SIN110 Algoritmos e Grafos

aula 12

Grafos

- Busca em Profundidade/Largura (E08)
- Aplicações Buscas em Largura e Profundidade.

Buscas em Largura e Profundidade

Exercícios E08 - solução

DFS(G)

```
para cada vértice u em G faça
      cor[u] = BRANCO
      pred[u] ← NIL
tempo ← 0
para cada vértice u em G faça
          se cor(u) = BRANCO
                então Visita_DFS(u)
devolve pred[1..n]
```

Visita_DFS(u)

```
 cor[u] ← CINZA
tempo ← tempo + 1
d(u)  ← tempo
para cada v em Adj(u) faça
          se cor[v] = BRANCO
                    então pred[v] ← u
                              Visita_DFS(v)
cor[u] ← PRETO
tempo ← tempo + 1
f(u)  ← tempo
```

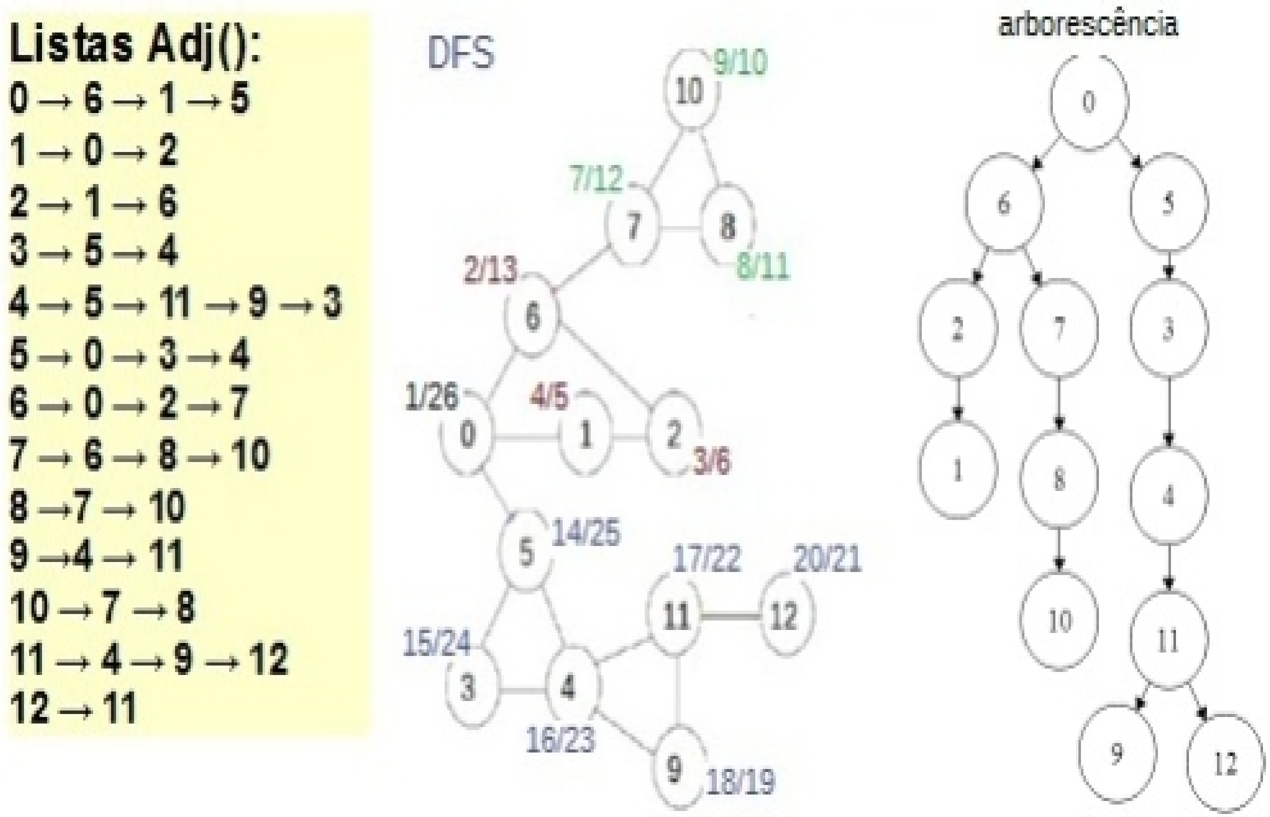
4. Busca em Largura

Arborescência da busca em largura

BFS(G,x)

```
para u ← 1 até n faça
        cor[u] ← BRANCO
        d[u] ← ∞
        pred(u) ← NIL
cor[x] ← CINZA
d[x] = 0
Q ← Inicialiaza-Fila(Q,x)
 enquanto Q ≠ ∅ faça
        u ← Primeiro-da-Fila(Q)
        para cada v em Adj[u] faça
                se cor[v] = BRANCO
                    então cor[v] ← CINZA
                          dist[v] ← dist[u]+1
                          pred(v) ← u
                          Insira-na-Fila(Q,v)
        Remova-da-Fila(Q)
        cor[u] ← PRETO
 devolve dist[1..n], pred[1..n]
```

1) Simule a execução da busca em profundidade (DFS) no grafo G_1 definido pelo conjunto de arestas: **0-6 0-1 0-5 1-2 2-6 6-7 7-8 7-10 10-8 5-3 5-4 4-11 4-9 4-3 9-11 11-12**
(Adote a representação por listas de adjacência e insira as arestas, na ordem dada, num grafo inicialmente vazio.) Faça um desenho da arborescência de busca em profundidade do grafo.

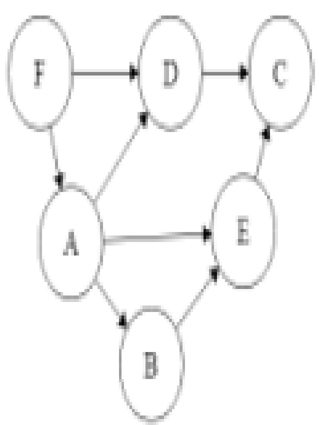


2) A tabela abaixo define o dígrafo G_2 com vértices A, B, C, D, E, F. Suponha que, em cada vértice, a lista de adjacências dos arcos que saem do vértice está em ordem alfabética (a, b, c,...). Lista de arcos:

vértice	F	F	A	B	E	A	D	A	inicio
arco	***a***	***b***	***c***	***d***	***e***	***f***	***g***	***h***	
vértice	A	D	D	E	C	E	C	B	fim

Simule a execução da função de busca em largura (BFS). Em que ordem os vértices serão visitados se executarmos uma *busca em largura* a partir do vértice F? Faça um desenho da arborescência da *busca em largura* a partir de F até o vértice mais distante.

digrafo:



Listas Adj()

A → D → E → B

B → E

C

D → C

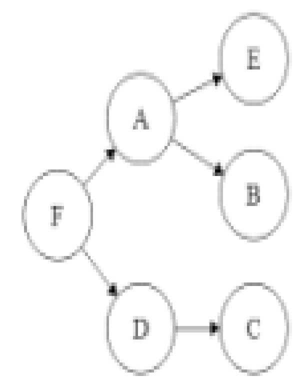
E → C

F → A → D

Sequencia de visitas (preenchimento da fila)

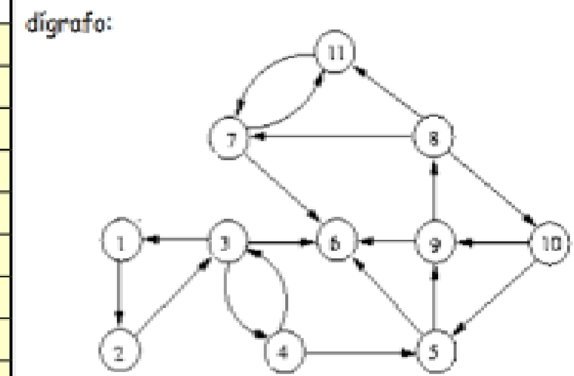
F → A → D → C → E → B

Arborescencia:



