GRAFOS E APLICAÇÕES

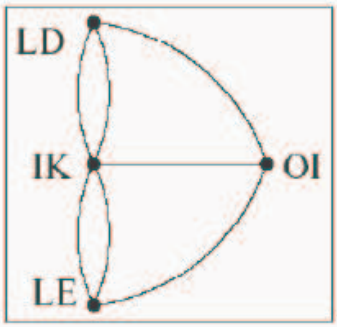
No século XVIII, na Prússia, havia uma controvérsia entre os moradores de Königsberg que chegou aos ouvidos do matemático Leonhard Euler.

Euler descreveu a controvérsia da seguinte forma:

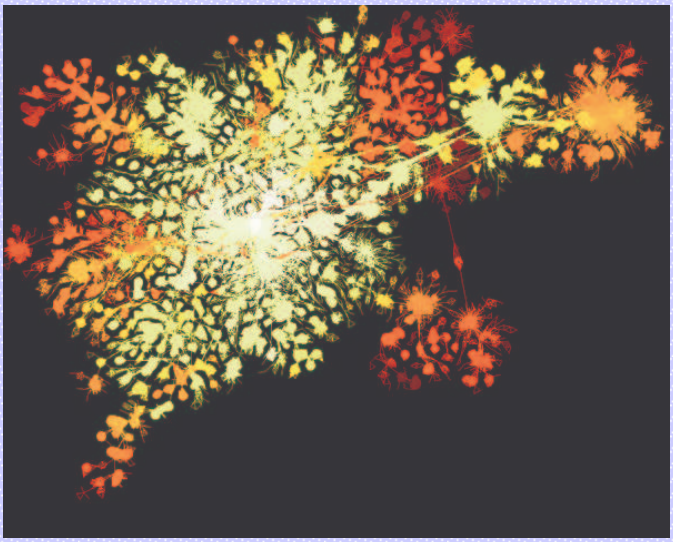
"... Na cidade de Königsberg, na Prússia, há uma ilha chamada Kneiphhof, com os dois braços do rio Pregel fluindo em volta dela. Há 7 pontes – a, b, c, d, e, f e g – cruzando estes dois braços.

...A questão é se uma pessoa pode planejar uma caminhada de modo que ela cruze cada uma destas pontes uma única vez, e não mais que isso. . . "

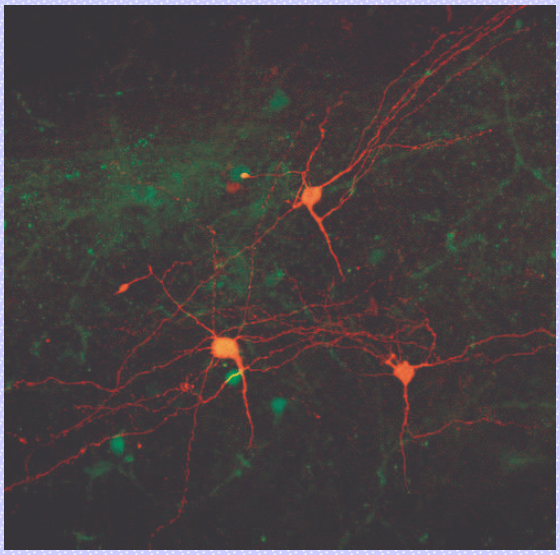
Como representar este problema?



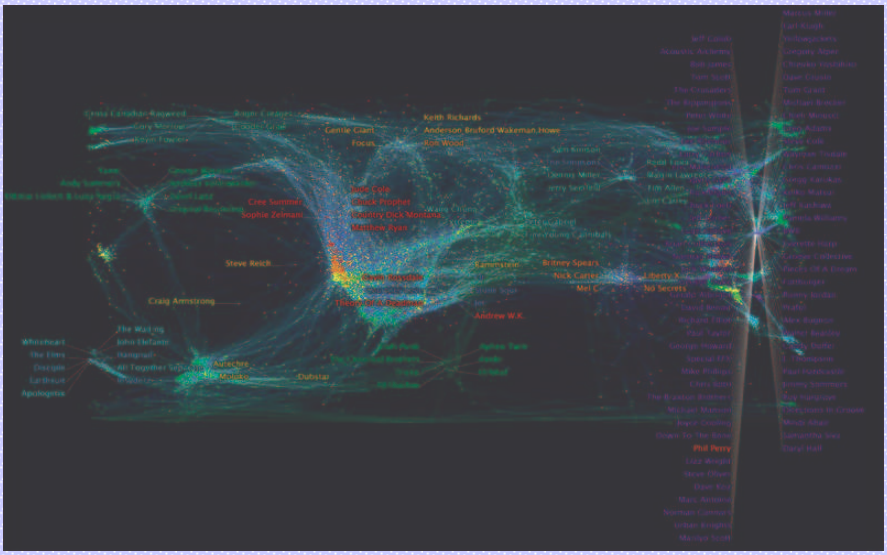
Por que não foi possível fazer tal trajeto?



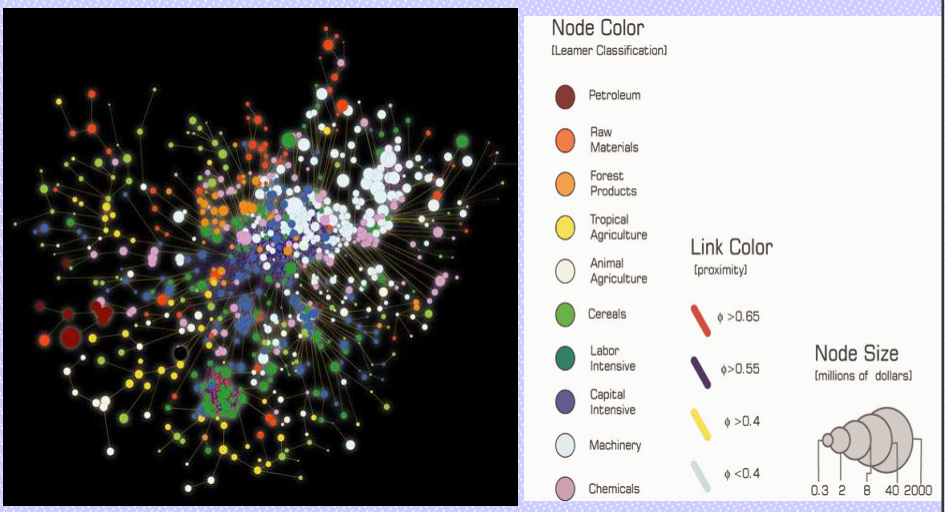
Protein Homology Graph: This network maps protein function by connecting proteins that share sequence similarity. Each of the 30,727 vertices represents a protein, and each of the 1,206,654 connections represents a similarity in amino acid sequence.



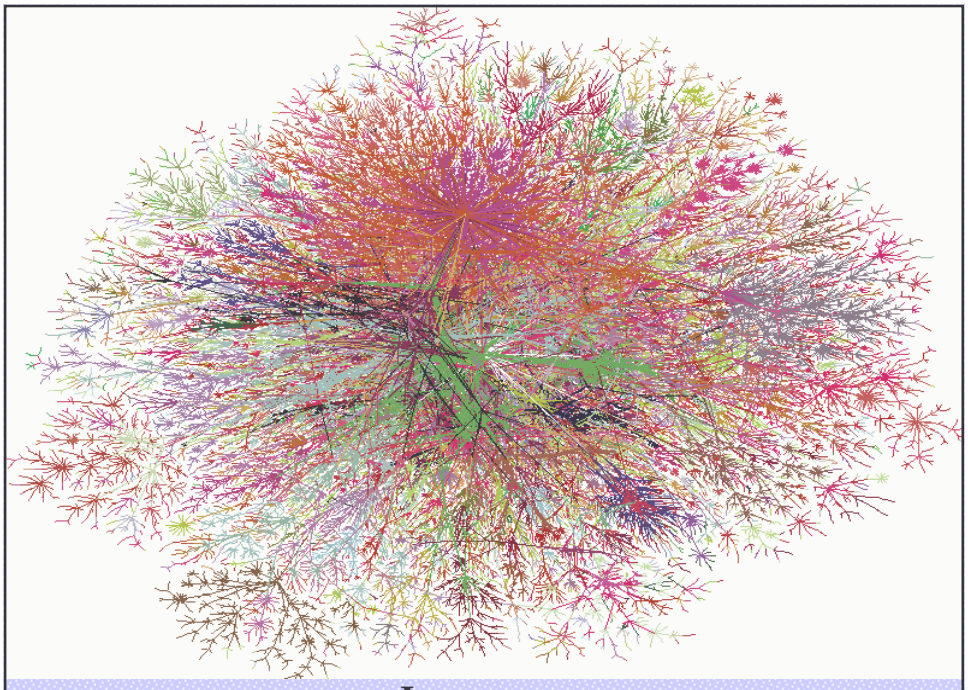
Neuronal Network: Three cortical mouse neurons were imaged at a magnification of 1,000x. The resulting picture shows how neurons join to form a complex network. According to author Mark Miller, a typical cortical neuron receives 1,000 to 10,000 contacts from other neurons and contacts 100 to 1,000 additional neurons.



