->Vertices[v].visit = TRUE;    cont++;
    G->Vertices[v].nvis = cont;
Empilhar (v, &P);
    p = G->Vertices[v].adjs;
    while (p != NULL)  {
        if (G->Vertices[p->vert].visit == FALSE)
            BuscaProf(p->vert, G);
        else if (Procurar (p->vert, P) == TRUE)
            aciclico = FALSE;
        p = p->prox;
    }
Desempilhar (&P);
}

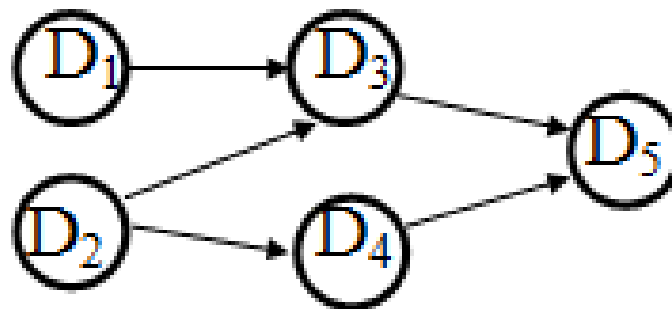
```

## c) Ordenação topológica

- Um grande projeto costuma ser dividido numa coleção de tarefas, algumas das quais são **pré-requisitos** de outras.
- **Exemplo:** disciplinas  $D_1, D_2, D_3, D_4$  e  $D_5$  de um curso



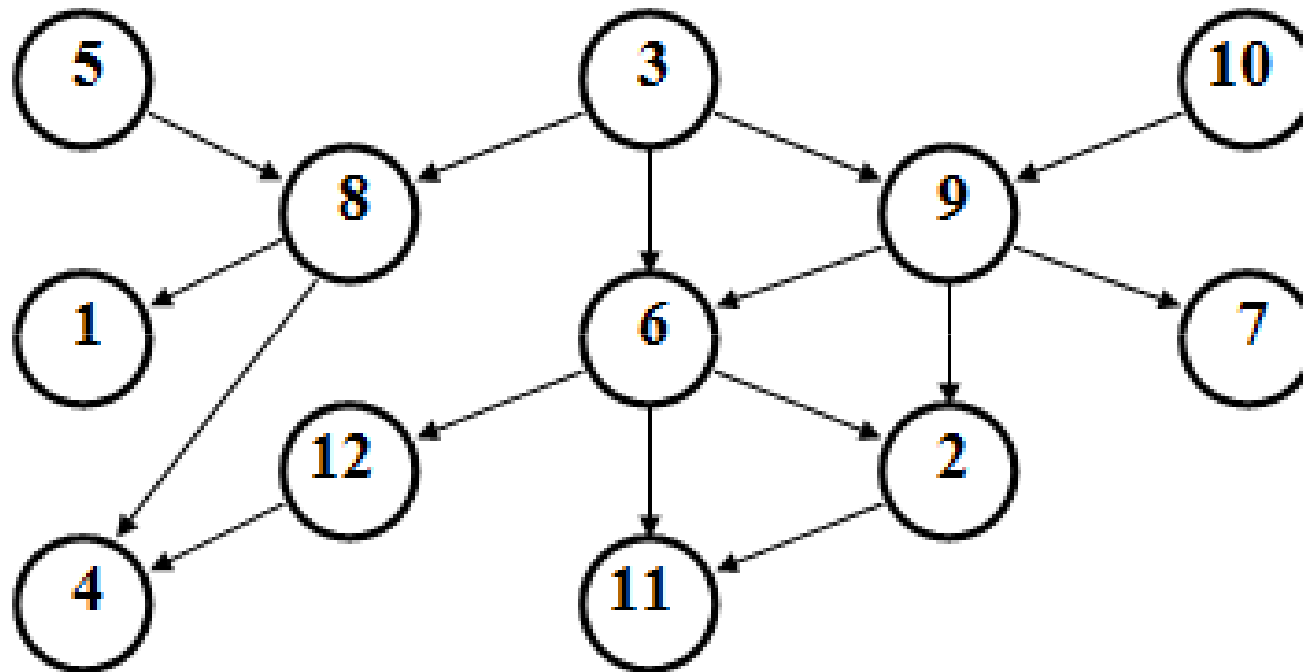
- **Ordenação topológica:** processo de ordenar linearmente os vértices de um dga, de forma que
  - Se há um arco do vértice  $i$  para o vértice  $j$ , então  $i$  aparece antes de  $j$  na ordenação.
- **Exemplo:**  $(D_1, D_2, D_3, D_4, D_5)$  e  $(D_2, D_4, D_1, D_3, D_5)$  são ordenações topológicas do dga abaixo



- Essa ordenação pode ser feita, usando como esqueleto a **busca em profundidade**
- No final da chamada da rotina **BuscaProf**, empilha-se o vértice argumento
- No final da rotina **Travessia**, enquanto houver elementos na pilha, retira-se o vértice do topo da pilha, imprimindo-o

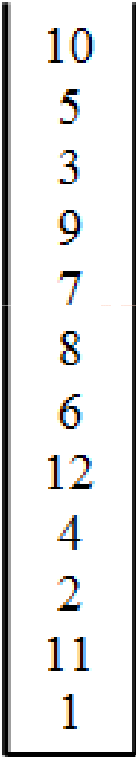
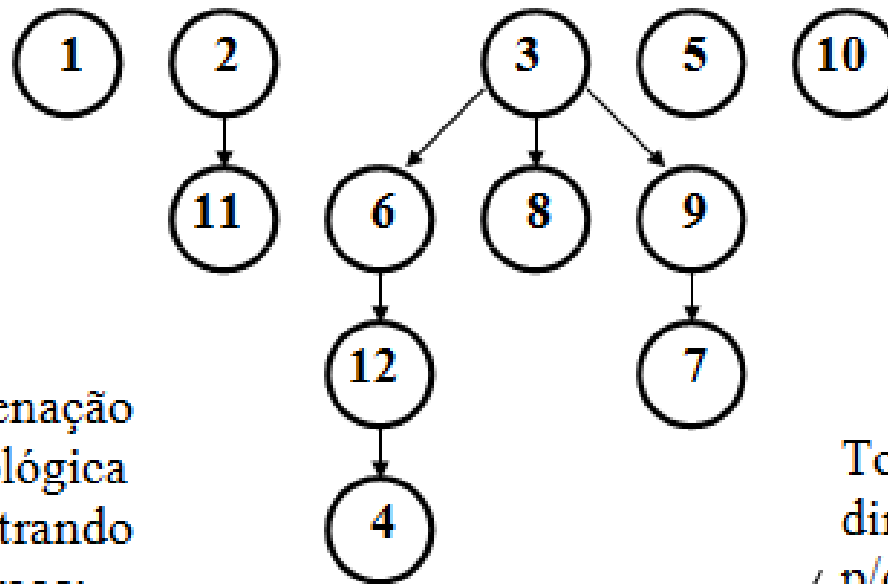
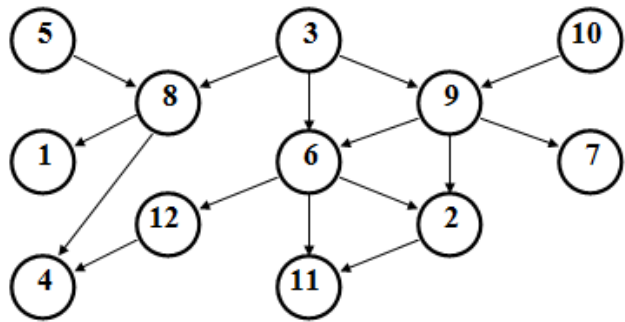


- **Exemplo:** seja um dga com numeração aleatória dos vértices:



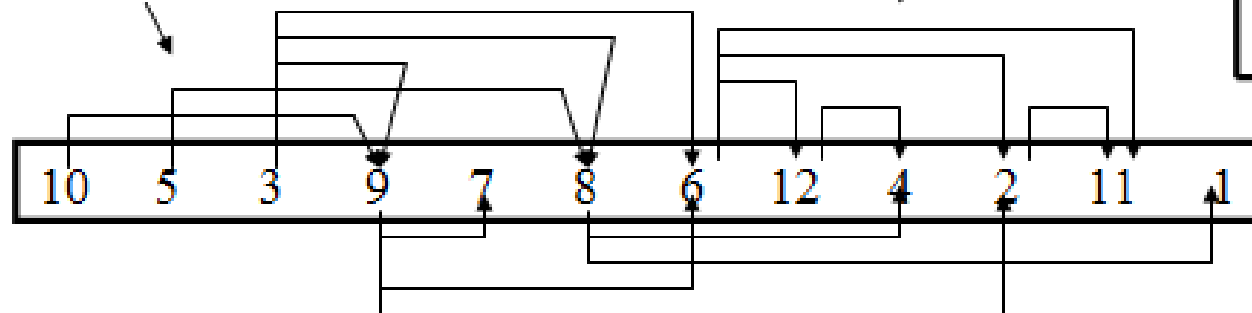