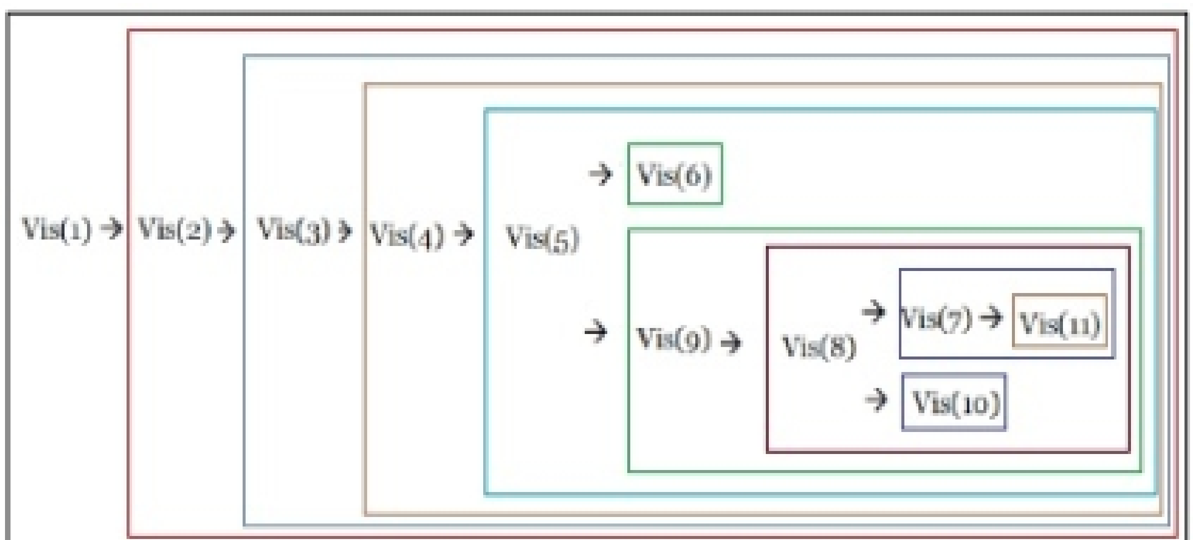
3) Considere o dígrafo G_3 de ordem 11, armazenado na matriz de adjacências abaixo:

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



a) Empregando a busca em profundidade (DFS), demonstre a sequência de visitas, e faça também um desenho da *floresta de busca em profundidade*.

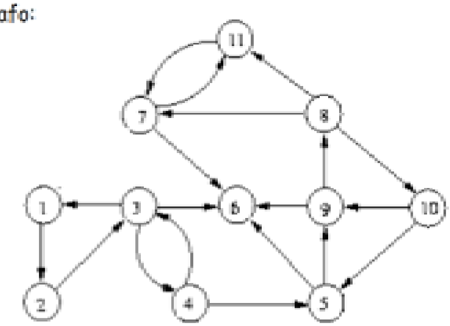
Sequencia de visitas:

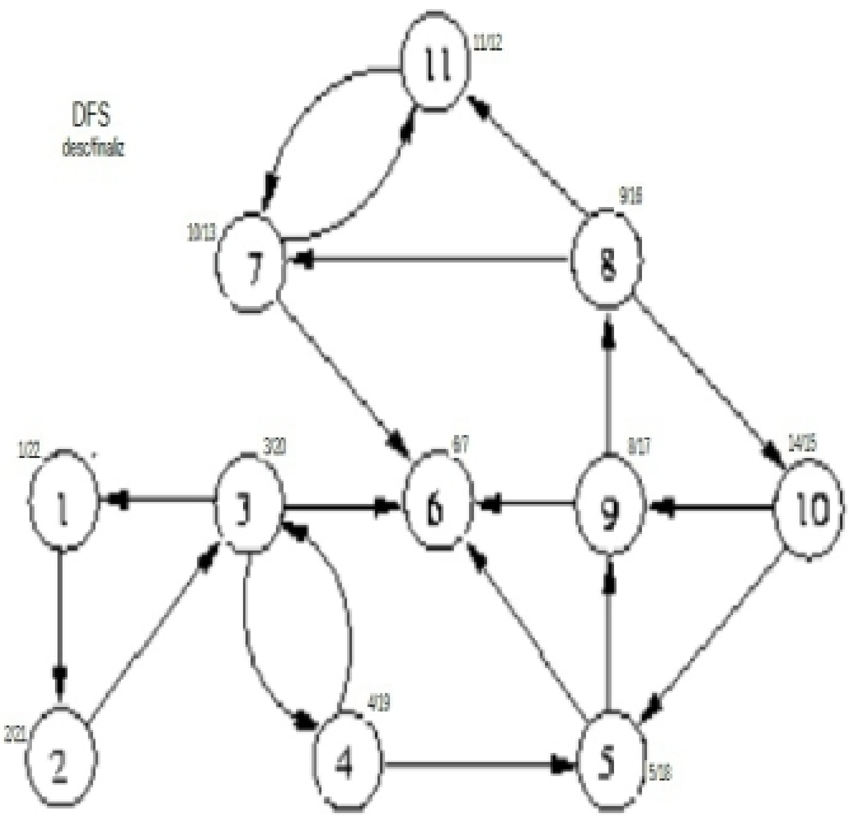


3) Considere o dígrafo G_3 de ordem 11, armazenado na matriz de adjacências abaixo:

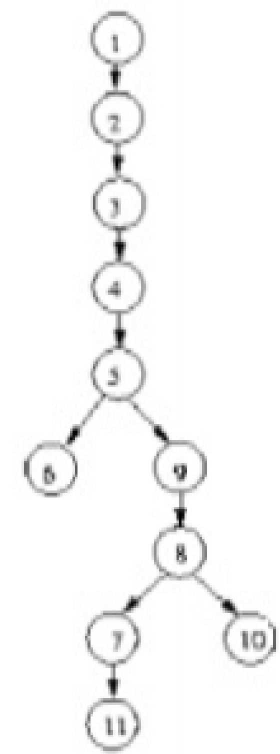
	1	2	3	4	5	6	7	8	9	10	11
1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0
3	1	0	0	1	0	1	0	0	0	0	0
4	0	0	1	0	1	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	1	0	0
6	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0	0	0	0	1
8	0	0	0	0	0	0	1	0	0	1	1
9	0	0	0	0	0	1	0	1	0	0	0
10	0	0	0	0	1	0	0	0	1	0	0
11	0	0	0	0	0	0	1	0	0	0	0

dígrafo:





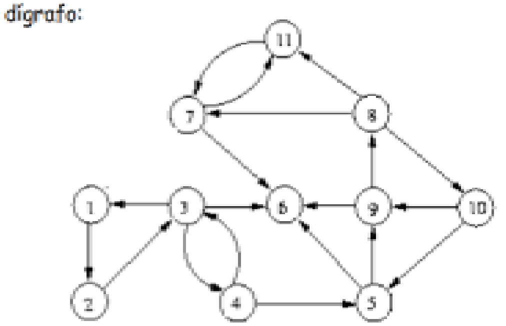
Arborescência:



3) Considere o dígrafo G_3 de ordem 11, armazenado na matriz de adjacências abaixo:

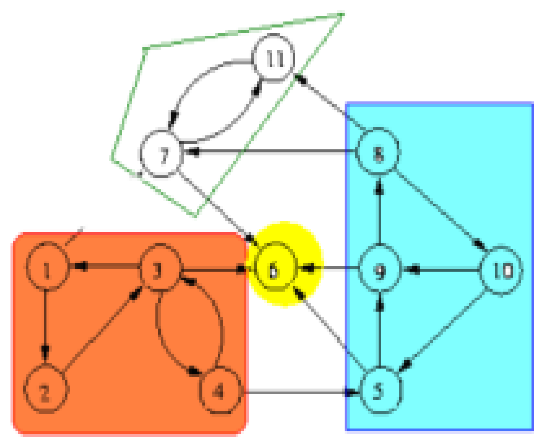
	1	2	3	4	5	6	7	8	9	10	11
1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0
3	1	0	0	1	0	1	0	0	0	0	0
4	0	0	1	0	1	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	1	0	0
6	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0	0	0	0	1
8	0	0	0	0	0	0	1	0	0	1	1
9	0	0	0	0	0	1	0	1	0	0	0
10	0	0	0	0	1	0	0	0	1	0	0
11	0	0	0	0	0	0	1	0	0	0	0

dígrafo:



b) Determine o número de componentes e, demonstre quais vértices pertencem a cada componente. Tem 1 componente ... a arborescência contêm todos os vértices.

c) Existem ciclos nesse dígrafo? Mostre os ciclos.