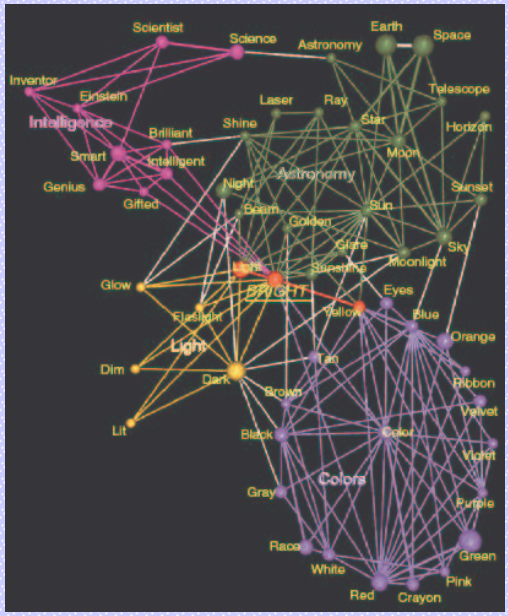
The World of Music: SDP Layout of High Dimensional Data: Researchers used a set of ratings from the LAUNCHcast team at Yahoo! to map artists, connecting those who were rated similarly by a large number of users.



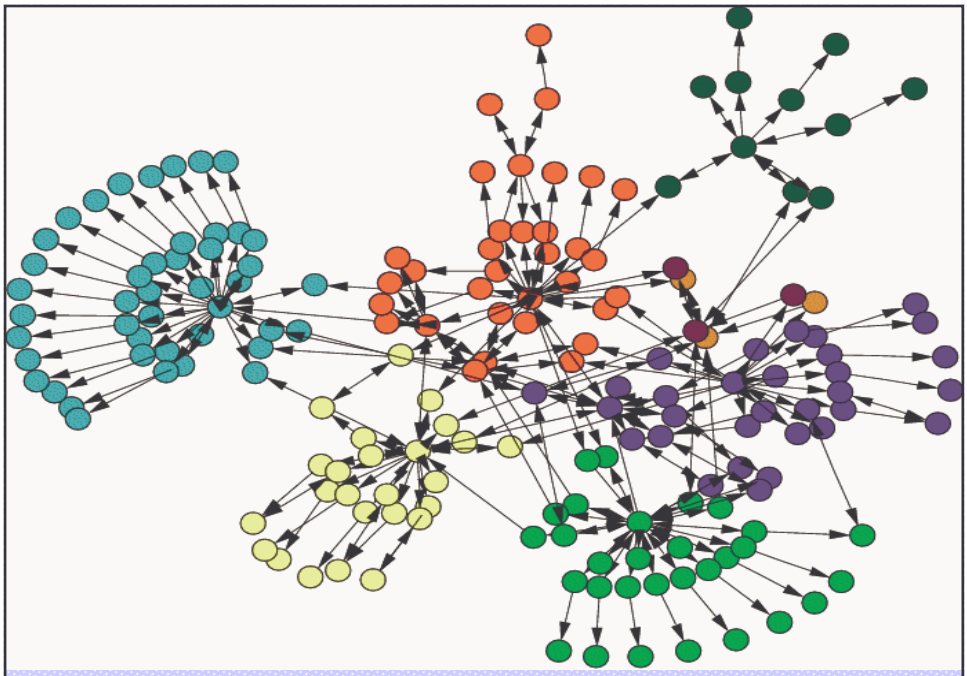
Mapping the World Economy: SDP Layout of High Dimensional Data: Research partners include the Center for Complex Network Research (CCNR) at the University of Notre Dame and the Center for International Development (CID) at Harvard Kennedy School. Collaborators include C.A. Hidalgo, B. Klinger, A-L Barabasi, and R. Hausmann.



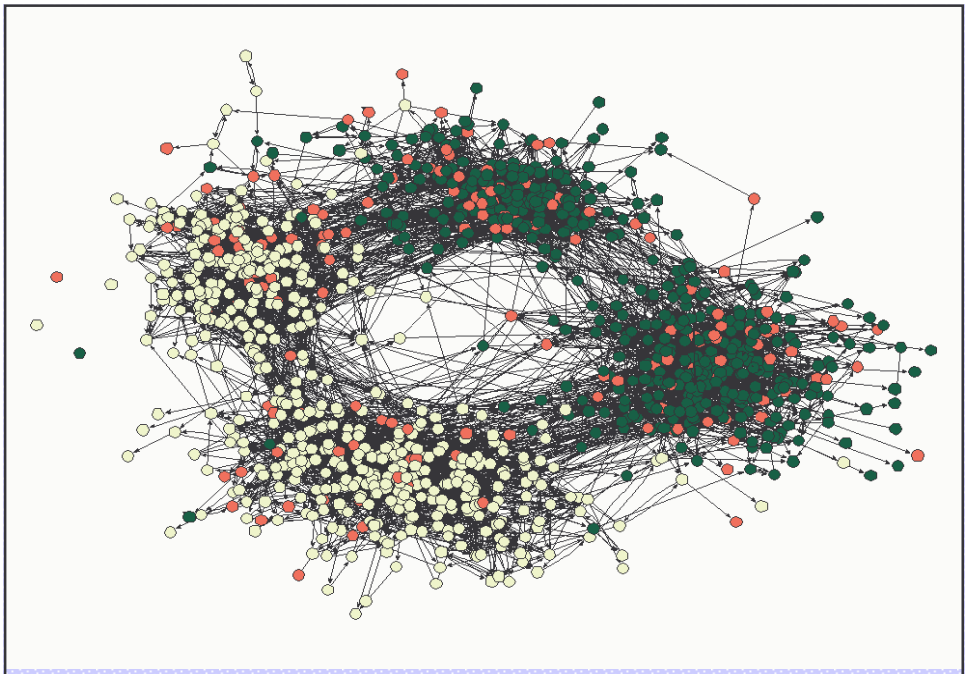
Internet



Overlapping communities in a network of word association. The groups, labeled by the colors, were detected with the Clique Percolation Method by Palla et al.