Opta-se por iniciar uma busca pelo vértice de menor número

# Floresta da busca em profundidade (só com os arcos de árvore):



Ordenação topológica mostrando os arcos:

Todas as direções p/direita



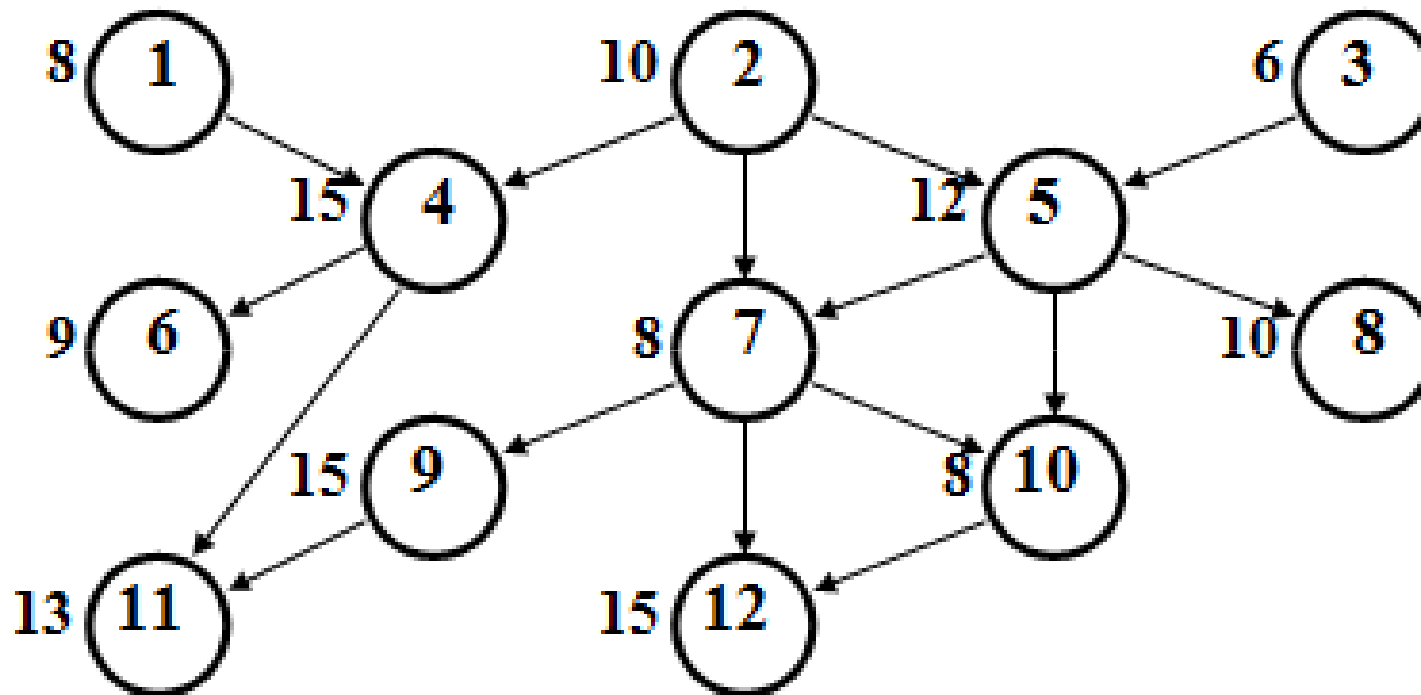
Esta é uma das muitas possíveis ordenações topológicas

- Ordenação topológica ajuda estabelecer uma ordem de execução das tarefas de um grande projeto
- Isso é crítico quando apenas um pequeno número de tarefas pode ser executado simultaneamente
- É o caso de um curso com módulos semestrais, com cinco ou seis disciplinas por módulos
- Faz-se um grafo de pré-requisitos das disciplinas e uma ordenação topológica mais sofisticada, para obter a simultaneidade requerida



## d) Caminho crítico um dga

- Seja o seguinte dga:



Os vértices são **tarefas** de um projeto

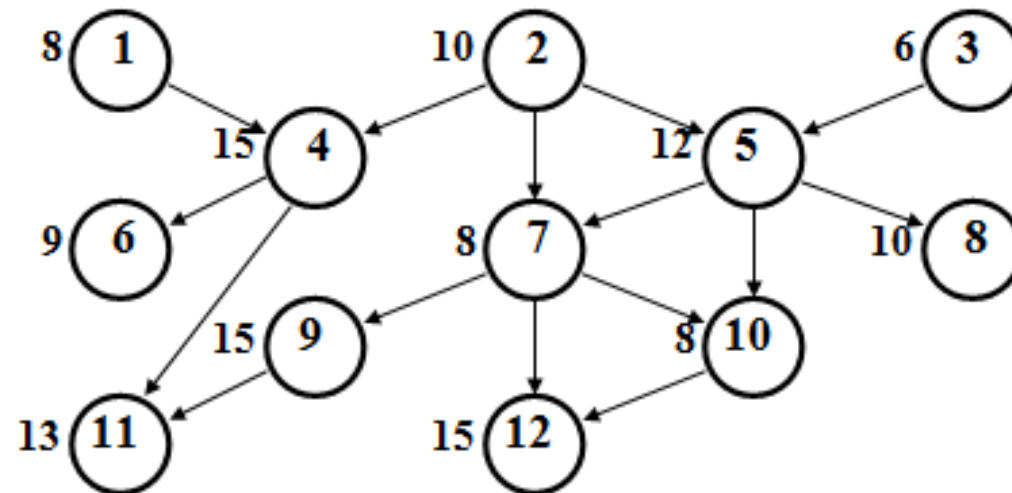
A **duração** de cada tarefa aparece ao lado do vértice

Num arco, o **destino** não começa antes **do** término da **fonte**

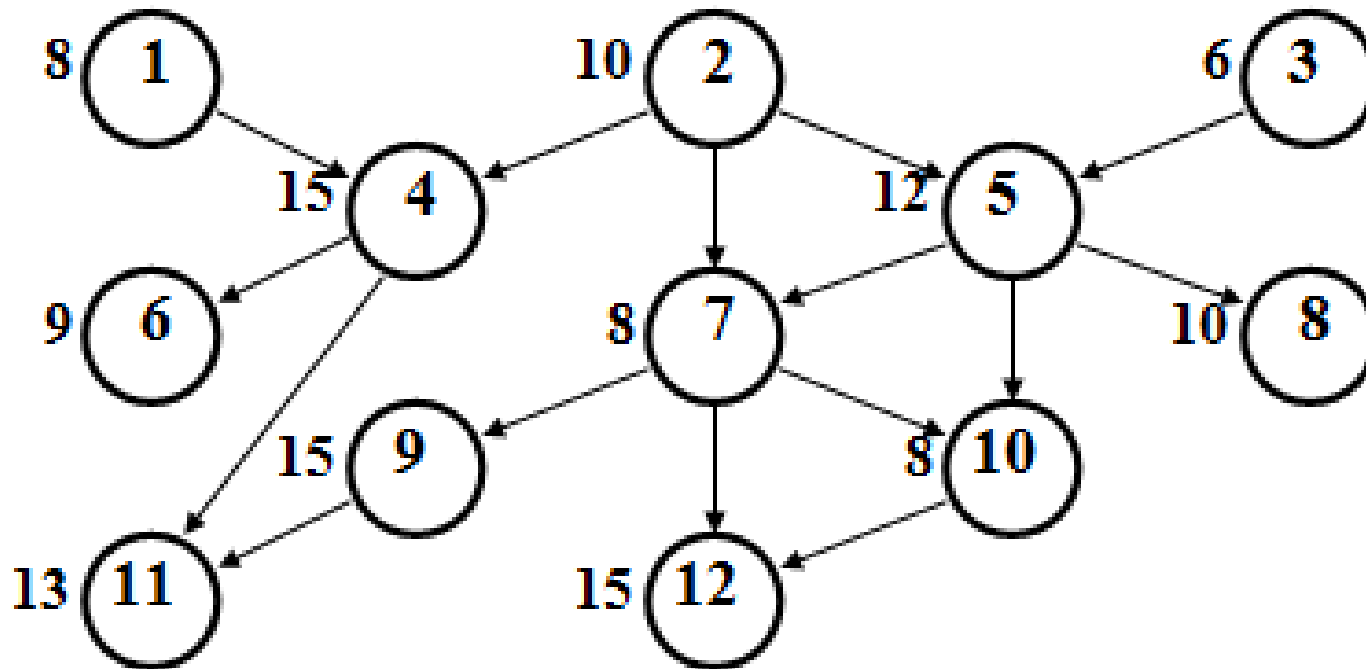


## Problemas:

- Qual o **tempo mínimo** para que o projeto seja feito?
- Quais as tarefas que não podem sofrer qualquer aumento no tempo de execução, para que esse tempo mínimo não aumente?
- A seqüência formada pelas referidas tarefas é chamada de **caminho crítico** (pode haver mais de um)

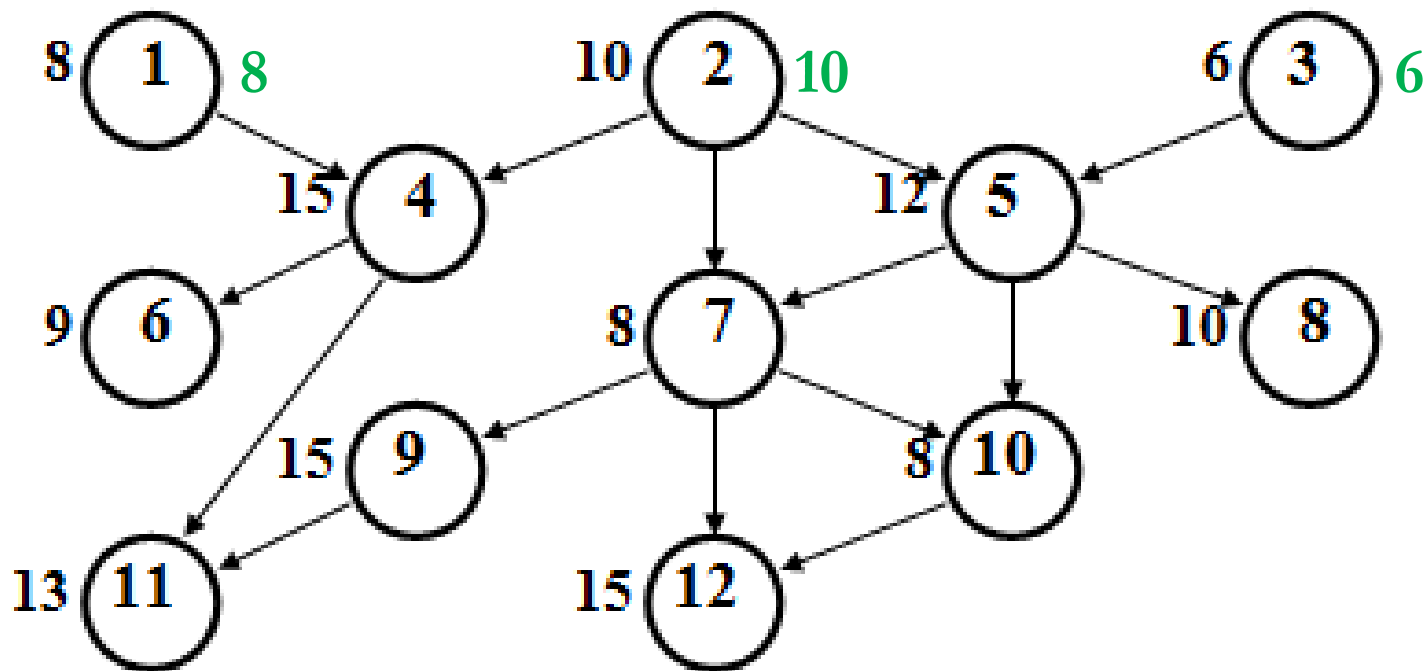


- É útil calcular o **tempo mínimo de término (TMT)** de cada vértice
- O tempo mínimo procurado é o maior de todos os **TMT's das tarefas finais**.