Ciclos:
1 - 2 - 3 - 1
3 - 4 - 3
7 - 11 - 7
5 - 9 - 8 - 10 - 5
8 - 10 - 9 - 8

4) Considere o grafo G_4 de ordem 11, armazenado na matriz de adjacências abaixo:

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

a) Empregando a busca em largura(BFS), demonstre a sequência de visitas a partir do vértice 1 e, faça também um desenho da *floresta da busca em largura* e mostrando as distâncias percorridas.

Sequencia de visitas: 1 → 2 → 3 → 4 → 6 → 5 → 7 → 10 → 8 → 11 → 9

```
                 2
               ↗
floresta:  1          4 → 5 → 10        ( só tem uma arborescência)
               ↘    ↗
                 3            8 → 9
                   ↘        ↗
                     6 → 7                 [distâncias = número de arcos a partir de 1]
                            ↘
                              11
```

b) Determine o número de componentes e, demonstre quais vértices pertencem a cada componente. Como só tem uma arborescência = 1 componente.

Aplicações

Busca em largura (BFS)

e

Busca em profundidade(DFS)

Aplicação busca em largura

4. Busca em Largura

Caminhos mínimos

Dados um vértice x de um grafo, encontrar um caminho de comprimento mínimo de " x " a cada um dos demais vértices.

Como todos esses caminhos podem ser representados?

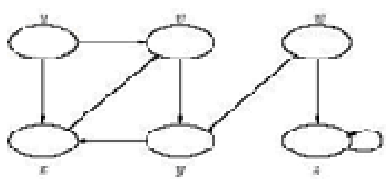
→ *arborescência de caminhos mínimos*

O procedimento imprime os vértices em um caminho mínimo, desde x até v, supondo que BFS já foi executado para calcular a arborescência de caminhos mínimos:

```
Arborescência-BFS(G,x, v)
 1.   se v = x
 2.        então imprime x
 3.        senão se pred(v) = NIL
 4.                  então escreve " não há caminho "
 5.                  senão Arborescência-BFS(G, x, pred(v))
 6.                           escreve "v → "
```

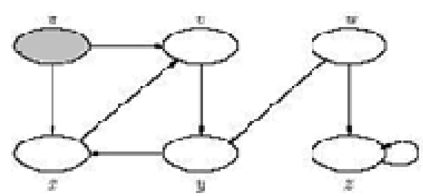
4. Busca em Largura

Exemplo: revendo o andamento do algoritmo BFS em um dígrafo começando a busca por "u"



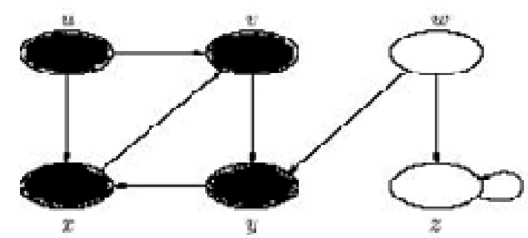
Dígrafo

⇒



(i) Fila: u

⇒

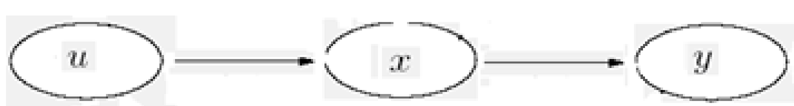


(viii) Fila: ∅

Configuração final dos vetores:

	[u]	[v]	[w]	[x]	[y]	[z]
pred:	-	u	-	u	x	-
dist:	0	1	∞	1	2	∞

Arborescência para o caminho u - y:



Que é impressa: u → x → y

Busca em profundidade
caminhos e componentes

Busca em profundidade

Utilizando busca em profundidade:

- É possível verificar se um grafo é acíclico, conexo, bipartido, planar;
- Pode-se obter as componentes conexas e biconexas de um grafo e seus pontos de articulação;
- Aplicabilidade na solução de diversos problemas.

Caminho entre Vértices

- Um caminho de **comprimento** k de um vértice x a um vértice y em um grafo $G = (V, A)$ é uma seqüência de vértices $(v_0, v_1, v_2, \ldots, v_k)$ tal que $x = v_0$ e $y = v_k$, e $(v_{i-1}, v_i) \in A$ para $i = 1, 2, \ldots, k$.

Caminho entre Vértices

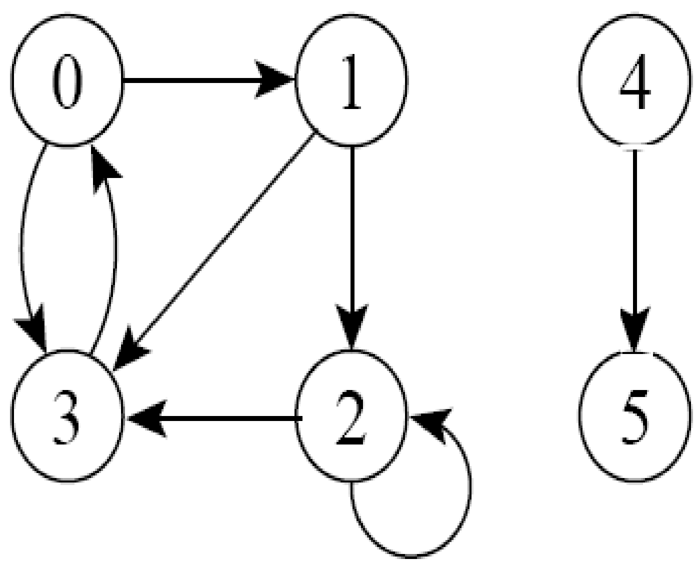
- O comprimento de um caminho é o número de arestas nele, isto é, o caminho contém os vértices $v_0, v_1, v_2, \ldots, v_k$ e as arestas $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.

Caminho entre Vértices

- Se existir um caminho c de x a y então y é **alcançável** a partir de x via c.
- Um caminho é **simples** se todos os vértices do caminho são distintos.

Caminho entre Vértices

Ex.: O caminho $(0, 1, 2, 3)$ é simples e tem comprimento 3. O caminho $(1, 3, 0, 3)$ não é simples.

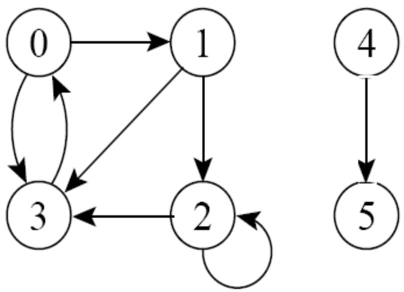


Caminho entre vértices: algoritmo de pesquisa

```
Caminho(G, s,t)
para cada vétice u em G faça
    cor[u] ← BRANCO
devolva VisitaR(G, s, t).

VisitaR (G, u, t)
se u = t
    então devolve 1
cor[u] ← CINZA
para cada v em Adj(u) faça
    se G cor[v] = BRANCO
        então se VisitaR(G,v,t) = 1
                then devolva 1
devolva 0
```

Caminho entre Vértices

Ex.: O caminho $(0, 1, 2, 3)$ é simples e tem comprimento 3. O caminho $(1, 3, 0, 3)$ não é simples.



Verif caminho 3 → 2:

Caminho(G,3,2)

VisitaR(G,3,2)	cor[3]=Cz
VisitaR(G,0,2)	cor[0]=Cz
VisitaR(G,1,2)	cor[1]=Cz
VisitaR(G,2,2)	cor[2]=Cz

Devolve “1”: existe um caminho simples

```
Caminho(G, s,t)
para cada vétice u em G faça
    cor[u] ← BRANCO
devolva VisitaR(G, s, t).

VisitaR (G, u, t)
se u = t
    então devolve 1
cor[u] ← CINZA
para cada v em Adj(u) faça
    se G cor[v] = BRANCO
        então se VisitaR(G,v,t) = 1
                então devolva 1
devolva 0
```

Componentes de grafos

Um conjunto X de vértices é *isolado* se não existe aresta alguma com uma ponta em X e outra fora. Há dois casos degenerados que vale a pena destacar: o conjunto de todos os vértices e o conjunto vazio são sempre isolados.