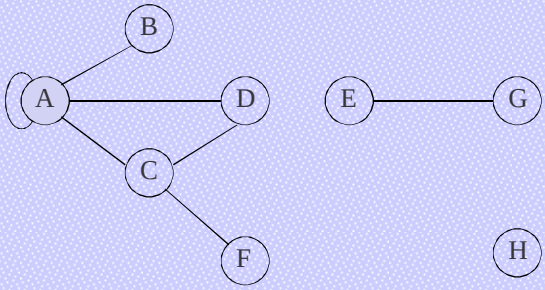
Web site



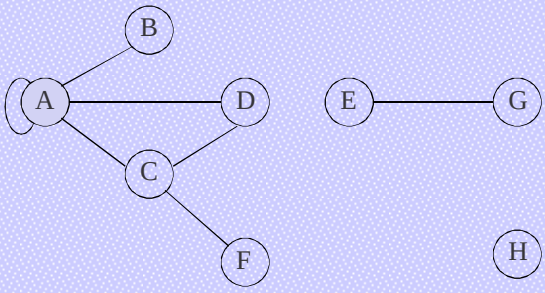
High school friendship

1. INTRODUÇÃO

1) Um **grafo** $G = (V, E)$ consiste num conjunto de **nós** (ou **vértices**) V e num conjunto de **arcos** (ou **arestas**) E. Cada arco é representado por um par de nós. No seguinte exemplo, a seqüência de nós é $V(G) = \{A, B, C, D, E, F, G, H\}$ e o conjunto de arcos é $E(G) = \{(A, B), (A, D), (A, C), (C, D), (C, F), (E, G), (A, A)\}$

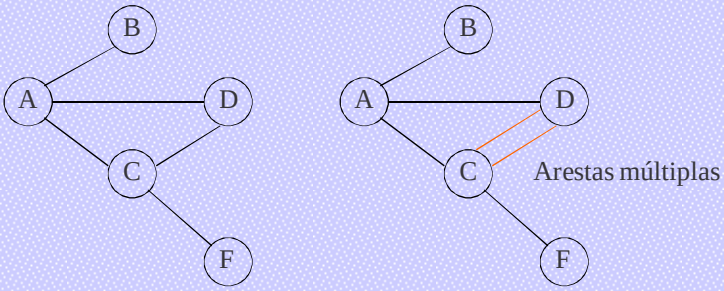


2) A **ordem** de um grafo é dada pela cardinalidade do conjunto de vértices $|V(G)|$, ou seja, pelo número de vértices de G. O número de arestas é denotado por $|E(G)|$.



$|V(G)| = 8, \quad |E(G)| = 6$

3) Quanto um grafo possui mais de uma aresta interligando os mesmos dois vértices diz-se que este grafo possui **arestas múltiplas** ou **arestas paralelas**. Um grafo simples não possui arestas múltiplas. Caso contrário, trata-se de um **multigrafo** ou **grafo múltiplo**.



4) Um grafo é dito **trivial** se for de ordem 0 ou 1.

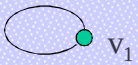
Exemplo,

$V = \{v_1\}$, $E = \varnothing$, $|V| = 1$, $|E| = 0$

Um grafo **vazio** $G = (\varnothing, \varnothing)$ pode ser representado somente por $G = \varnothing$.

5) Se houver uma aresta e do grafo G que possui o mesmo vértices como extremos, ou seja, e = (x, x), então é dito que este grafo possui um **laço**.

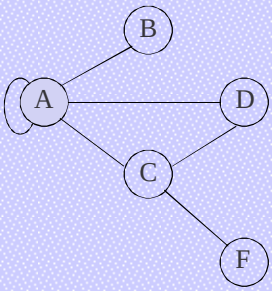
Exemplo,



$V = \{v_1\}$, $E = \{(v_1, v_1)\}$, $|V| = 1$, $|E| = 1$

6) Um nó *v* **incide** em um arco *e* se *v* for um de seus dois nós que constituem *e* (dizemos também que *e* incide em *v*). Um nó v_i será **adjacente** (ou **vizinho**) a um nó v_j se existir um arco entre v_i e v_j.

O **grau** $d(v)$ de um nó *v* é o número de arcos incidentes nesse nó, ou seja, é o número de vértices adjacentes a *v*.

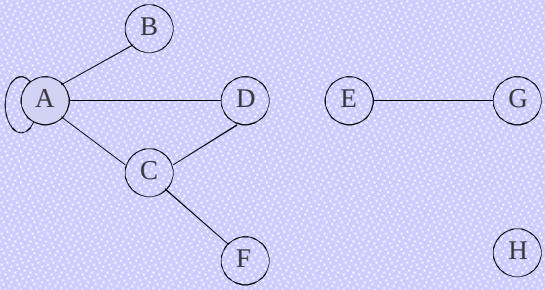


$d(A) = 4$
$d(B) = 1$
$d(C) = 3$
$d(D) = 2$
$d(F) = 1$

A é adjacente de A, B, D e C, mas não é adjacente de F.

7) Diz-se que um grafo é **regular** se todos os vértices tem o mesmo grau.

8) Diz-se que duas arestas são **adjacentes**, ou **vizinhos**, quanto estas possuírem um mesmo extremo, ou vértice.



(A, B) é adjacente de (A, C), mas não é adjacente de (C, D).

9) Um grafo é dito **completo** se todos os seus vértices forem adjacentes.

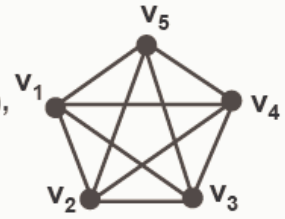
Um grafo completo K_n possui n(n-1)/2 arestas.

Exemplo:

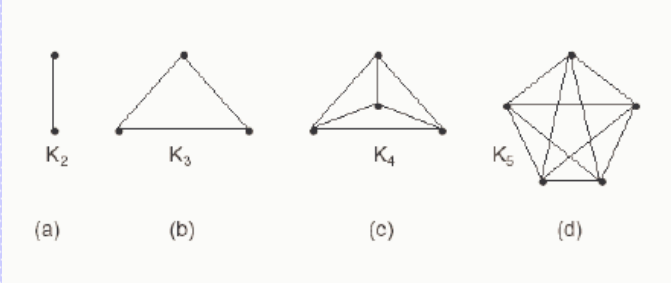
$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_5),$
$(v_2, v_3), (v_2, v_4), (v_2, v_5),$
$(v_3, v_4), (v_3, v_5), (v_4, v_5)\}$

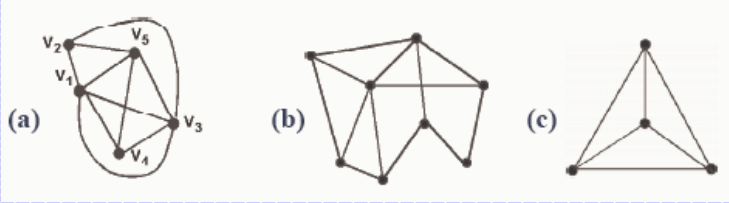
$|V| = 5$ e $|E| = 5(5-1)/2 = 10$



Grafo K_5



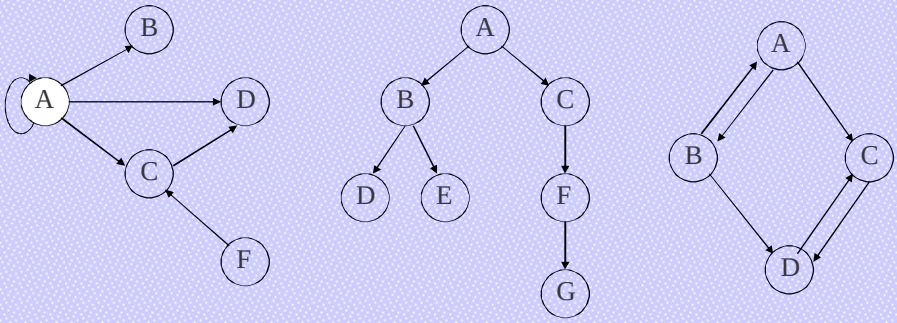
Exercícios



1) Qual a ordem e o número de arestas de cada grafo?
2) Quais dos grafos acima são completos?
3) Quais dos grafos acima são simples?
4) No grafo (a), quais vértices são adjacentes a v_3 e quais arestas são adjacentes a (v_3, v_5)?

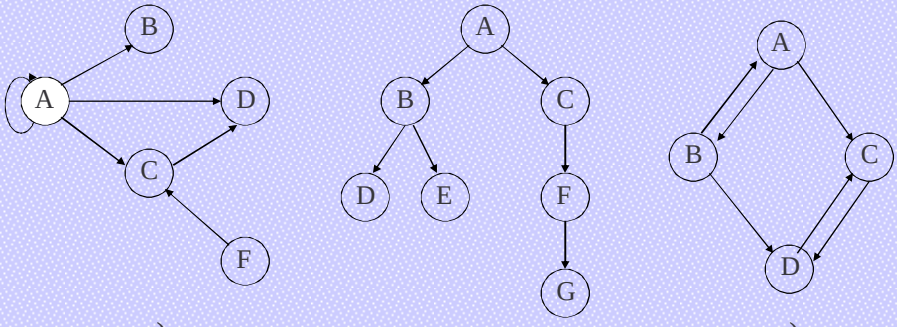
10) Em um grafo D = (V, E), se os pares de nós que formam os arcos forem pares ordenados, diz-se que o grafo é um **grafo orientado** (ou **dígrafo**).