- **Tarefas finais** são aquelas que não tem adjacentes



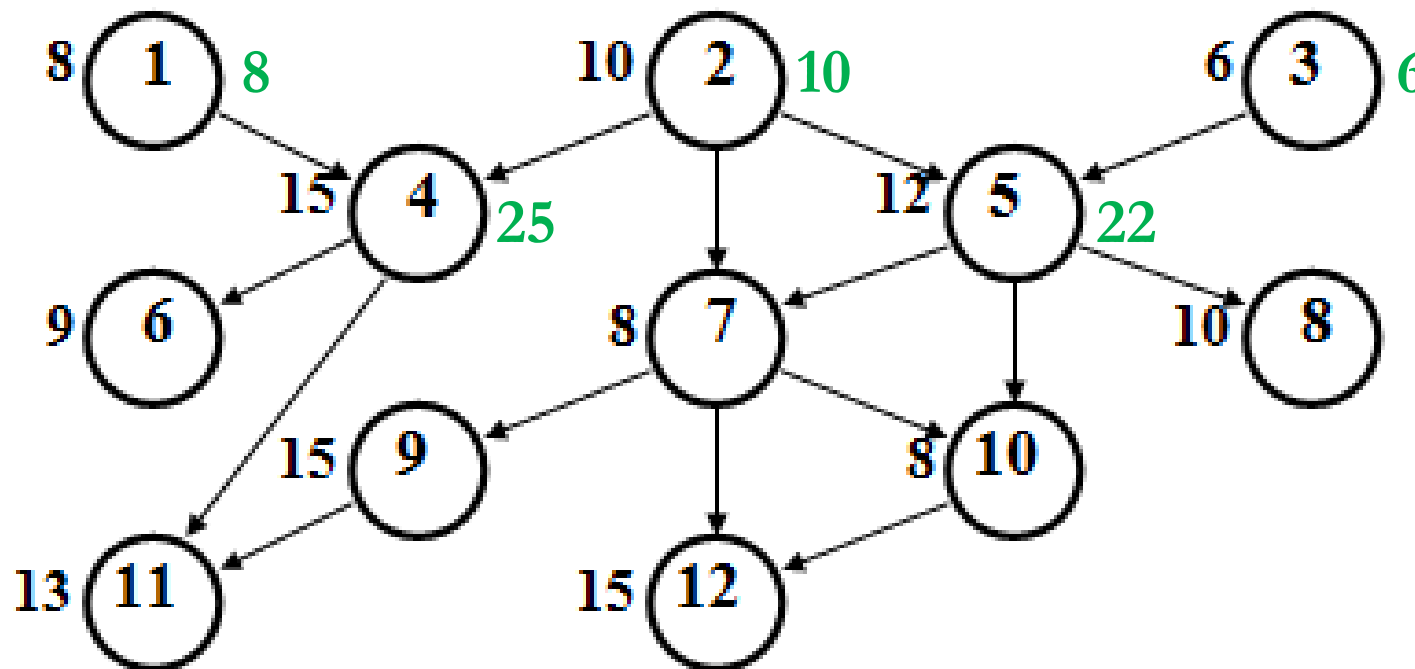
- TMT's das tarefas de partida **1**, **2** e **3** são imediatos:

- É a duração delas: **8**, **10** e **6**



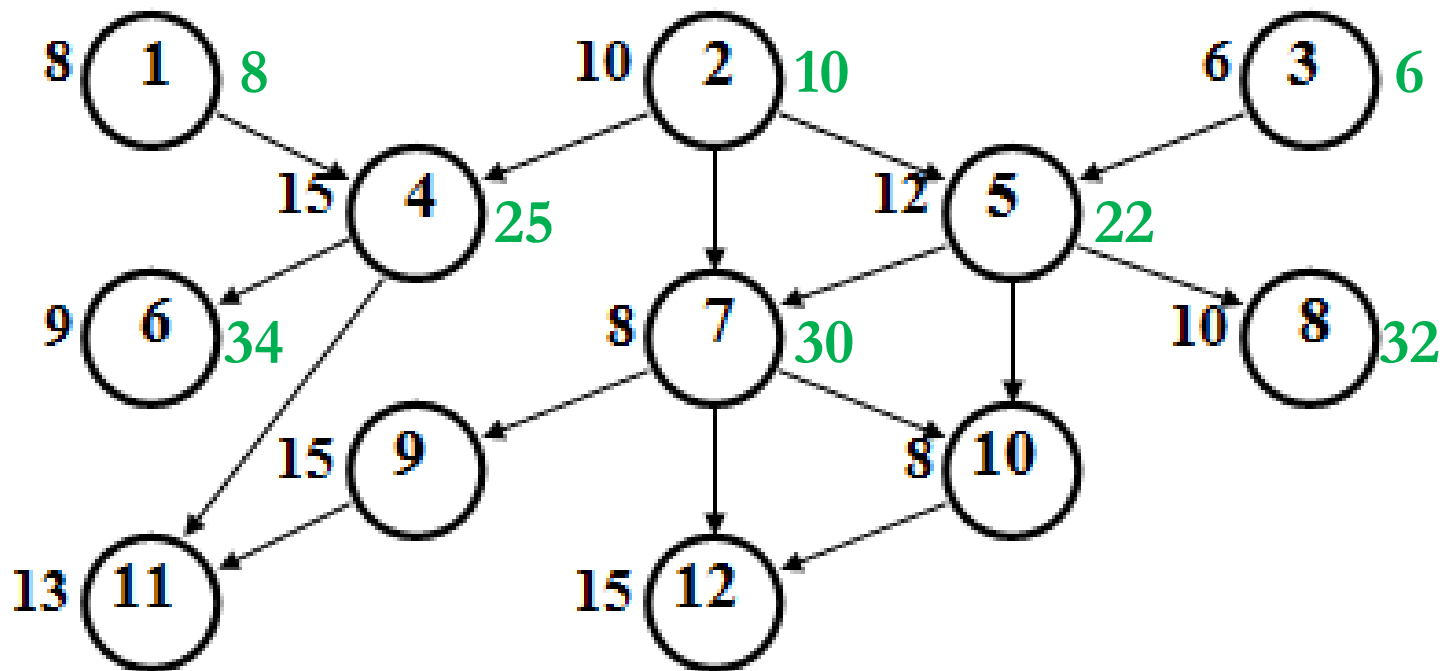
o O TMT das tarefas dependentes apenas de **1**, **2** e **3** já pode ser calculado:

- $TMT(4) = \max(TMT(1), TMT(2)) + \text{Duração}(4) = 25$
- $TMT(5) = \max(TMT(2), TMT(3)) + \text{Duração}(5) = 22$



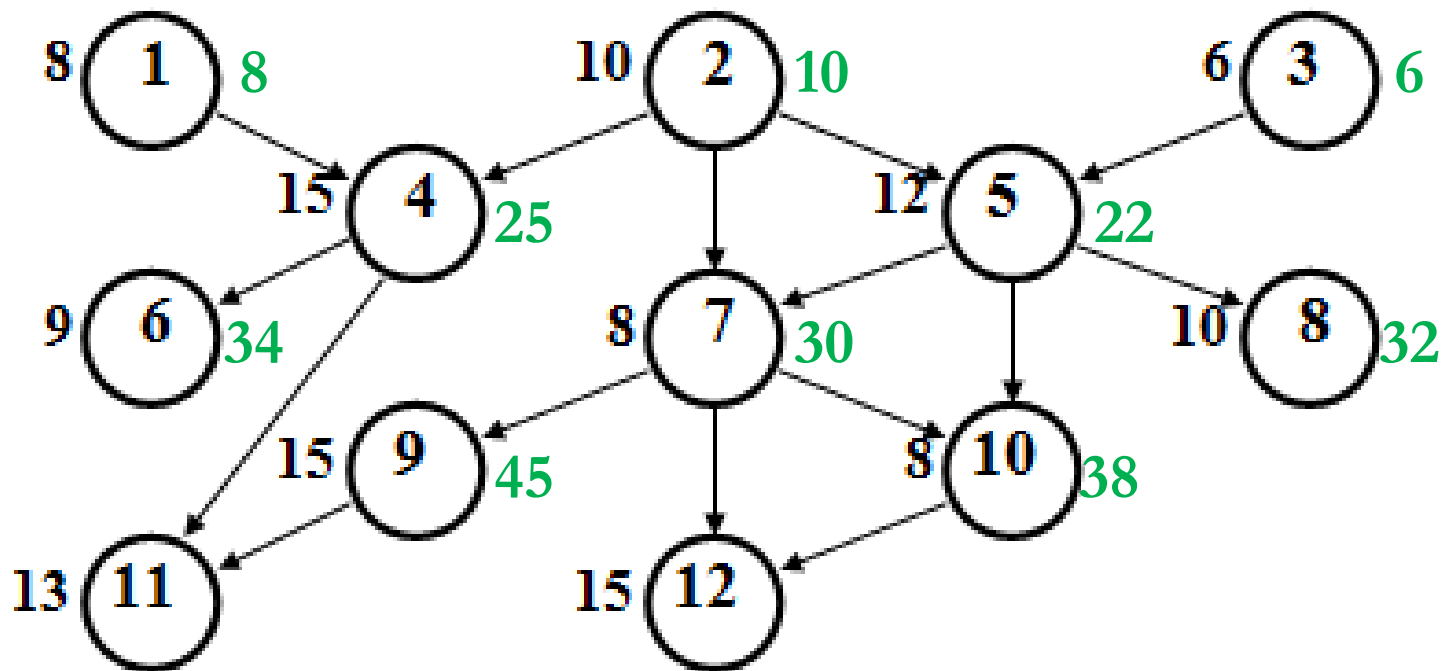
○ Assim prosseguindo:

- $TMT(6) = TMT(4) + Duração(6) = 34$
- $TMT(7) = \max(TMT(2), TMT(5)) + Duração(7) = 30$
- $TMT(8) = TMT(5) + Duração(8) = 32$



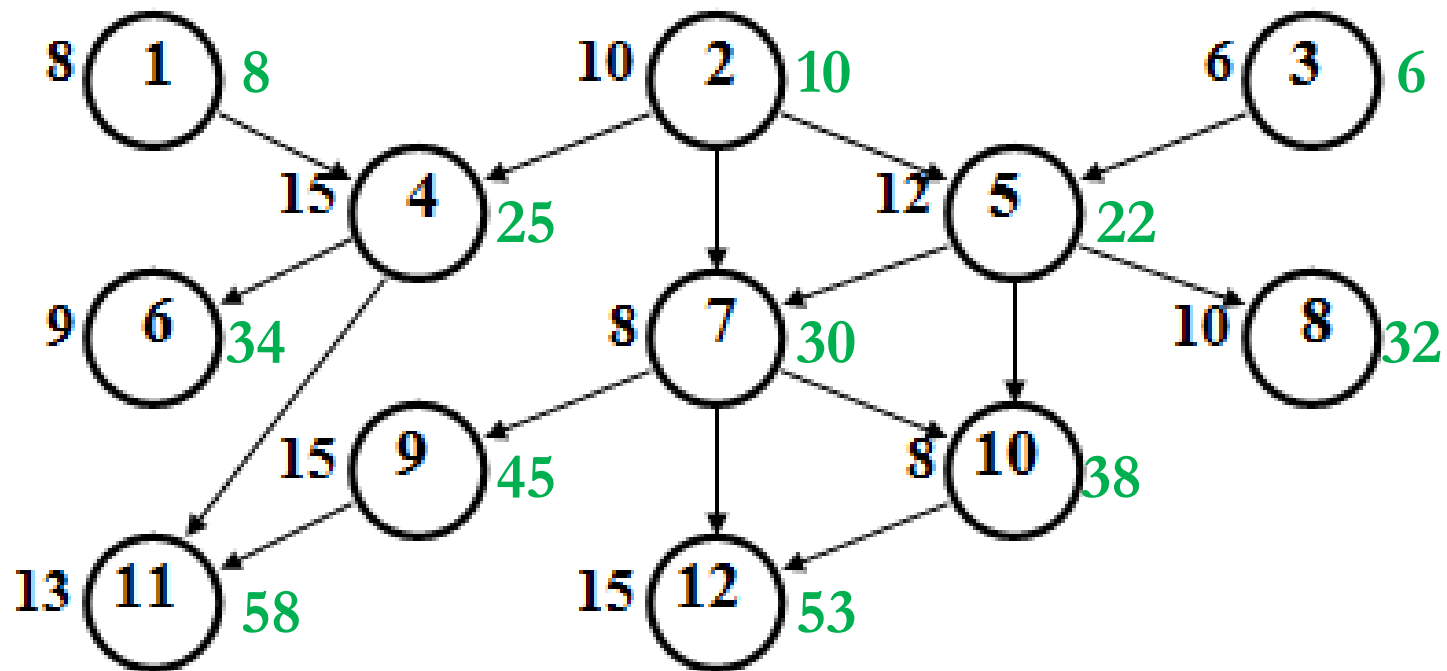
○ Assim prosseguindo:

- $TMT(9) = TMT(7) + \text{Duração}(9) = 45$
- $TMT(10) = \max(TMT(5), TMT(7)) + \text{Duração}(10) = 38$