Um *componente* de um grafo é um conjunto isolado não vazio *mínimo:* digamos que A é o conjunto de *todos* os conjuntos isolados distintos de $\emptyset$.

Um elemento X de A é *mínimo* se não contém algum outro elemento de A, portanto, um componente é um conjunto isolado que não inclui propriamente outro conjunto isolado, exceto o vazio.

Podemos empregar a busca em profundidade para determinar as componentes de um (dí)grafo, basta uma modificação no algoritmo DFS para contar as componentes:

Componentes-DFS(G)

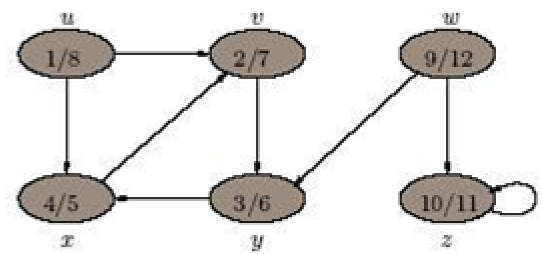
```
para cada vértice u em G faça
        comp[u] = 0
cont ← 0
para cada vértice u em G faça
        se comp[u] = 0
              então  cont ← cont + 1
                     Visita-Componentes-DFS(G, u, cont)
devolve cont
```

Visita-Componentes-DFS(G, u, cont)

```
comp[u] ← cont
para cada v em Adj(u) faça
        se comp[v] = 0
              então  Visita-Componentes-DFS(G, u, cont)
```

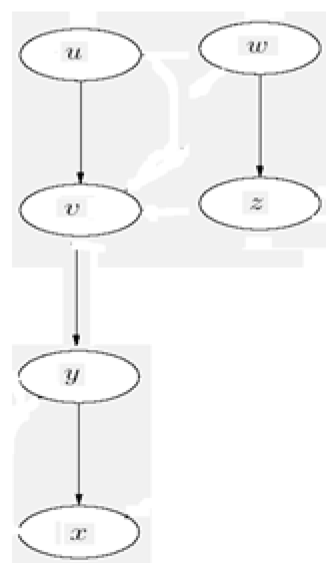
Exemplos: verificando as componentes nos exemplos anteriore

DFS - total



comp:	1	1	2	1	1	2
	[u]	[v]	[w]	[x]	[y]	[z]

Arborescências:

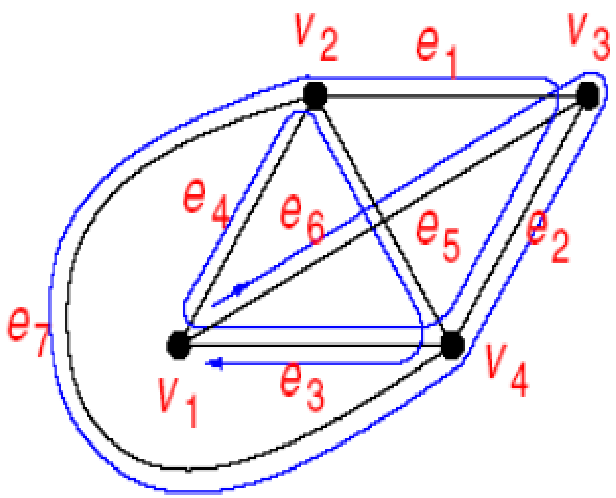


Verificamos 2 componentes (cont = 2)

Ciclos, pontes, articulações

Ciclos

Um ciclo num digrafo é um **caminho fechado** de **comprimento** pelo menos 2, sem vértices repetidos, exceto o último (que coincide com o primeiro).



Um possível caminho fechado é:

$$v_1 e_6 v_3 e_2 v_4 e_7 v_2 e_1 v_3 e_2 v_4 e_3 v_1 e_4 v_2 e_5 v_4 e_3 v_1$$

Um caminho fechado com pelo menos uma aresta é chamado de **ciclo**.

Pesquisa de Ciclos

O algoritmo DFS pode ser usado para esta tarefa:

verificar se o (dí)grafo é acíclico ou contém um ou mais ciclos.

Se um arco reverso é encontrado durante a busca de profundidade, então o dígrafo tem ciclo.

Se um grafo não tem sorvedouro então certamente tem algum ciclo (embora a recíproca não seja verdadeira).

Algoritmos para pesquisar ciclos:

1 – pesquisa em dígrafos representados por listas de adjacências

```
Pesquisa_Ciclos-dígrafos(G)
para cada vértice u em G faça
        para cada v em Adj(u) faça
                ciclo ← Caminho(G, v, u)
                se ciclo = 1
                        então devolve 1
devolve 0
```

Empregamos a função Caminho(G, x, y), apresentada na seção anterior, para verificar a existência de caminhos (0 = não, 1 = sim). A indicação da existência de ciclos também será sinalizada por 0 ou 1.

2 – pesquisa em grafos representados por listas de adjacência:

```
Pesquisa_Ciclos-grafos(G)
para cada vértice u em G faça
        para cada v em Adj(u) faça
                se u < v
                        então  Remove-aresta(G, v, u)
                               ciclo ← Caminho(G, u, v)
                               Inclui-aresta(G, v, u)
                               se ciclo = 1
                                       então devolve 1
devolve 0
```

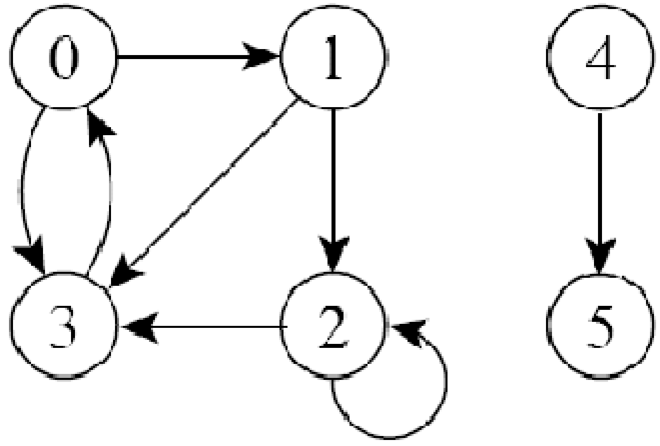
Essa versão para grafos acrescenta a remoção e a re-inclusão de uma aresta *v-w* para verificar se existe um ciclo não trivial entre dois, empregando a mesma sinalização de existência ou não.

Exemplo: aplicando a pesquisa ao dígrafo:

```
Pesquisa_Ciclos-dígrafos(G)
para cada vértice u em G faça
        para cada v em Adj(u) faça
              ciclo ← Caminho(G, v, u)
              se ciclo = 1
                    então devolve 1
devolve 0
```

```
Caminho(G, s,t)
para cada vétice u em G faça
   cor[u] ← BRANCO
devolva VisitaR(G, s, t).
```

```
VisitaR (G, u, t)
se u = t
   então devolve 1
cor[u] ← CINZA
para cada v em Adj(u) faça
     se G cor[v] = BRANCO
        então se VisitaR(G,v,t) = 1
                 então devolva 1
devolva 0
```



Pesquisa_Ciclos-dígrafos(G)

u ← 0

ciclo ← Caminho(G,1,0) — cor[0,1,2,3,4,5]=Cz

VisitaR(G,1,0) — cor[1]=Cz

VisitaR(G,2,0) — cor[2]=Cz

VisitaR(G,3,0) — cor[3]=Cz

VisitaR(G,0,0) — (devolve 1)

ciclo = 1 aponta “tem ciclo = 1)

1 - pesquisa mais eficiente em dígrafos representados por listas de adjacências

Busca_Ciclos-dígrafos(G)

```
para cada vértice u em G faça
    ord(u) ← 0
tempo ← 0
para cada vértice u em G faça
    se ord(u) = 0
        então se Visita-d(G,u) = 1
                  então devolve 1
devolve 0.
```

Visita-d(G,u)

```
tempo ← tempo + 1
ord(u) ← tempo
para cada v em Adj(u) faça
    se ord[v] = 0
        então se Visita-d(G,v) = 1
                  então devolve 1
        senão se ord(v) < ord(u)
                  então devolve 1
devolve 0.
```

A indicação de ciclos será sinalizada por: 0 = dígrafo acíclico ou 1 = existe um ou mais ciclos.