As seguintes figuras ilustram três dígrafos.



a) b) c)

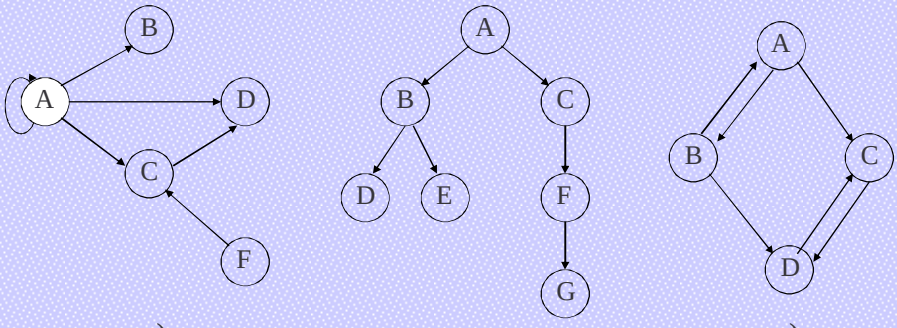
11) Em um grafo orientado**,** cada aresta e = (x, y) possui uma única direção de x para y. Diz-se que (x, y) é **divergente** de x e **convergente** a y.



a) b) c)

(A, B) é divergente de A e convergente a B

12) Em um grafo orientado**,** o **grau de saída** $d_{out}(v)$ de vértice v corresponde ao número de arestas divergentes de v; o **grau de entrada** $d_{in}(v)$ de vértice v corresponde ao número de arestas convergentes a v.



a) b) c)

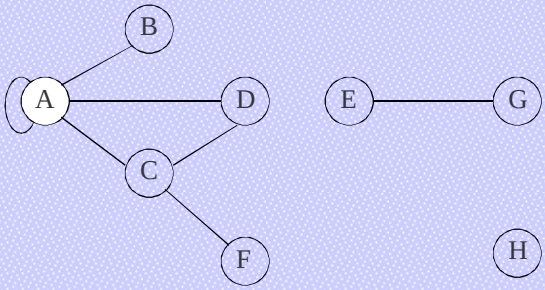
$d_{out}(A) = ?$ $d_{in}(A) = ?$ $d_{out}(C) = ?$ $d_{in}(C) = ?$

13) Um vértice v com grau de saída nulo, ou seja, $d_{out}(v) = 0$, é chamado **sumidouro** (ou **sorvedouro**); Um vértice v com grau de entrada nulo, ou seja, $d_{in}(v) = 0$, é chamado **fonte**.

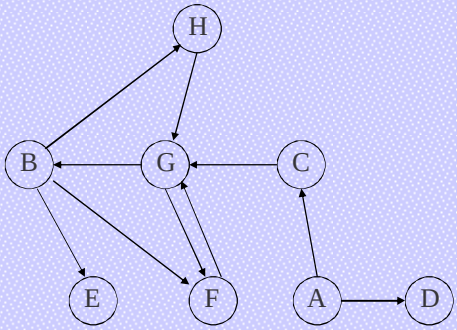
Exercícios

1) O grafo (a) é regular? Por quê?
2) Existe alguma fonte ou sumidouro no grafo (b)?

14) Um **caminho de comprimento *k*** do nó *a* ao nó *b* é definido como uma seqüência de k+1 nós v_1, v_2, ..., v_{k+1}, tal que $v_1 = a$, $v_{k+1} = b$ e adjacente(v_i, v_{i+1}) é verdadeiro para todo i entre 1 e k. Um caminho de um nó para si mesmo é chamado um **laço**. Um caminho v_1, v_2 ... v_k com $v_1 = v_k$, é chamado um **ciclo**. Se um grafo contiver um ciclo, ele será **cíclico**; caso contrario, será **acíclico**.



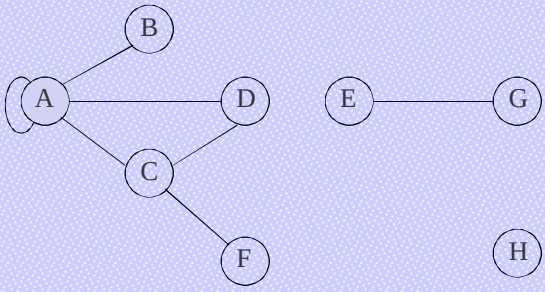
Exemplo



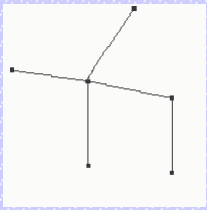
Existe um caminho de comprimento 1 de A até C, dois caminhos de comprimento 2 de B até G e um caminho de comprimento 3 de A até F. Não existe um caminho de B até C. Existem ciclos de B para B, de F para F e de H para H.

15) **Grafo Conexo**

Um grafo $G(V, E)$ é **conexo** se todo vértice for atingível a partir de qualquer outro. Caso contrário, G é **desconexo**.



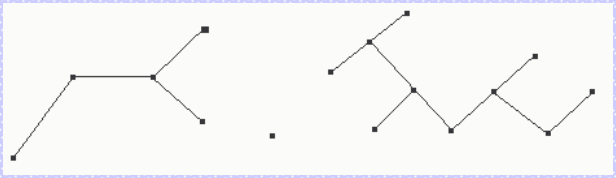
16) **Árvores -** Um grafo que é acíclico e conexo *T*(*V*, *E*)



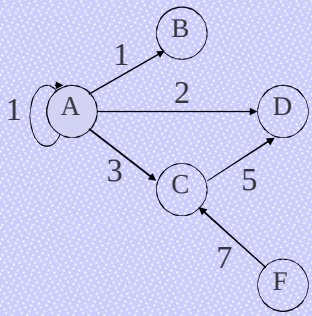
Folha - Um vértice *v* da árvore é uma folha se possuir d(v) <= 1.

Vértice Interior - Um vértice *v* é interior se d(v) > 1.

Floresta - Um conjunto de árvores. Todo grafo acíclico é uma floresta.



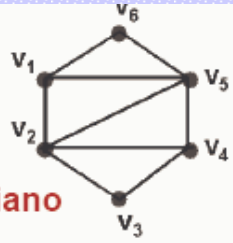
17) Um grafo no qual existe um numero associado a cada arco, é chamado **grafo ponderado** ou **rede**. O numero associado a um arco é chamado **peso**. Este peso pode ser uma distância geográfica, um tamanho geométrico, um custo de tempo ou dinheiro, etc.



18) **Caminho Hamiltoniano** é aquele que contém cada vértice do grafo exatamente uma vez. Um ciclo $v_1, v_2, ..., v_k, v_1$ é Hamiltoniano quando o caminho $v_1, v_2, ..., v_k$ for um caminho Hamiltoniano.

$v_1, v_6, v_5, v_2, v_3, v_4$ é hamiltoniano

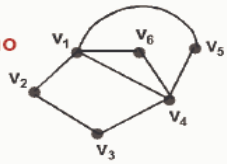
$v_6, v_5, v_4, v_3, v_2, v_1, v_6$ é um ciclo hamiltoniano



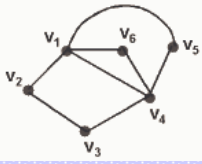
19) **Caminho Euleriano** é aquele que contém cada aresta do grafo exatamente uma vez. Um grafo é Euleriano se há um ciclo em G que contenha todas as arestas.