○ Finalmente:

- $TMT(11) = \max(TMT(4), TMT(9)) + \text{Duração}(11) = 58$
- $TMT(12) = \max(TMT(7), TMT(10)) + \text{Duração}(12) = 53$

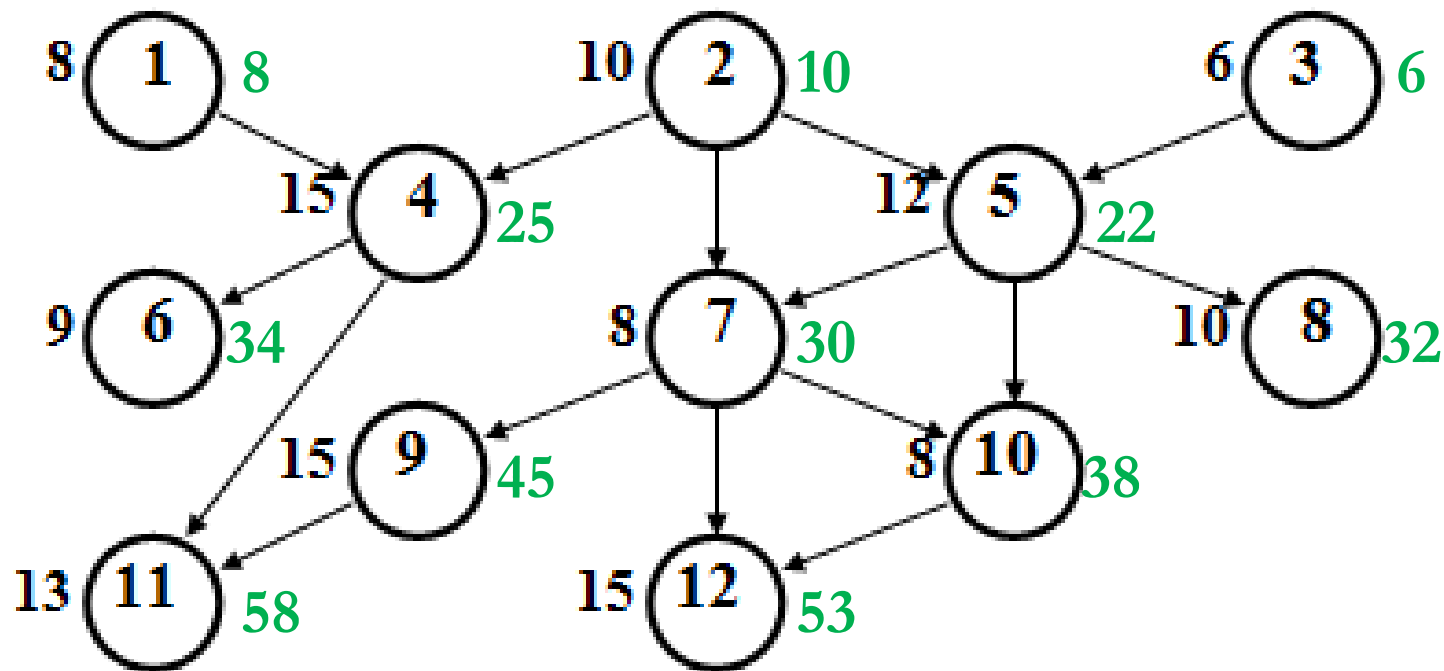


○ Tarefas finais: **6, 8, 11 e 12**

– TMT (Projeto) =

$$= \max (\text{TMT} (6), \text{TMT} (8), \text{TMT} (11), \text{TMT} (12))$$

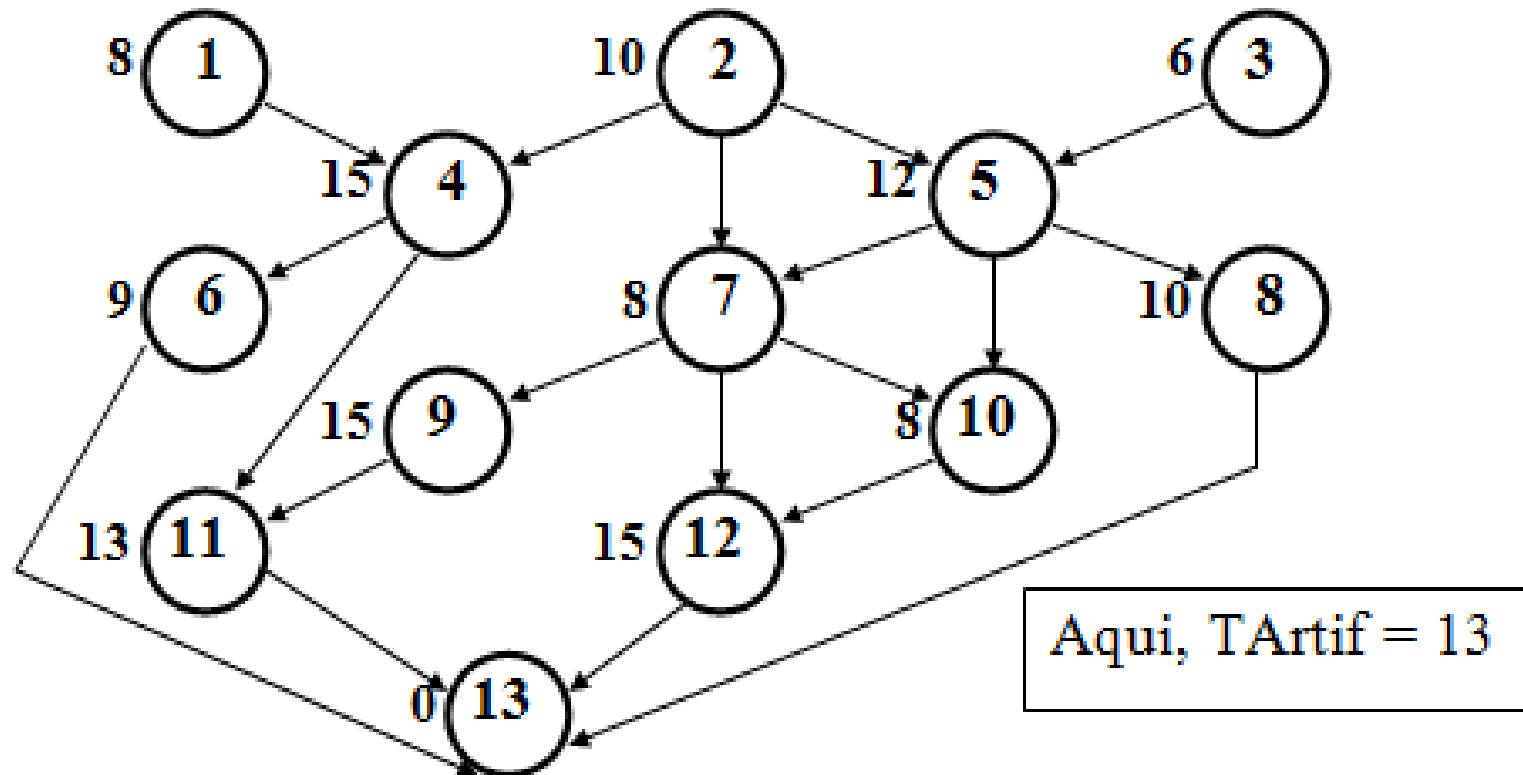
$$= \max (34, 32, 58, 53) = 58$$





## Algoritmo recursivo para o cálculo do TMT do projeto:

- Primeiramente introduz-se uma tarefa final artificial **TArtif** de duração zero:

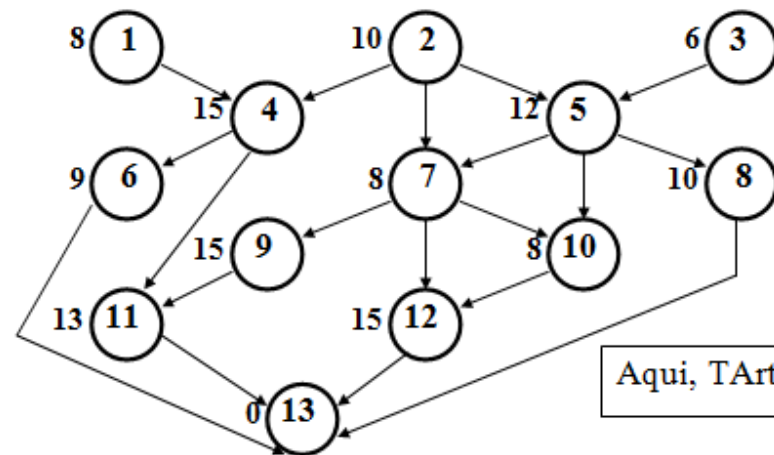


## Algoritmo:

```
int Termino (Tarefa T) {  
    int maior, aux; Tarefa X;  
    maior = 0;  
    para (cada tarefa X da qual T é adjacente) {  
        if(X.tmt não foi calculado)  