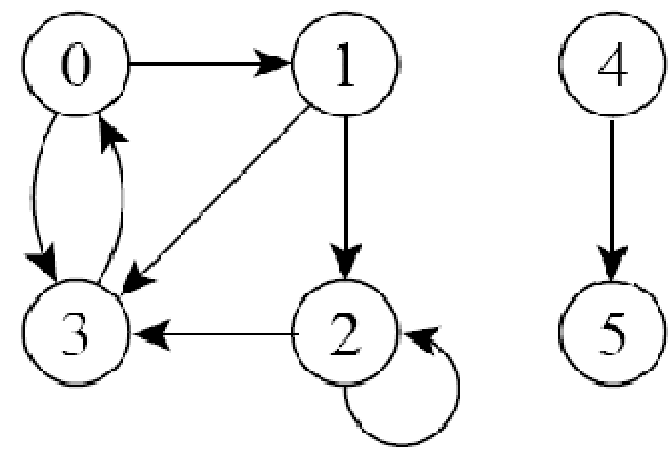
2 - pesquisa eficiente de ciclos não - triviais em grafos representados por listas de adjacência:

```
Busca_Ciclos-grafos(G)
para cada vértice u em G faça
    ord(u) ← 0
tempo ← 0
para cada vértice u em G faça
    se ord(u) = 0
        então pred(u) ← u
              se Visita-g(G,u) = 1
                  então devolve 1
devolve 0.

Visita-g(G,u)
tempo ← tempo + 1
ord(u) ← tempo
para cada v em Adj(u) faça
    se ord[v] = 0
        então pred(v) ← u
              se Visita-d(G,v) = 1
                  então devolve 1
        senão se ord(v) < ord(u)  E  v ≠  pred(u)
                  então devolve 1
devolve 0.
```

para evitar os ciclos triviais é utilizado o vetor de predecessores (pred[ ])

Exemplo: aplicando a pesquisa ao dígrafo:

```
Busca_Ciclos-dígrafos(G)
    para cada vértice u em G faça
          ord(u) ← 0
    tempo ← 0
    para cada vértice u em G faça
           se ord(u) = 0
              então  se Visita-d(G,u) = 1
                          então devolve 1
    devolve 0.

Visita-d(G,u)
    tempo ← tempo + 1
    ord(u)  ← tempo
    para cada v em Adj(u) faça
           se ord[v] = 0
              então se Visita-d(G,v) = 1
                         então devolve 1
              senão se ord(v) < ord(u)
                         então devolve 1
    devolve 0.
```



Busca_Ciclos-dígrafos(G)

ord[0, 1, 2, 3, 4, 5] ← 0   tempo = 0

u ← 0, Visita-d(G,0)   ord[0]=1

v←1   *Visita-d(G,1)*   ord[1]=2

v←2   Visita-d(G,2)   ord[2]=3

v←3   Visita-d(G,3)   ord[3]=4

v←0   ord[0]<ord[3]   **devolve 1**

Visita-d(G,0)=1 aponta que “tem ciclo”

Grafos bipartidos e ciclos ímpares

Um grafo é bipartido se existe uma bipartição do seu conjunto de vértices tal que cada aresta tem uma ponta em uma das partes da bipartição e a outra ponta na outra parte.

Outra maneira de formular : *um grafo é bipartido se for possível atribuir uma de duas cores a cada vértice de tal forma que as pontas de cada aresta tenham cores diferentes.*

grafos que têm ciclos de comprimento ímpar não são bipartidos.

É difícil provar que todo grafo "sem ciclos ímpares" admite bipartição.

O algoritmo devolve 1 se o grafo é bipartido e devolve 0 em caso contrário. Além disso, se o grafo é bipartido, a função atribui uma "cor" a cada vértice, cores dos vértices, representadas por 0 e 1, são registradas no vetor cor[ ]

Pesq_Bipartição(G)

```
para cada vértice u em G faça
      cor(u) ← -1
ref ← 0
para cada vértice u em G faça
         se cor(u) = -1
              então  se VisitaB-g(G, u, 0) = 0
                               então devolve 0
devolve 1.
```

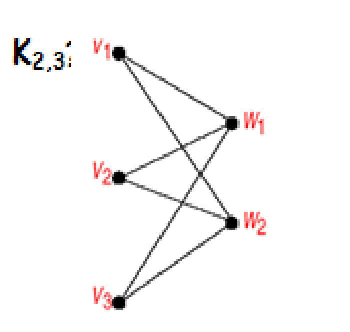
VisitaB-g(G, u, ref)

```
cor(u)  ← 1 - ref
para cada v em Adj(u) faça
         se cor[v] = -1
              então se VisitaB-d(G, v, 1-ref) = 0
                             então devolve 0
              senão se cor(v) = (1 – ref)
                             então devolve 0
devolve 1.
```

Exemplo: aplicando a pesquisa bipartição

```
Pesq_Bipartição(G)
    para cada vértice u em G faça
        cor(u) ← -1
    ref ← 0
    para cada vértice u em G faça
        se cor(u) = -1
            então  se VisitaB-g(G, u, 0) = 0
                       então devolve 0
    devolve 1.
```

```
VisitaB-g(G, u, ref)
    cor(u)  ← 1 - ref
    para cada v em Adj(u) faça
        se cor[v] = -1
            então se VisitaB-d(G, v, 1-ref) = 0
                      então devolve 0
            senão se cor(v) = (1 – ref)
                      então devolve 0
    devolve 1.
```



	V_1	V_2	V_3	W_1	W_2
V_1	-	-	-	1	1
V_2	-	-	-	1	1
V_3	-	-	-	1	1
W_1	1	1	1	-	-
W_2	1	1	1	-	-

Pesq_Bipartição(G)

cor[V1, V2, V3, W1, W3] ← -1   ref ← 0

u ← V1, VisitaB-g(G,V1, 0) cor[V1] ←1-0

VisitaB-g(G,W1,1-0) cor[W1] ←1-1

VisitaB-g(G,V2,1-1) cor[V2] ←1-0

VisitaB-g(G,W2,1-0) cor[W2] ←1-1

VisitaB-g(G,V3,1-1) cor[V3] ← 1-0

Pesq_Bipartição(G) = 1 aponta a “bipartição”

Pontes e articulações em grafos

grafos que deixam de ser conexos quando perdem uma de suas arestas

Trataremos apenas de grafos, pois os conceitos a serem discutidos não fazem sentido em dígrafos não-simétricos.

Pontes e aresta-biconexão

Uma aresta é uma ponte se ela é a única que atravessa algum corte do grafo.

ponte é uma aresta cuja remoção aumenta o número de componentes do grafo.

Problema a tratar: "Encontrar as pontes de um grafo dado."

Um grafo é aresta-biconexo se for conexo e não tiver pontes, portanto, é preciso remover pelo menos duas arestas que ele deixe de ser conexo.

Articulações e biconexão

articulação ou vértice de corte é um vértice cuja remoção aumenta o número de componentes.

Um grafo é biconexo se é conexo e não tem articulações, é preciso remover pelo menos 2 vértices para que deixe de ser conexo.

Pesquisa de Pontes

adaptação do algoritmo de busca em profundidade

ponto de partida em qualquer grafo,

uma aresta é uma ponte s se não faz parte de um ciclo não - trivial.

Ou seja, toda aresta ou é uma ponte ou pertence a um ciclo não - trivial.

Propriedade: em qualquer busca em profundidade, um arco *v-w* da floresta da Busca em Profundidade faz parte (juntamente com *w-v*) de uma ponte se e somente se não existe arco de retorno que ligue um descendente de *w* a um ancestral de *v*.