$v_1, v_6, v_4, v_1, v_2, v_3, v_4, v_5, v_1$ é euleriano

Portanto, este grafo é euleriano



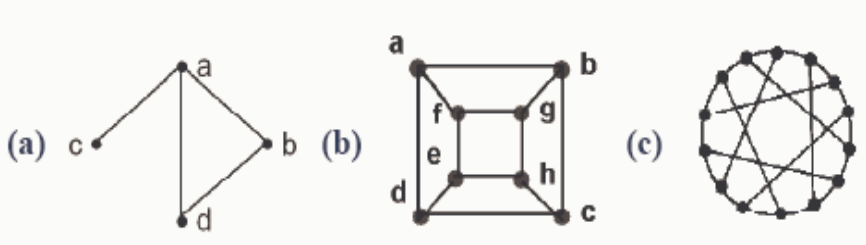
Todos grafos Eulerianos é conexo e todos os seus vértices possuem **grau par**.



É euleriano

Não é euleriano

Exercícios

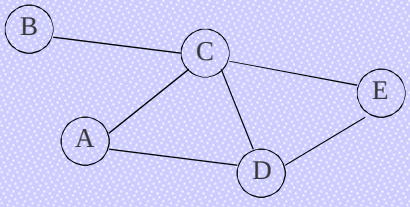


1) Qual dos grafos acima são cíclicos?
2) Indique os grafos que são conexos.
3) Qual(is) dos grafos acima Eulerianos? Quais são Hamiltoniano?

2 REPRESENTAÇÕES DE GRAFOS

2.1 Matriz de Adjacência

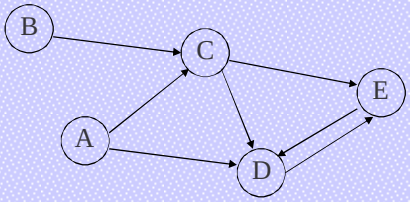
Seja $G = (V, E)$ um grafo com n vértices. A matriz de adjacência para G é um array bidimensional nxn, que denotaremos por A, onde $A(i,j)=1$ se a aresta está presente em G. A matriz de adjacência para grafos não dirigidos é sempre simétrica.



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

O grau de um vértice em um grafo não dirigido, representado por matriz de adjacência, pode ser obtido pela soma de sua linha (ou coluna) correspondente.

Matriz de adjacência para digrafo:



	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

Para um dígrafo, a soma dos elementos na linha *i* representa o grau de saída do vértice v_i enquanto a soma dos elementos na coluna *j* representam o grau de entrada de v_j.

Três Representações de Matriz de Adjacência em C

1) Representação Completa

```
#define MAXNODES 50
struct node {
   /* informacoes associada a cada noh */
}
struct arc {
    bool adj;
   /* informacoes associada a cada arco */
}
struct graph {
   struct node nodes[MAXNODES];
   struct arc arcs[MAXNODES][MAXNODES];
};
struct graph g;
```

Nesta representação, cada nó do grafo é representado por um numero inteiro entre 0 e MAXNODES-1, e o campo vetor *arcs* é um vetor bidimensional representando todo possível par ordenado de nós. O valor de *g. arcs[i][j].adj* é TRUE ou FALSE, dependendo de o nó *j* ser ou não adjacente ao nó *i*.O vetor bidimensional *g.arcs[][].adj* é chamado matriz de adjacência.

2) Um grafo ponderado com um numero fixo de nós pode ser declarado por:

```
struct arc {
    int adj;
    int weight;
}
struct arc g[MAXNODES][MAXNODES];
```

3) Freqüentemente, os nós de um grafo são numerados e nenhuma informação é atribuída a eles. Alem disso, talvez interesse apenas a existência de arcos mas não um peso ou outras informações sobre eles. Neste caso, o grafo poderia ser declarado simplesmente por

```
int adj[MAXNODES][MAXNODES];
```

Operações Primitivas

1. Join - inclui um arco entre dois nós (representação 3))

```
join (adj, node1, node2)
int adj[][MAXNODES];
int node1, node2;
{
    adj[node1][node2] = TRUE;
}
```

2. Elimina um arco (representação 3))

```
remv (adj, node1, node2)
int adj[][MAXNODES];
int node1, node2;
{
   adj[node1][node2] = FALSE;
}
```

3. adjacent - detecta se dois nós foram nós de adjacências (representação 3)).

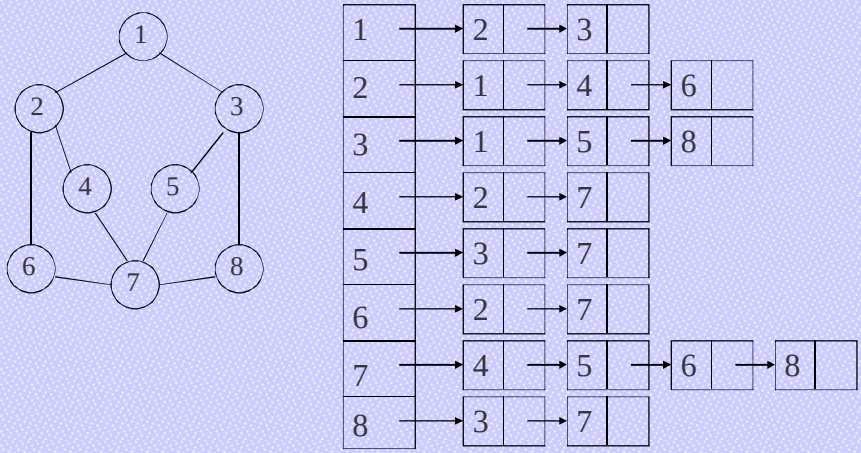
```
adjacent (adj, node1, node2)
int adj[][MAXNODES];
int node1, node2;
{
     return((adj[node1][node2] == TRUE) ? TRUE : FALSE);
}
```

4. joinwt - inclui um arco com determinado peso *wt* (representação 2))

```
joinwt (g, node1, node2, wt)
struct arc g[][MAXNODES];
int node1, node2, wt;
{
   g[node1][node2].adj = TRUE;
   g[node1][node2].weight = wt;
}
```

2.2 Representação Ligada

- A representação em matriz de adjacência de um grafo é freqüentemente inadequada porque 1) requer o conhecimento prévio do numero de nós; 2) mesmo que um grafo tenha muito poucos arcos tal que a matriz de adjacência seja esparsa, será necessário reservar espaço para todo possível arco entre dois nós, quer este arco exista, quer não.
- Como você já deve imaginar, a solução é usar uma estrutura ligada, alocando e liberando nós a partir de uma lista disponível.



- Nesta representação as *n* linhas da matriz de adjacência são representadas como *n* listas encadeadas. Existe uma lista para cada vértice em G. Os nós na lista *i* representam os vértices que são adjacentes ao vértice *i*.
- Cada nó na lista possui pelo menos dois campos: *vértice*, que armazena o índice do vértice que é adjacente *i* e *next*, que é um ponteiro para o próximo nó adjacente. As cabeças das listas podem ser armazenas em um array de ponteiros para facilitar o acesso aos vértices. É conveniente armazenarmos em cada entrada do array de cabeças de listas um campo *flag* que será utilizado posteriormente para indicar se um dado vértice possui alguma propriedade, por exemplo em buscas, este *flag* pode indicar se um dado vértice já foi visitado ou não.

Definição em C - Representação Ligada

```
#define MAXNODES 500
typedef struct adj_node;
typedef struct nodetype;

struct adj_node{
  int vertice;
  adj_node  *next;
};

struct nodetype{
  int info;
  int flag;
  adj_node  *adj;
};

struct nodetype node[MAXNODES];
```

3. PERCURSO DE GRAFOS

- Freqüentemente, desejamos percorrer uma estrutura de dados, i.e., visitar cada elemento de uma maneira sistemática.
- Os elementos do grafo a ser percorrido são os nós do grafo.

- Um percurso de um grafo é mais complexo do que a um percurso de uma lista ou arvore por três razoes:

1) Em geral, não existe um primeiro nó “natural” num grafo a partir do qual o percurso deva começar. Alem disso, assim que um nó inicial é determinado e todos os nós no grafo atingíveis a partir desse nó são visitados, podem restar outros nós no grafo que não foram visitados por não serem atingíveis a partir do nó inicial.