            aux = Termino (X);  
        else aux = X.tmt;  
        if(aux > maior) maior = aux;  
    }  
    T.tmt = maior + T.duracao;  
    return T.tmt;  
}
```

É útil manter para cada tarefa, uma lista das tarefas das quais ela é adjacente, além da lista das adjacentes a ela (grafo reverso)

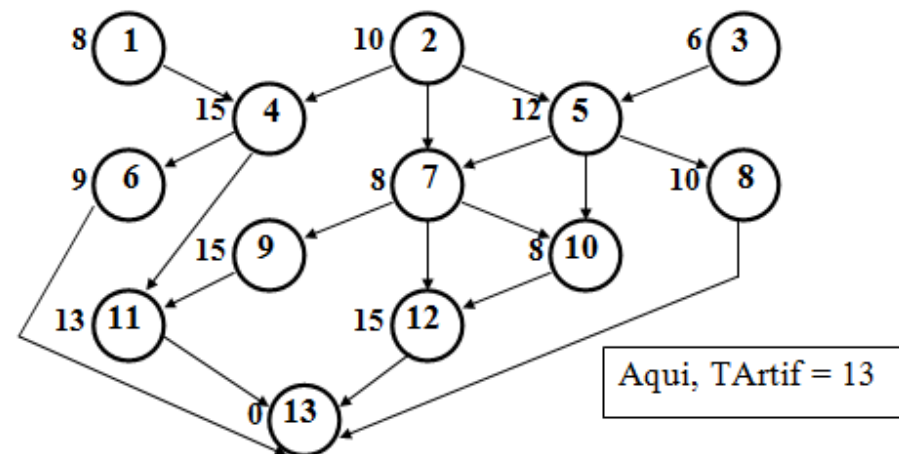
O tempo mínimo para a execução do projeto é calculado invocando **Termino (TArtif)**



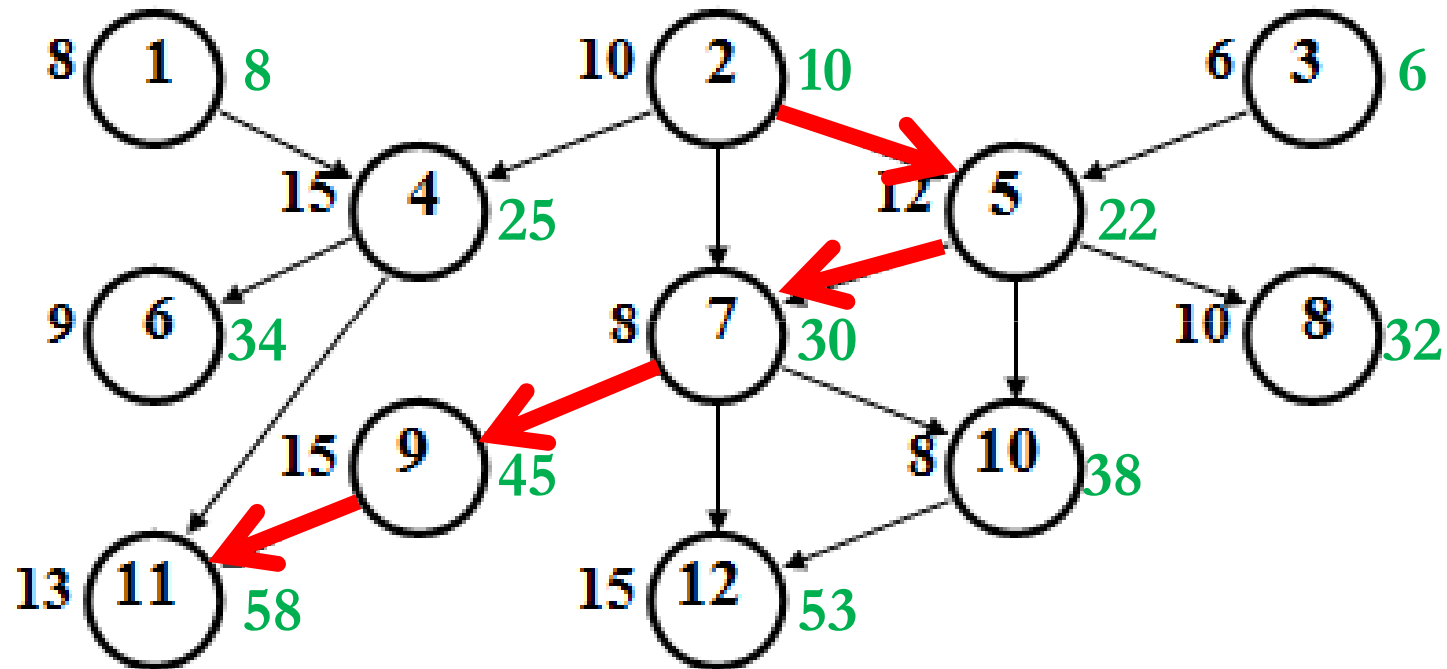
Aqui, TArtif = 13

As tarefas do caminho crítico são assim escolhidas:

- A primeira delas é a de maior TMT entre as finais, excluindo **TArtif**
- Para cada tarefa T acrescentada ao caminho crítico, a próxima é aquela de maior TMT dentre as que T é adjacente
- A última a ser acrescentada é uma que não é adjacente de nenhuma outra.