A propriedade pode ser assim reformulada: *em qualquer floresta de busca em profundidade de um grafo, um arco de arborescência x-y faz parte de uma ponte se e somente se pré-ord[y] = ord[y].*

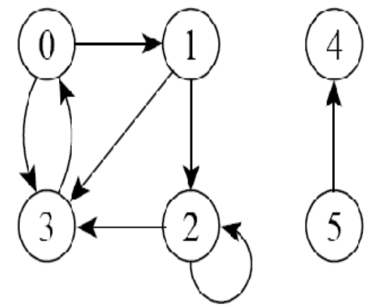
Algoritmo para pesquisar pontes

```
Busca_Pontes(G)
 1.  para cada vértice u em G faça
 2.     ord(u) ← 0
 3.  tempo ← conta ← 0
 4.  para cada vértice u em G faça
 5.          se ord(u) = 0
 6.              então  pred(u) ← u
 7.                     tempo ← tempo + 1
 8.                     VisitaP-g(G, u, tempo, conta).

VisitaP-g(G,u, tempo, conta)
 1.  pré-ord ← ord(u)  ← tempo
 2.  para cada v em Adj(u) faça
 3.          se ord[v] = 0
 4.              então pred(v) ← u
 5.                     tempo ← tempo + 1
 6.                     VisitaP-g(G, v, tempo, conta-ponte)
 7.                     se ord(u) > ord(v)
 8.                          então ord(u) ← ord(v)
 9.                     se pré-ord(v) = ord(v)
10.                          então conta ← conta + 1
11.                                   escreve: "ponte", conta, u"–"v
12.              senão se v ≠ pred(u)  E  pré-ord(u) > ord(v)
13.                          então pré-ord(u) ← ord(v)
```

teste da linha 12, " v ≠ pred(u)", garante que *u-v* é um arco de retorno (e não um arco - pai).

Exemplo: aplicando a pesquisa de pontes ao dígrafo:

```
Busca_Pontes(G)
para cada vértice u em G faça
  ord(u) ← 0
tempo ← conta-ponte ← 0
para cada vértice u em G faça
      se ord(u) = 0
         então  pred(u) ← u
                tempo ← tempo + 1
                VisitaP-g(G, u, tempo, conta-ponte).

VisitaP-g(G,u)
pré-ord ← ord(u)  ← tempo
para cada v em Adj(u) faça
      se ord[v] = 0
         então pred(v) ← u
               tempo ← tempo + 1
               VisitaP-g(G, v, tempo, conta-ponte)
               se ord(u) > ord(v)
                   então ord(u) ← ord(v)
               se pré-ord(v) = ord(v)
                   então conta-ponte ← conta-ponte + 1
                         escreve: "ponte - conta-ponte: u – v"
         senão se v ≠ pred(u)  E  pré-ord(u) > ord(v)
                   então pré-ord(u) ← ord(v)
```



Adj(u)

$0 \rightarrow 1 \rightarrow 2$

$1 \rightarrow 0 \rightarrow 3$

$2 \rightarrow 0 \rightarrow 4$

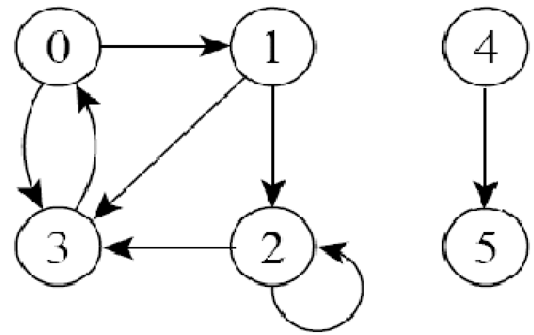
$3 \rightarrow 1 \rightarrow 4$

$4 \rightarrow 2 \rightarrow 3$

Exemplo: aplicando a pesquisa de pontes ao dígrafo:

```
Busca_Pontes(G)
para cada vértice u em G faça
   ord(u) ← 0
tempo ← conta ← 0
para cada vértice u em G faça
     se ord(u) = 0
         então  pred(u) ← u
                tempo ← tempo + 1
                VisitaP-g(G, u, tempo, conta).

VisitaP-g(G,u, tempo, conta)
pré-ord ← ord(u)  ← tempo
para cada v em Adj(u) faça
     se ord[v] = 0
         então pred(v) ← u
               tempo ← tempo + 1
               VisitaP-g(G, v, tempo, conta-ponte)
               se ord(u) > ord(v)
                   então ord(u) ← ord(v)
               se pré-ord(v) = ord(v)
                   então conta ← conta + 1
                         escreve: "ponte", conta, u"-"v
         senão se v ≠ pred(u)  E  pré-ord(u) > ord(v)
                   então pré-ord(u) ← ord(v)
```

Adj(u)
$0 \rightarrow 1 \rightarrow 3$
$1 \rightarrow 2 \rightarrow 3$
$2 \rightarrow 2 \rightarrow 3$
$3 \rightarrow 0$
$4 \rightarrow 5$
5

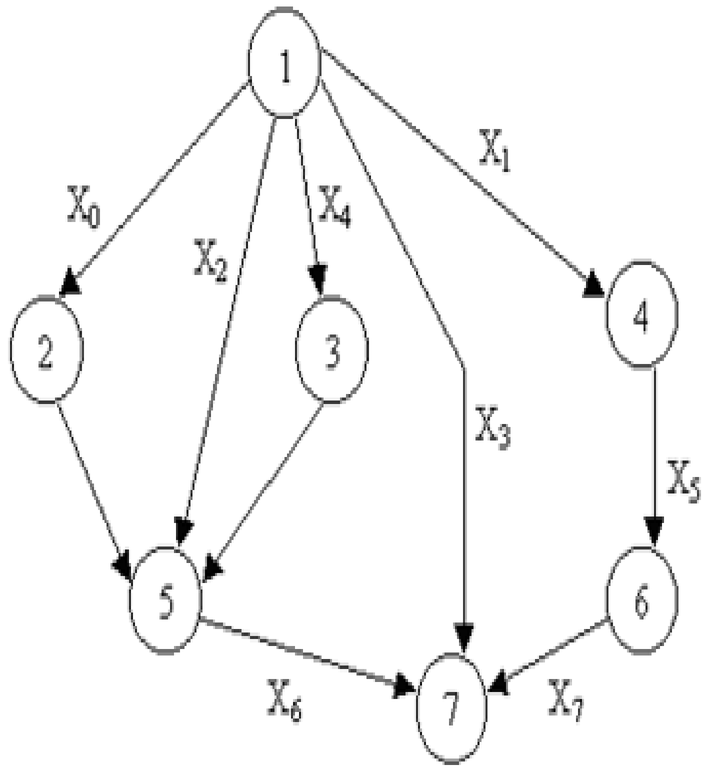


Dígrafo tem 2 componentes …
… e pontes nos arcos 0 -1, 1 -2 e 4-5

Busca_Pontes(G)
ord[0, 1, 2, 3, 4, 5] ← 0   tempo = conta ← 0
u ← 0, tempo = 1 pred[0]=0   VisitaP-g(G,0,1,0)   pre-ord[0] = ord[0]=1
v←1 pred[1]=0 tempo=2 VisitaP-g(G,1,2,0)   pre-ord[1] = ord[1]=2
v←2 pred[2]=1 tempo=3 VisitaP-g(G,2,3,0)   pre-ord[2] = ord[2]=3
v←2 2≠pred[2] E pred-ord[2] = ord[2] … nenhuma ação
v←3 pred[3]=2 tempo=4 VisitaP-g(G,3,4,0)   pre-ord[3] = ord[3]=4
v←0 ..ord[0]≠0 ... 0≠pred[3] E pre-ord[3] > ord[0] então: pre-ord[3] ← ord[0] = 1
………………… ord[2] < ord[3] não altera … pre-ord[3] < ord[3] não altera.
ord[1]<ord[2] não altera … pre-ord[2]=ord[2] … então ESCREVE "ponte1 1-2"
ord[0]< ord[1] não altera … pred-ord[0]=ord[0] … então ESCREVE "ponte2 0-1"
u ← 4, tempo = 5 pred[4]=4 **VisitaP-g(G, 4, 5, 0)** pre-ord[4] = ord[4]=5
v←5 pred[5]=4 tempo=6 VisitaP-g(G,5,6,0)   pre-ord[5] = ord[5] = 6
ord[4]<ord[5] não altera … pre-ord[4]=ord[4] … então ESCREVE "ponte3 4-5"