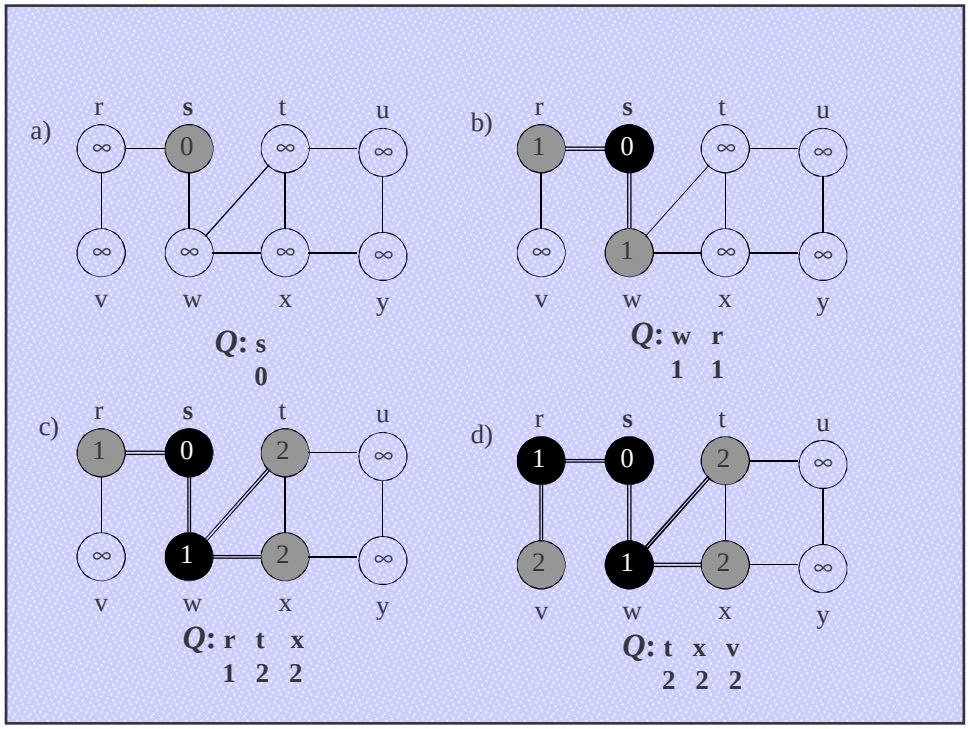
2) Não existe uma seqüência natural entre os sucessores de determinado nó. Conseqüentemente, não existe uma ordem previa na qual os sucessores de determinado nó devam ser percorridos.
3) Um nó de um grafo pode ter mais de um predecessor. Portanto, é possível que um nó seja visitado antes de um de seus predecessores.

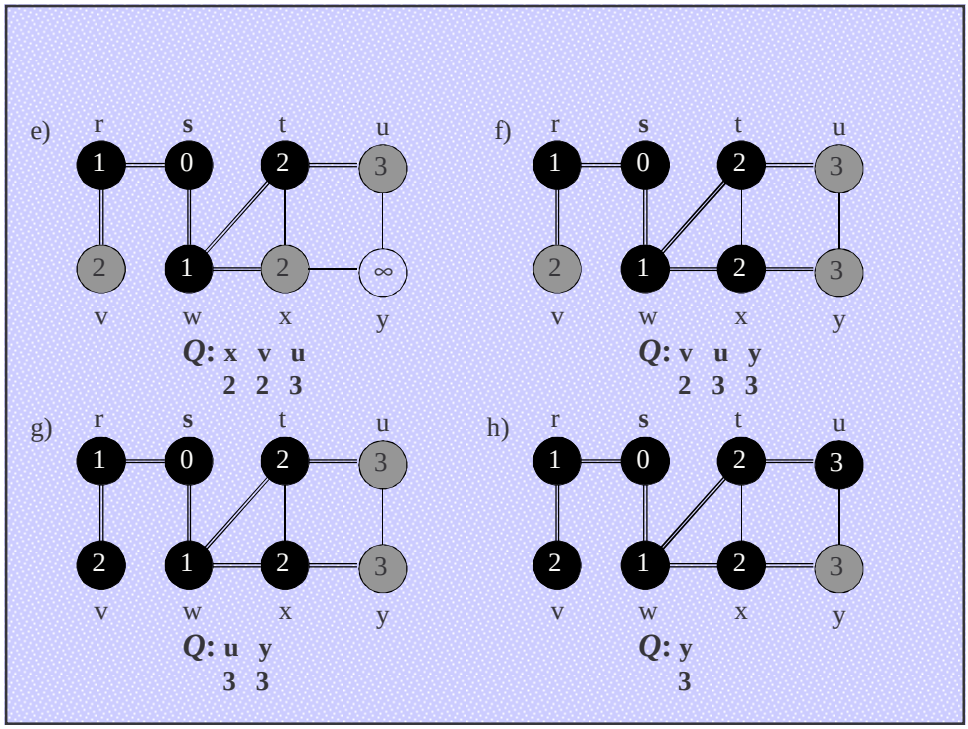
Percurso em Largura

• Um percurso em largura visita todos os sucessores de um nó visitado antes de visitar quaisquer sucessores de qualquer um desses sucessores. Enquanto o percurso em profundidade tende a criar arvores estreitas, muito longas, o percurso em largura tende a criar arvores baixas e muito largas.

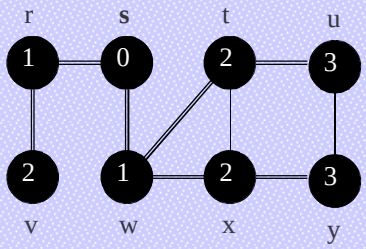
Percurso em Largura (cont.)

• Ao implementar o percurso em profundidade, cada nó visitado é colocado numa pilha (por meio da recursividade), refletindo o fato de que o ultimo nó visitado é o primeiro nó cujos sucessores serão visitados. O percurso em largura é implementado usando uma fila, representando o fato de que o primeiro nó visitado é o primeiro nó cujos sucessores serão visitados.


a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q: s
0
b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q: w r
1 1
c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q: r t x
1 2 2
d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q: t x v
2 2 2


e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q: x v u
2 2 3
f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q: v u y
2 3 3
g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q: u y
3 3
h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q: y
3

i)



***Q*: NULL**

Eficiência do Algoritmo:

- Se for implementado por matriz de adjacência, $T(n) = O(n^2)$
- Se for implementado por lista de adjacência, $T(n) = O(n+e)$, onde e é numero de arestas no grafo.

Algoritmo de Percurso em Largura

G = (V, E) : grafo de entrada;

Q : fila para auxiliar o percurso;

s : primeiro vértice selecionado para começar o percurso;

color[u] : armazena o cor (ou seja, o estado) de cada vértice $u \in V$;

color[u] = WHITE : vértice u ainda não foi descoberto;

color[u] = GRAY : vértice u foi descoberto, mas ainda não foi visitado;

color[u] = BLACK : vértice u foi visitado.

d[u] : armazena a distancia de nó *s* até nó *u*;

π(u) : armazena o predecessor de *u*.

```
BFS(G, s)
1   for (cada vértice u ∈ V[G] - {s}) {
2         color[u] = WHITE;
3         d[u] = ∞;
4         π[u] = NIL;
    }
5   color[s] = GRAY;
6   d[s] = 0;
7   π[s] = NIL;
8   Q ← {s}
9   while (Q != NULL) {
10        u = head[Q];
11        for (cada v ∈ adj[u])
12            if (color[v] = WHITE) {
13                  color[v] = GRAY;
14                  d[v]++;
15                  π[v] = u;
16                  ENQUENE(Q, v);
                }
17          DEQUENE(Q);
18          color[u] = BLACK;
    }
```

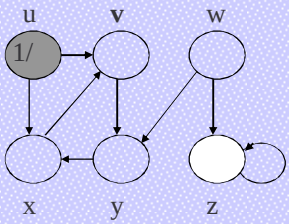
Percurso em Profundidade

• Um percurso em profundidade, como o próprio nome indica, percorre um único caminho do grafo até onde ele possa chegar, isto é, até visitar um nó sem sucessores ou um nó cujos sucessores já tenham sido visitados. Em seguida, ele continua no ultimo nó no caminho recém-percorrido que tenha um sucessor não visitado e começa a percorrer um novo caminho emanando a partir desse nó. As arvores geradoras criadas por um percurso em profundidade por nível tendem a ser muito profundas.