## Exemplo: caminho crítico do grafo ilustrativo



## 3.4 – Componentes fortemente conexos (CFC's)

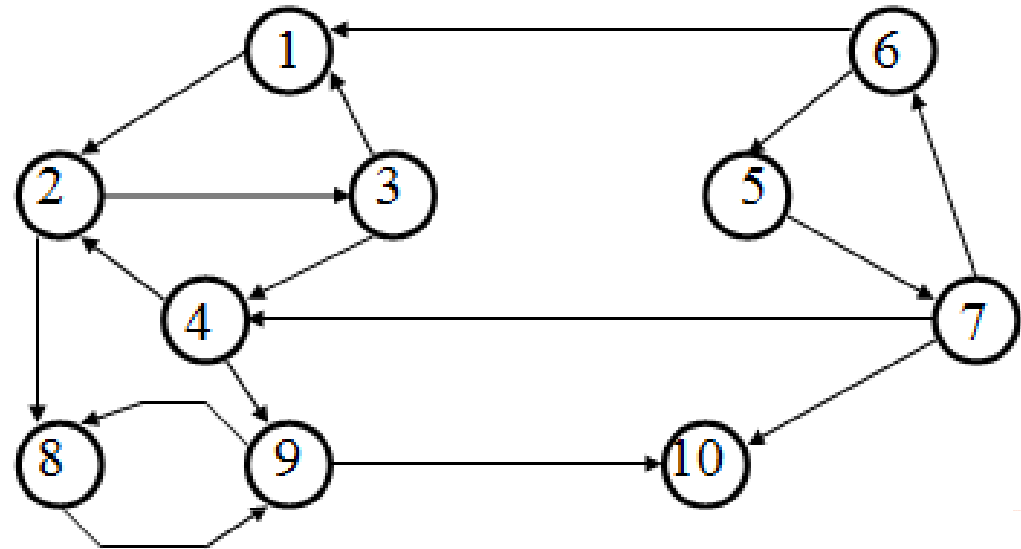
### a) Conceito

- **CFC** de um digrafo  $G = \{V, A\}$  é um sub-grafo de  $G$  contendo:
  - Um conjunto máximo de vértices de  $V$ , no qual há um caminho de qualquer desses vértices para qualquer outro do conjunto
  - Todos os arcos de  $A$  que ligam somente vértices desse conjunto
- Um CFC pode ser composto de apenas um vértice

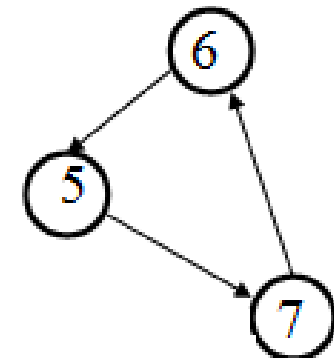
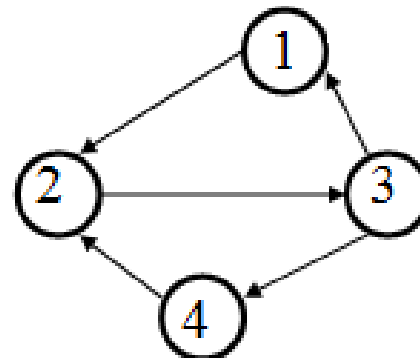


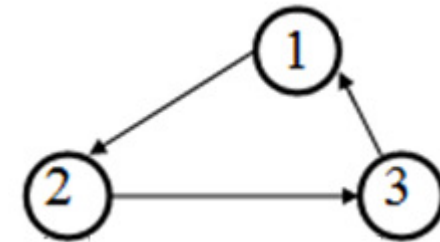
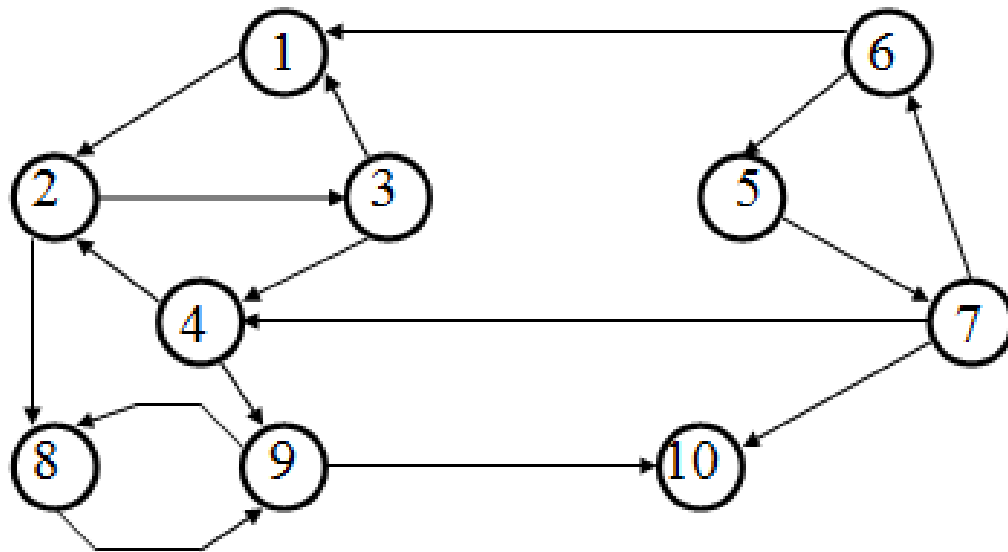
o **Exemplo:**

Seja  $G = \{V, A\}$   
o seguinte digrafo:



Eis seus CFC's:

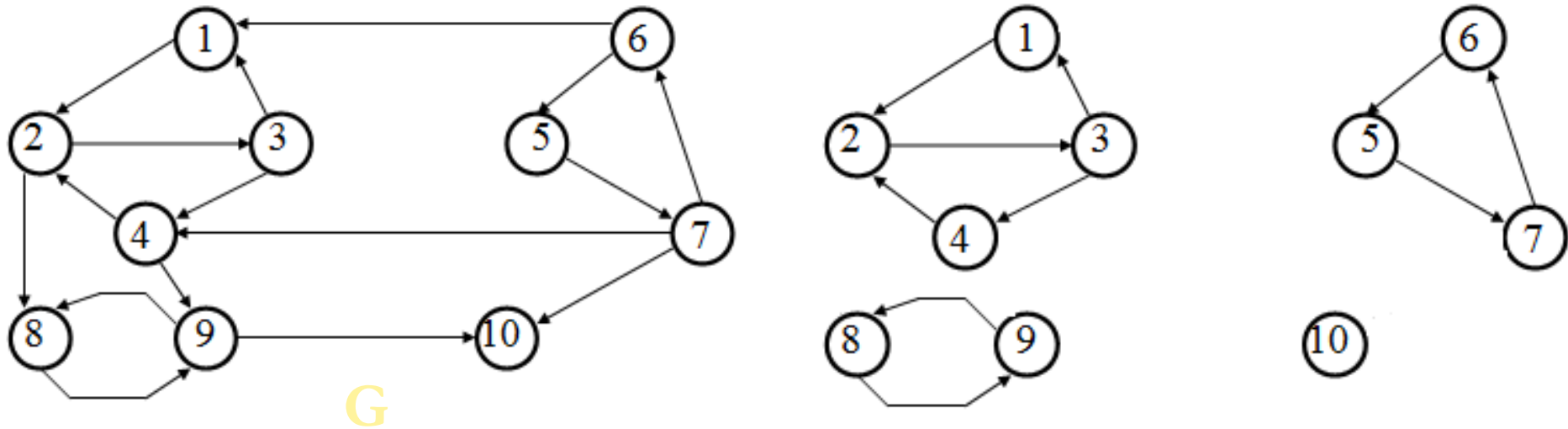




O sub-grafo  $G' = \{\{1, 2, 3\}, \{1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1\}\}$  não é um CFC, pois  $\{1, 2, 3\}$  não é máximo

Há caminho de 4 para 1, 2 e 3 e também desses para 4





- Qualquer vértice de  $G$  está em algum de seus CFC's
- Alguns arcos de  $G$  podem não estar em nenhum de seus CFC's; tais arcos são chamados **arcos que cruzam componentes**

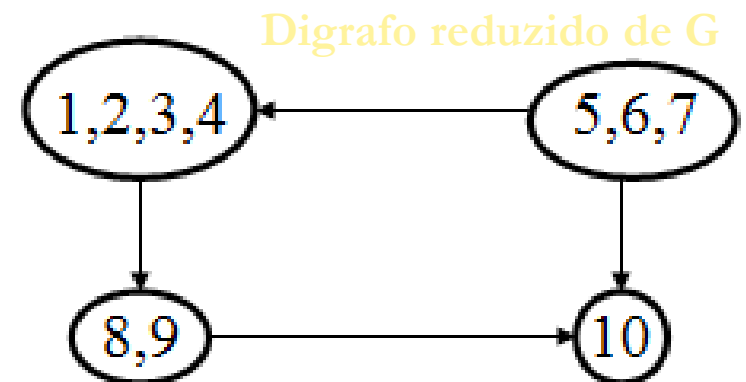
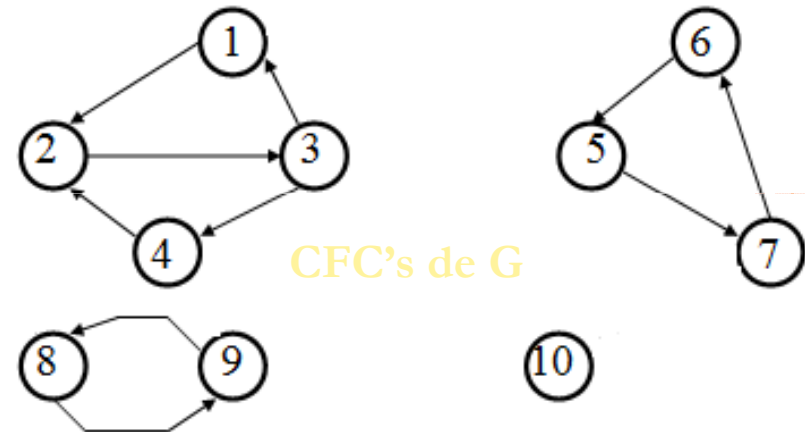
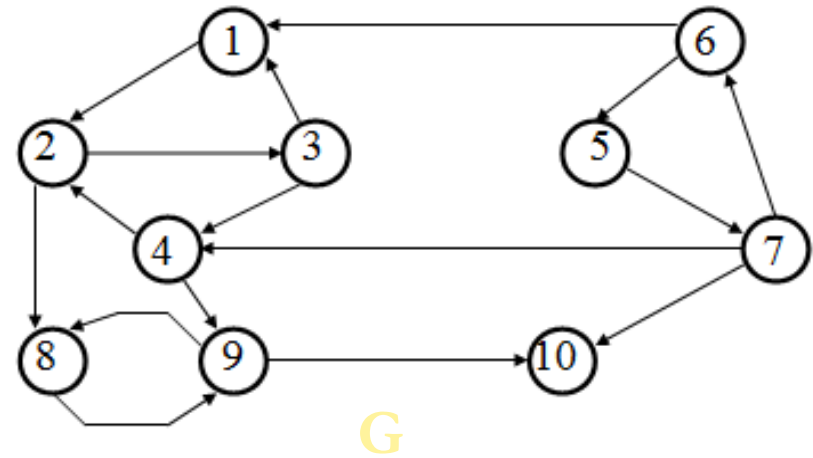




**Digrafo reduzido** de um digrafo é aquele:

- Em que cada vértice é o conjunto de vértices de um dos **CFC's** desse digrafo
- Cujos arcos são unificações de seus **arcos que cruzam componentes**

Digrafos reduzidos são sempre acíclicos.