ORDENAÇÃO TOPOLÓGICA

Um DAG (direct acyclic graph) é um dígrafo acíclico, ou seja, uma travessia em profundidade no dígrafo G não indica nenhum arco para trás. Exemplo de um DAG:

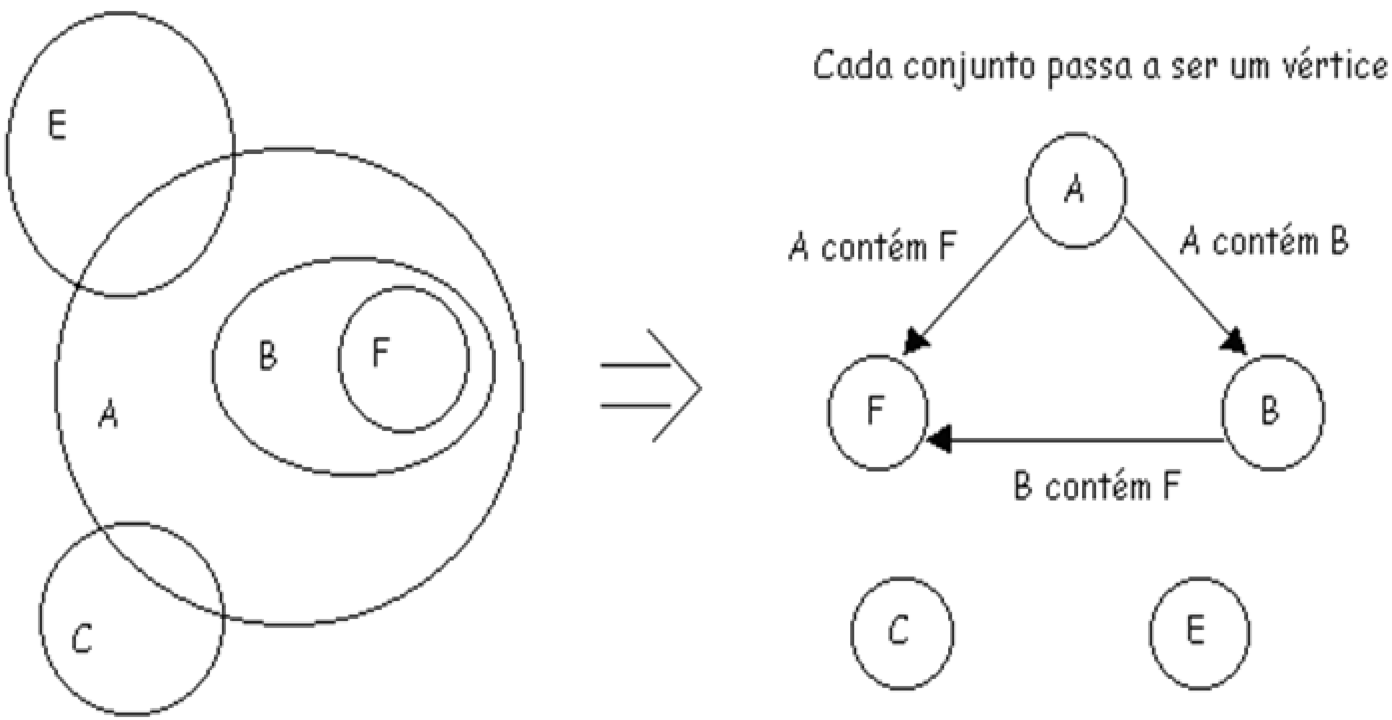


Dígrafo acíclico G

DAGs podem representar: operações de ordem parcial, redes de atividades, compiladores, pré-requisitos, jogos, etc.

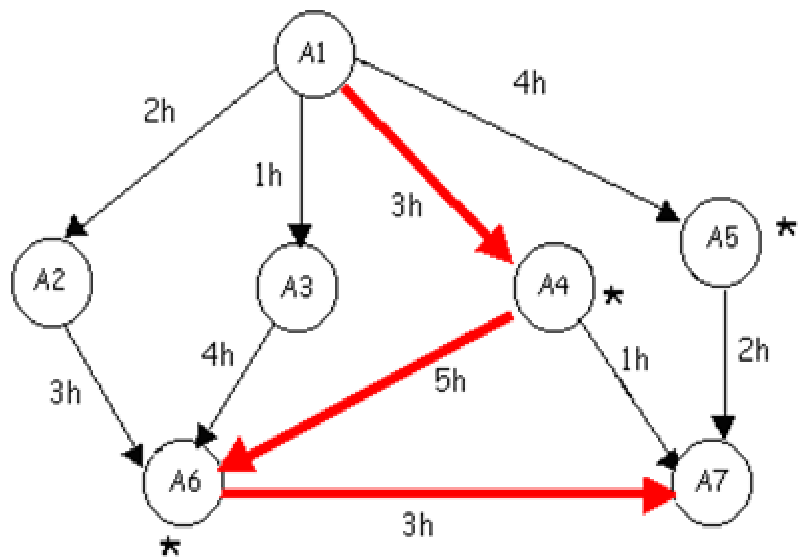
Vejamos alguns exemplos:

- *Operações de Ordem parcial:* exemplo de inclusão:



- *Redes de Atividades*:

- Exemplo: temos uma rede de atividades num processo industrial que pode ser representada na seguinte forma:



Como podemos observar, o peso máximo para chegar à atividade A7 é 11 (caminho a vermelho), ou seja, é a soma dos pesos desde a fonte até à folha, ao qual chamamos de *caminho crítico*.

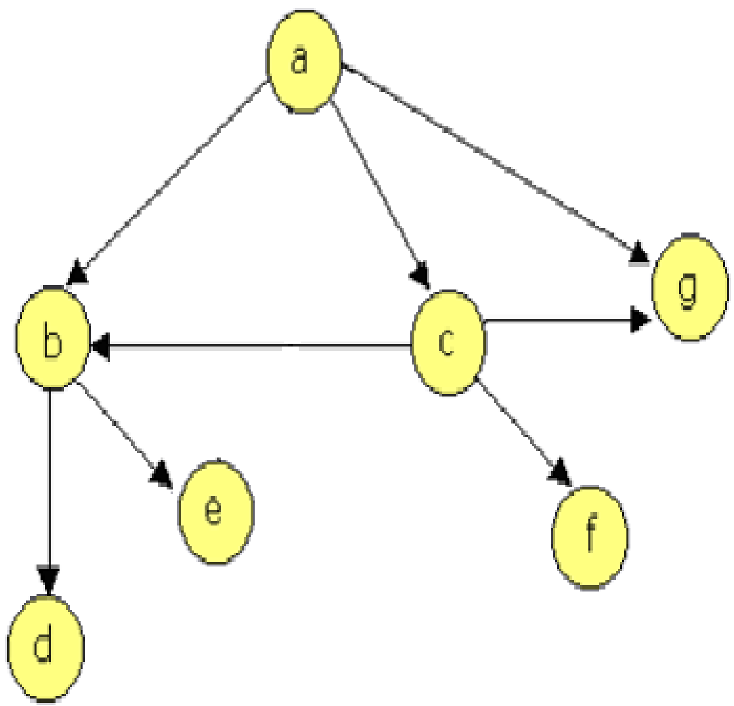
A ordenação topológica consiste em ordenar os vértices de um grafo acíclico G=(V,E) de forma que, se existe um caminho do vértice v para o vértice u, então v aparece antes de u na ordenação. De notar que uma ordenação topológica não é única e que ela não é possível se o grafo possuir ciclos.

Um exemplo comum de problemas de ordenação topológica, é a confecção de dicionários, onde desejamos que uma palavra B cuja definição dependa da palavra A, apareça depois de A no dicionário.