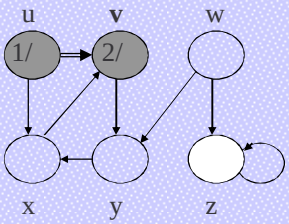
Percurso em Profundidade (cont.)

- Podem existir vários percursos em profundidade e arvores geradoras em profundidade para determinado grafo. A característica fundamental de um percurso em profundidade é que, depois que um nó é visitado, todos os descendentes do nó são visitados antes de seus irmãos não-visitados.

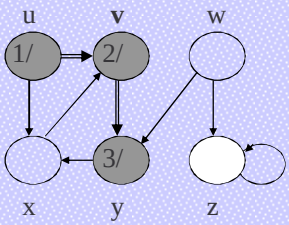
a)



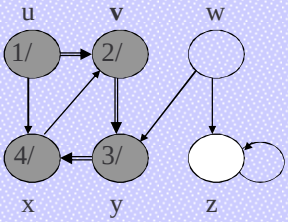
b)



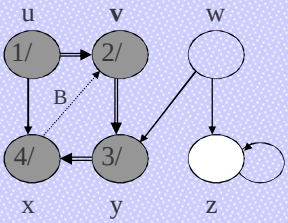
c)



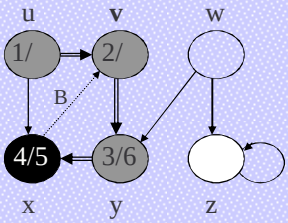
d)



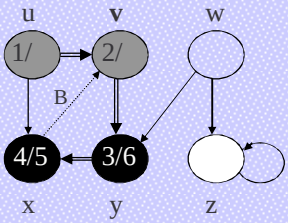
e)


u
v
w
1/
2/
B
4/
3/
x
y
z

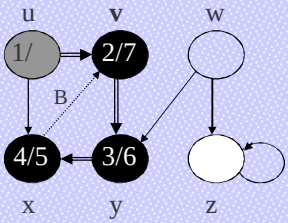
f)


u
v
w
1/
2/
B
4/5
3/6
x
y
z

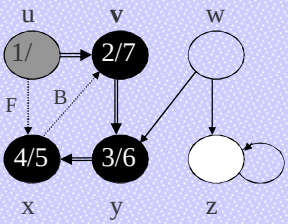
g)


u
v
w
1/
2/
B
4/5
3/6
x
y
z

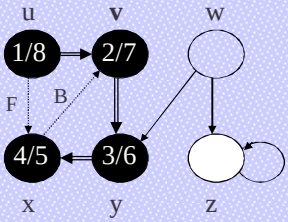
h)


u
v
w
1/
2/7
B
4/5
3/6
x
y
z

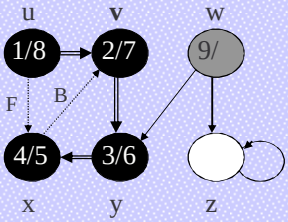
i)


u
v
w
1/
2/7
F
B
4/5
3/6
x
y
z

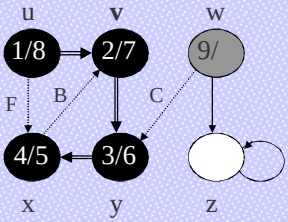
j)


u
v
w
1/8
2/7
F
B
4/5
3/6
x
y
z

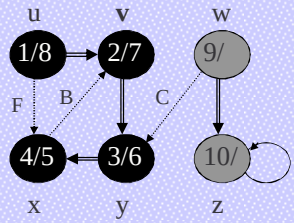
k)


u
v
w
1/8
2/7
9/
F
B
4/5
3/6
x
y
z

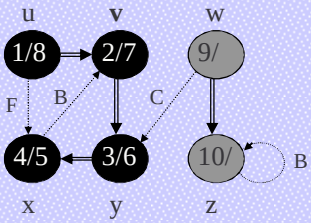
l)


u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
x
y
z

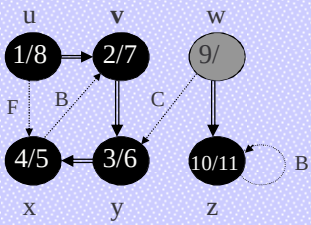
m)



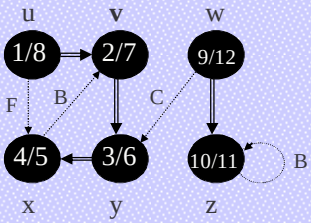
n)



o)



p)



Algoritmo de Percurso em Profundidade

G = (V, E) : grafo de entrada;

color[u] : armazena o cor (ou seja, o estado) de cada vértice $u \in V$;

color[u] = WHITE : vértice *u* ainda não foi descoberto;

color[u] = GRAY : vértice *u* foi descoberto, mas ainda não foi visitado;

color[u] = BLACK : vértice *u* foi visitado.

d[u] : registra o momento quando u for primeira vez descoberto;

f[u] : registra o momento quando u for visitado.

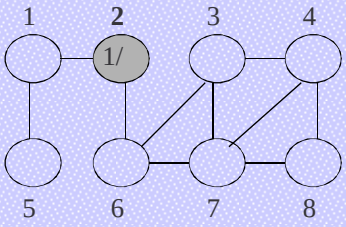
```
DFS(G)
1   for cada vértice u ∈ V[G] {
2           color[u] = WHITE;
3           π[u] = NIL;
    }
4   time = 0;
5   for cada vértice u ∈ V[G] - {s}
6           if (color[u] == WHITE)
7                   DFS_VISIT(u);

DFS_VISIT(u)
1   color[u] = GRAY;
2   d[u] = time = time++;
3   for cada v ∈ adj[u]
4           if (color[v] = WHITE) {
5                   π[v] = u;
6                   DFS_VISIT(v);
            }
7   color[u] = BLACK;
8   f[u] = time = time++;
```

Eficiência do Algoritmo:

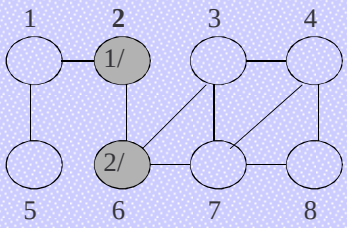
- Se for implementado por matriz de adjacência, $T(n) = O(n^2)$
- Se for implementado por lista de adjacência, $T(n) = O(n+e)$, onde e é numero de arestas no grafo.

Percurso em Profundidade – Grafo não direcionado

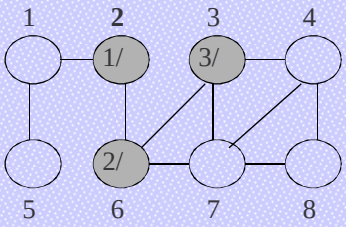
a)



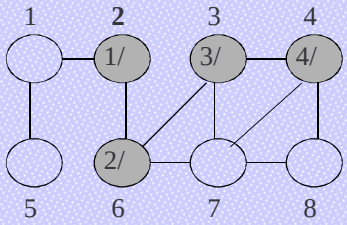
b)



c)

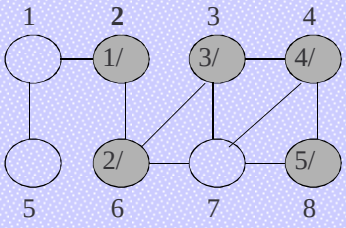


d)

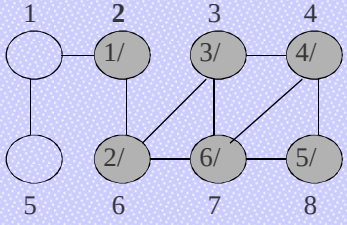


Percurso em Profundidade – Grafo não direcionado

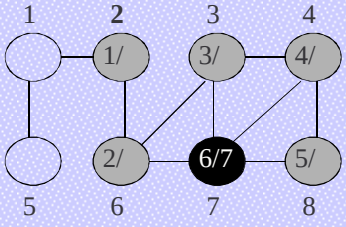
e)



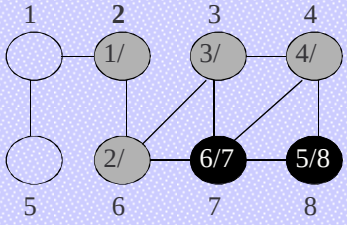
f)



g)