## **b) Aplicações de CFC's**

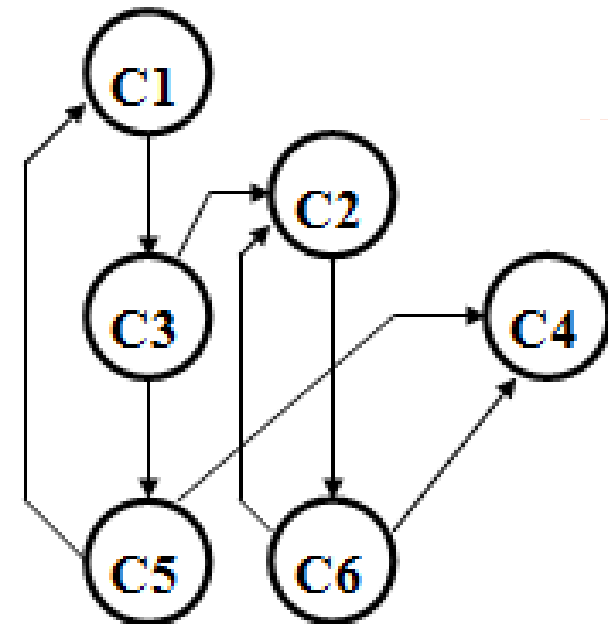
- **Divisão** de problemas sobre digrafos em **subproblemas**, um para cada CFC
- Estudo de privacidade em sistemas de comunicação
- Análise do fluxo de controle para a validação de programas
- Computer-aided design (CAD)
- Análise de circuitos eletrônicos – classes de equivalência
- Paralelização de laços seqüenciais



## Exemplo: Paralelização de laços seqüenciais

- Seja o seguinte laço em C e o grafo de dependências entre seus comandos:

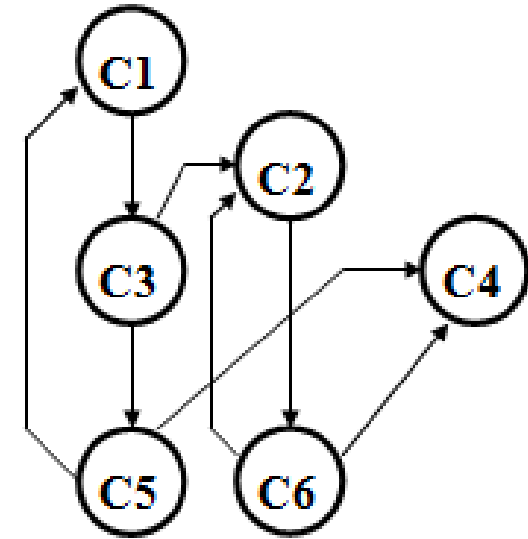
```
for (i = 1; i <= n; i++) {  
C1:      A[i] = B[i] + C[i];  
C2:      D[i] = E[i] + F[i];  
C3:      E[i+1] = A[i] + G[i];  
C4:      H[i] = F[i] + B[i];  
C5:      B[i+1] = E[i+1] + M[i];  
C6:      F[i+1] = D[i] + N[i];  
}
```



```

for (i = 1; i <= n; i++) {
C1:      A[i] = B[i] + C[i];
C2:      D[i] = E[i] + F[i];
C3:      E[i+1] = A[i] + G[i];
C4:      H[i] = F[i] + B[i];
C5:      B[i+1] = E[i+1] + M[i];
C6:      F[i+1] = D[i] + N[i];
}

```

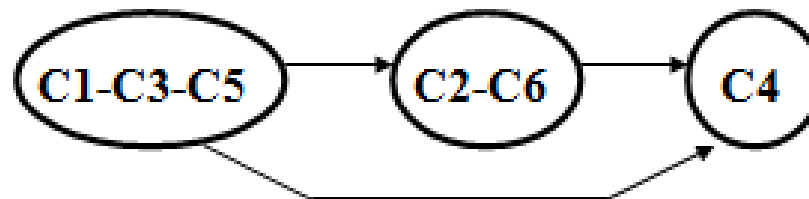


○ CFC's (só os vértices) e digrafo reduzido:

1º: C1-C3-C5

2º: C2-C6

3º: C4

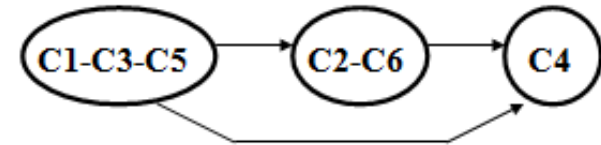


- O laço original pode ser decomposto em tantos laços quantos forem os CFC's:

```
for (i = 1; i <= n; i++) {  
C1:     A[i] = B[i] + C[i];  
C3:     E[i+1] = A[i] + G[i];  
C5:     B[i+1] = E[i+1] + M[i];  
}
```

```
for (i = 1; i <= n; i++) {  
C2:     D[i] = E[i] + F[i];  
C6:     F[i+1] = D[i] + N[i];  
}
```

```
for (i = 1; i <= n; i++) {  
C4:     H[i] = F[i] + B[i];  
}
```



Em muitos casos é mais fácil aplicar técnicas de paralelização em laços menores do que em laços maiores



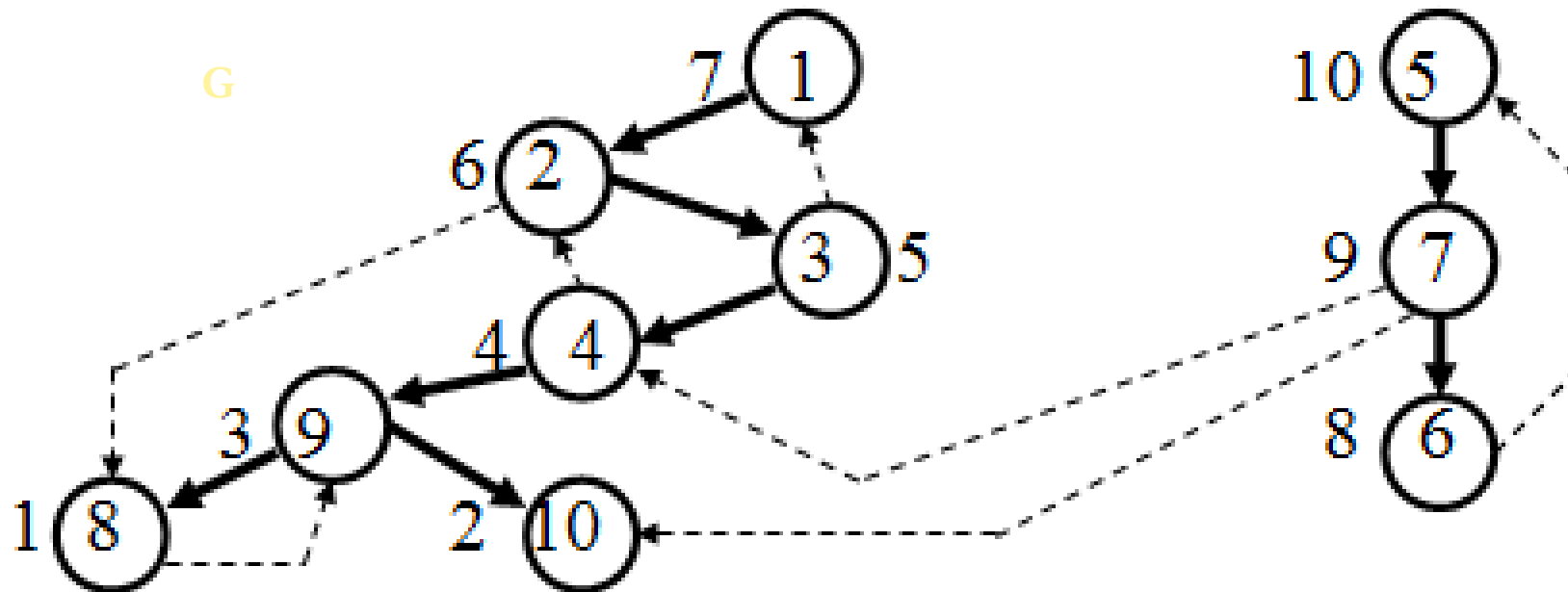
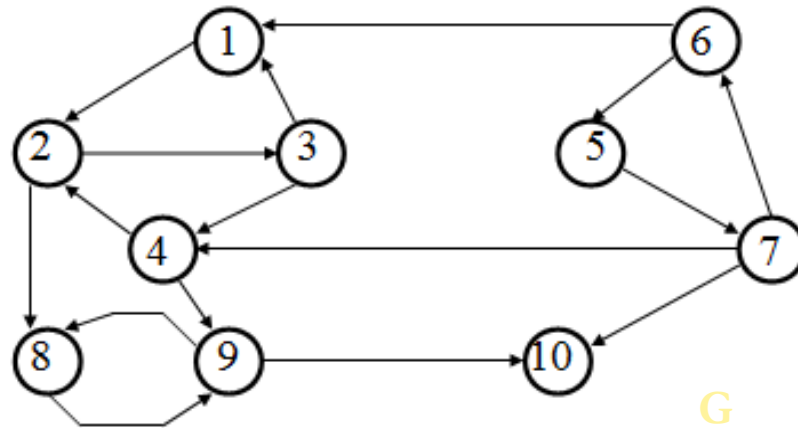
## c) Algoritmo para determinar os CFC's de um digrafo

- Será apresentado o Algoritmo de S.R. Kosaraju.