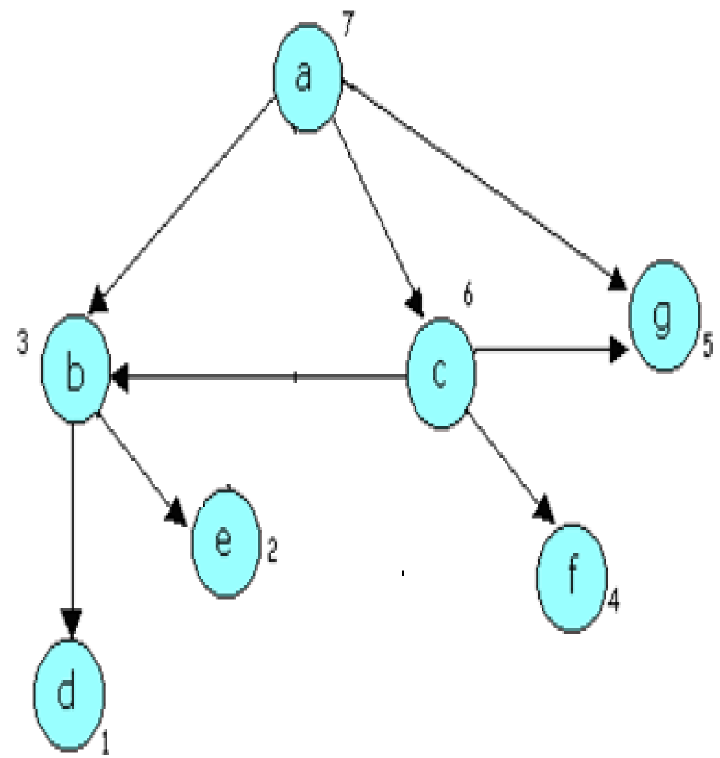
Uma forma simples de resolver esse problema é a utilização de uma versão do algoritmo de busca em profundidade que imprime o vértice antes de retornar a chamada. Obteremos assim os vértices em ordem topológica invertida:

```
OrdTopo(G)
para cada vértice u em G faça
cor[u] ← BRANCO
topo(u) ← -1
 k ← |V|                    (nº vértices de G)
 para cada vértice u em G faça
            se cor(u) = BRANCO
                então Visita(u, topo, k)
 devolve topo[1 .. |V|]


Visita(u, topo, k)
  cor[u] ← PRETO
 para cada v em Adj(u) faça
            se cor[v] = BRANCO
                   então  Visita(v)
 k ← k – 1
 topo(k) ←  u
 imprime u
```

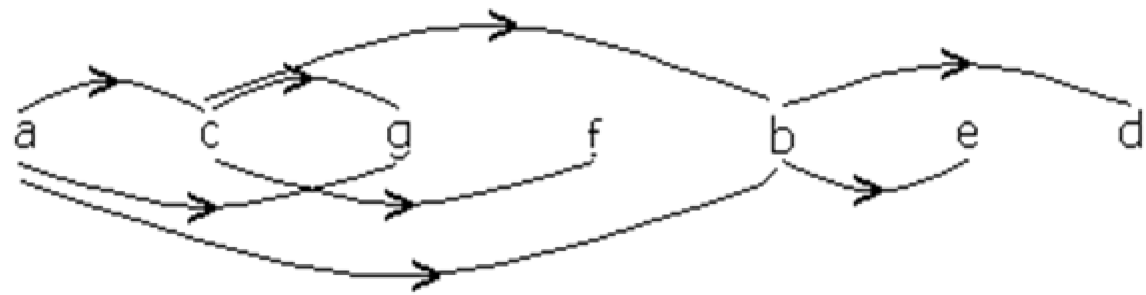
Vejamos um exemplo de ordenação topológica com seguinte DAG:



A execução do algoritmo de busca em profundidade nos dará a ordem de visita de cada vértice, considere que a lista de adjacências deste DAG resulte em:



O objetivo é re - escrever o DAG de forma linear, executando o algoritmo OrdTopo vamos obter os arcos todos na mesma direção:



E o resultado fica registrado no vetor “topo” que apresenta no exemplo o vértice “a” na 1ª posição e o vértice “d” na última.

Há que ter em atenção que a ordenação topológica não é única.

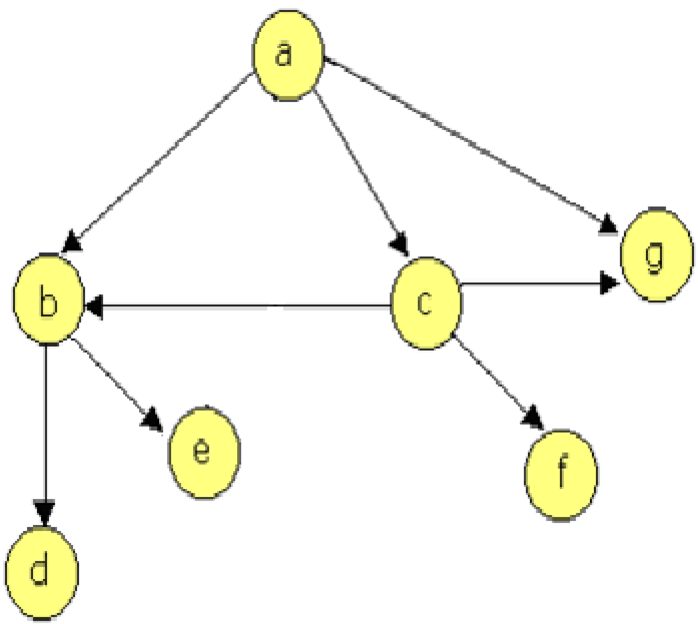
Existe mais de uma forma de escrever um dígrafo por ordem topológica.

No entanto, para realizarmos o algoritmo vamos ter que usar o DFS, sendo o resultado armazenado numa lista.

Outra alternativa para obter uma ordenação topológica é o algoritmo que trabalha com a informação do grau de entrada em cada vértice, denominado algoritmo da eliminação das fontes:

```
OrdTopo-elimfontes(G)
para cada vértice u em G faça
grau-entrada[u] ← 0
para cada vértice u em G faça
        para cada v em Adj(u) faça    (mapeia o grau de entrada de u)
                grau-entrada[v] ← grau-entrada[v] + 1
CriaFila(Q)
para cada vértice u em G faça  (somente fontes são inseridas na fila)
      se grau-entrada = 0
          então InsereFila(Q,u)
i ← 1
enquanto Q ≠ ∅ faça
        u ← TiraFila(Q)
        topo[i] ← u
         i ← i + 1
         para cada v em Adj(u) faça
                grau-entrada[v] ← grau-entrada[v] - 1
                se grau-entrada[v] = 0
                    então  InsereFila(Q,v)
devolve topo[1 .. n]
```

Simulando com o DAG:



i) grau de entrada [linhas 1 - 10]

vert: a b c d e f g

grau: 0 2 1 1 1 1 2

Fila: a

ii) Enquanto … [linhas 11 - 19]

$u \leftarrow a$ vert: **a** b c d e f g

grau: 0 1 0 1 1 1 1 Fila: c

$u \leftarrow c$ vert: **a** b **c** d e f g

grau: **0** 0 **0** 1 1 0 0 Fila: b, f, g

$u \leftarrow b$ vert: **a** **b** **c** d e f g

grau: **0** **0** **0** 0 0 0 0 Fila: f, g, e, d

$u \leftarrow f$ vert: **a** **b** **c** d e **f** g

grau: **0** **0** **0** 0 0 **0** 0 Fila: g, e, d

$u \leftarrow g$ vert: **a** **b** **c** d e **f** **g**

grau: **0** **0** **0** 0 0 **0** **0** Fila: e, d

$u \leftarrow e$ vert: **a** **b** **c** d **e** **f** **g**

grau: **0** **0** **0** 0 **0** **0** **0** Fila: d

$u \leftarrow d$ vert: **a** **b** **c** **d** **e** **f** **g**

grau: **0** **0** **0** **0** **0** **0** **0** Fila: $\emptyset$

Ordenação:

a $\rightarrow$ c $\rightarrow$ b $\rightarrow$ f $\rightarrow$ g $\rightarrow$ e $\rightarrow$ d