- Faz duas buscas em profundidade no digrafo
- Dividido em quatro passos
- O algoritmo não será provado, apenas ilustrado. Maiores informações e prova em: *A. V. Aho, J. E. Hopcroft, J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1975*

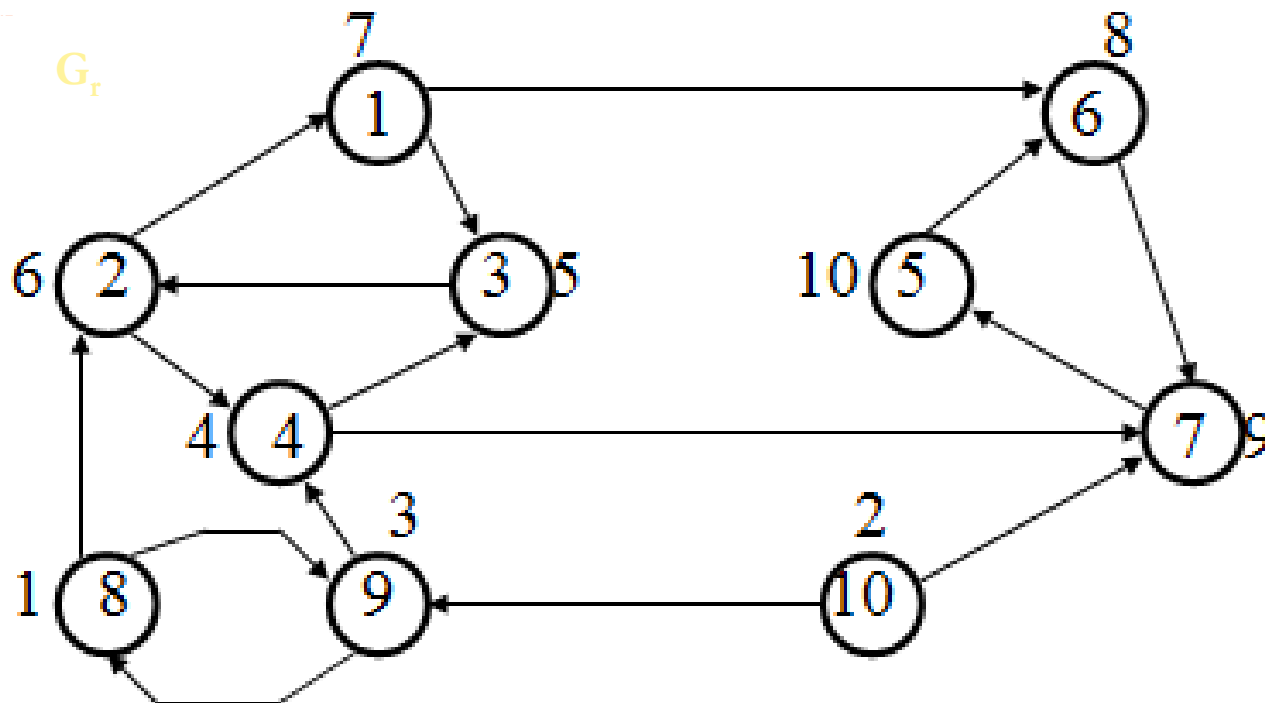
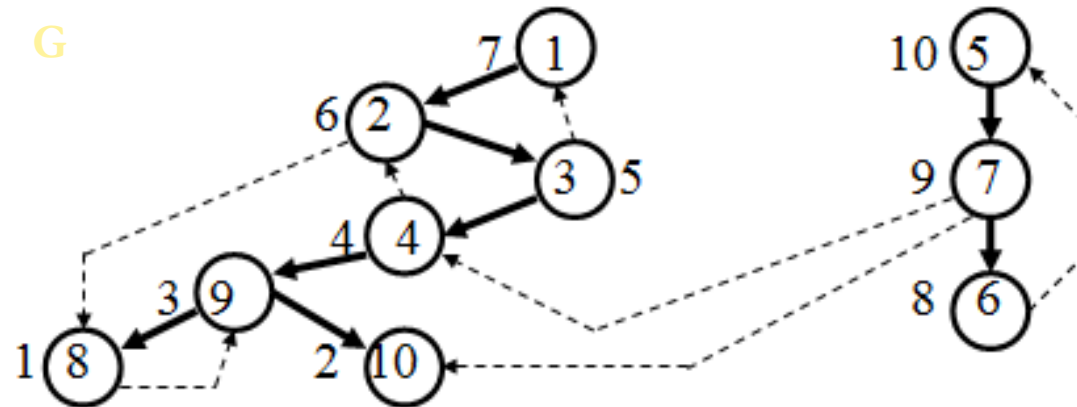


**Passo 1:** Fazer uma busca em profundidade em  $G$ , numerando os vértices no final da chamada recursiva de cada um.

Exemplo:



**Passo 2:** Construir um novo digrafo  $G_r$ , obtido a partir de  $G$ , invertendo-se a direção de todos os seus arcs (grafo reverso).  
Exemplo:



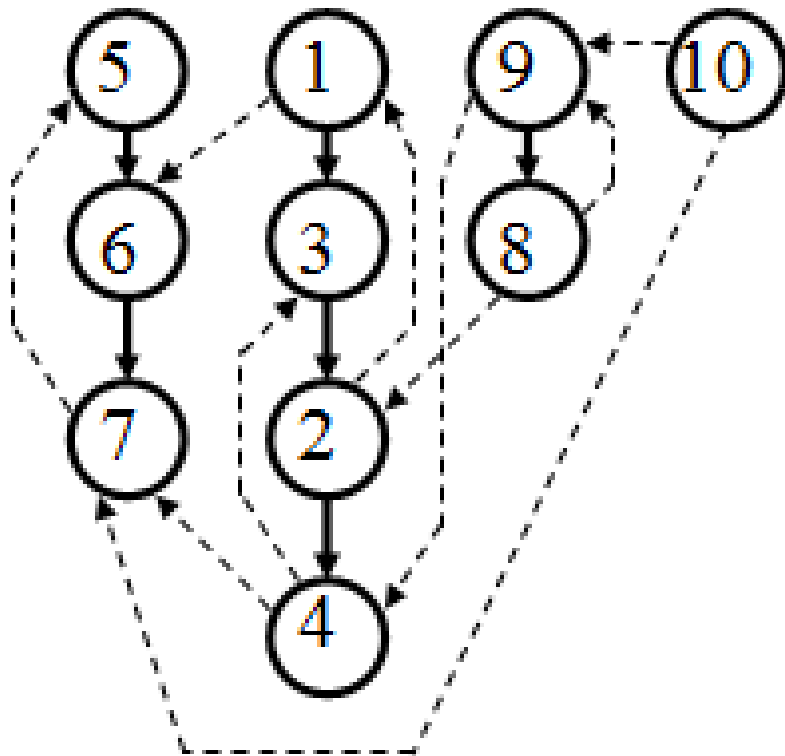
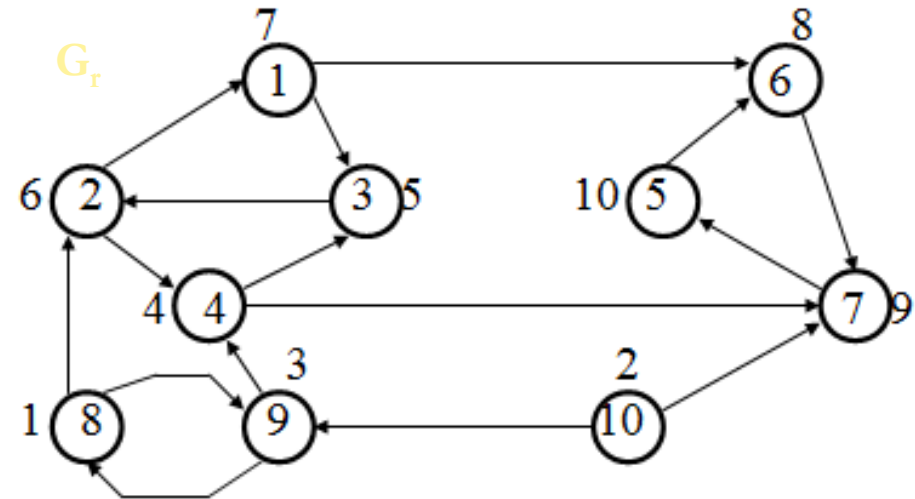
Manter a numeração obtida no passo 1





**Passo 3:** Fazer uma busca em profundidade em  $G_r$ , começando do vértice numerado no passo 1 com o número mais alto.

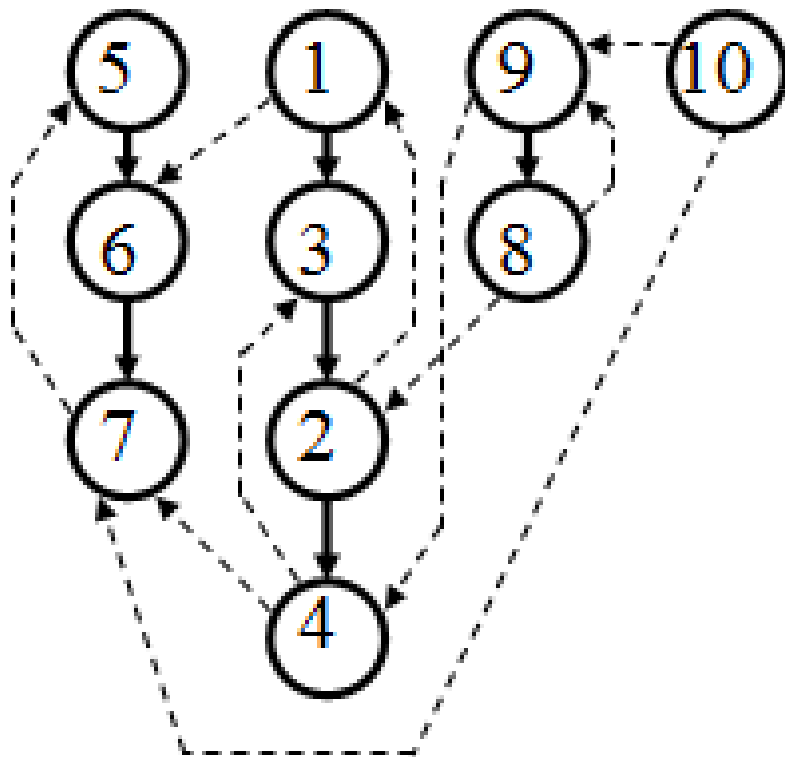
Exemplo:



Se a busca não visitar todos os vértices de  $G_r$ , começar novamente do vértice não visitado de numeração mais alta



**Passo 4:** Cada árvore na floresta da busca em profundidade em  $G_r$ , resultante do Passo 3 contém os vértices de um **CFC** de  $G$ . Exemplo:



Os vértices dos  
CFC's são:

1°: 5, 6, 7

2°: 1, 3, 2, 4

3°: 9, 8

4°: 10