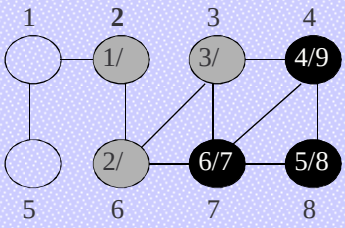


h)

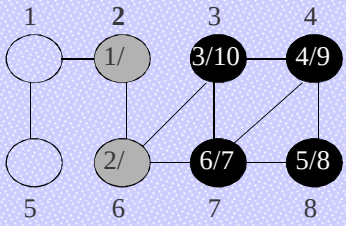


Percurso em Profundidade – Grafo não direcionado

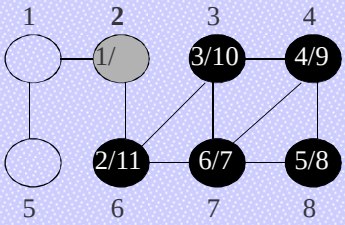
i)



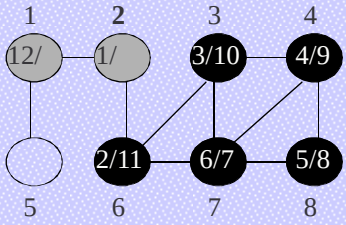
j)



k)

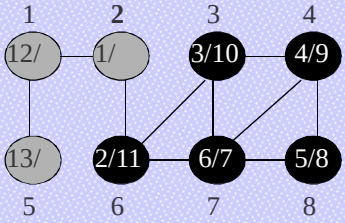


l)

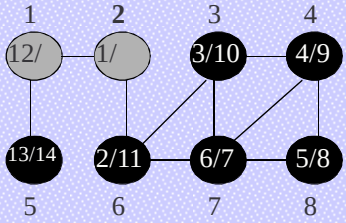


Percurso em Profundidade – Grafo não direcionado

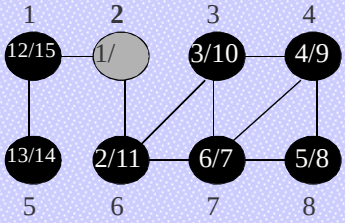
m)



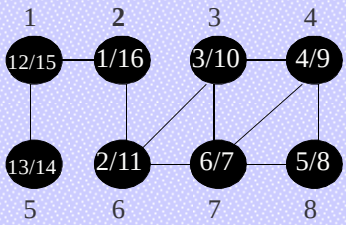
n)



o)



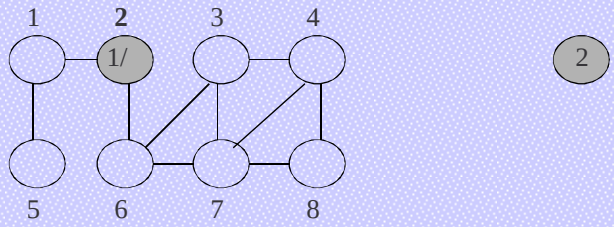
p)



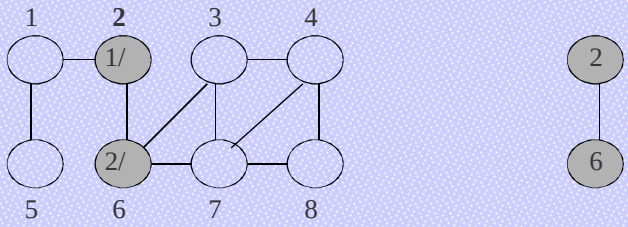
Árvore de Percurso em Profundidade

A execução do percurso em profundidade gera uma árvore chamada de árvore de percurso em profundidade.

a)

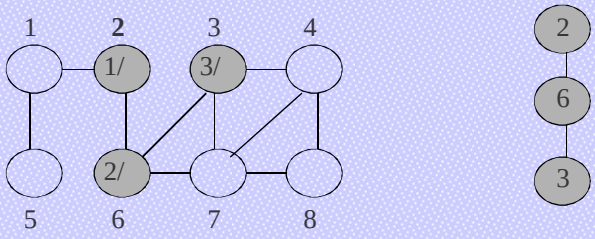


b)

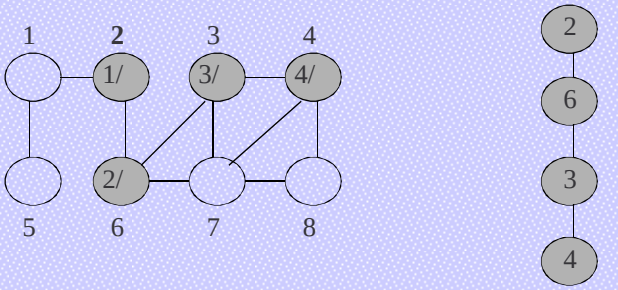


Árvore de Percurso em Profundidade

c)

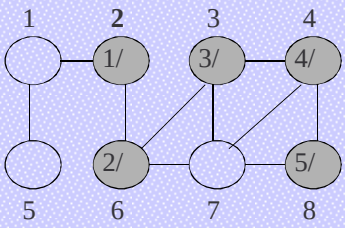


d)



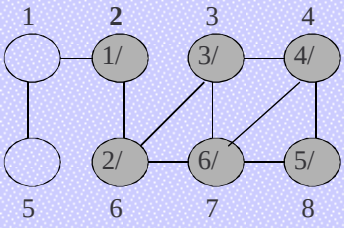
Árvore de Percurso em Profundidade

e)



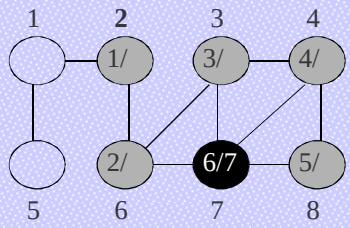
Árvore de Percurso em Profundidade

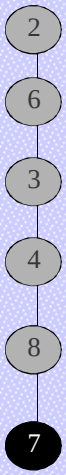
f)



Árvore de Percurso em Profundidade

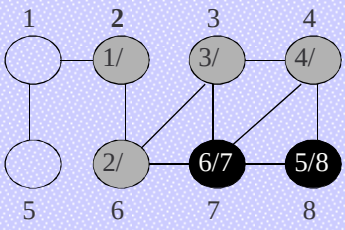
g)





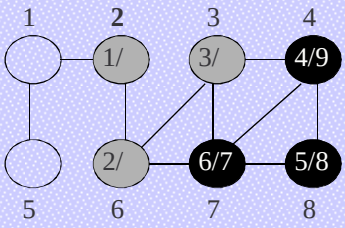
Árvore de Percurso em Profundidade

h)



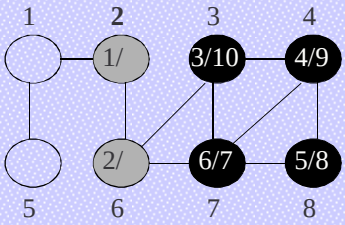
Árvore de Percurso em Profundidade

i)



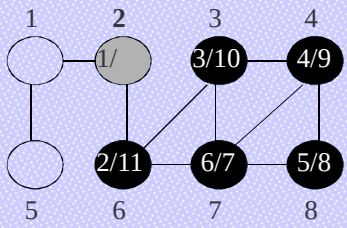
Árvore de Percurso em Profundidade

j)



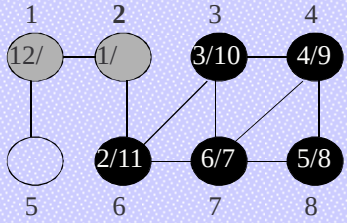
Árvore de Percurso em Profundidade

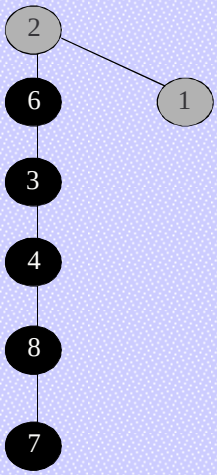
k)



Árvore de Percurso em Profundidade

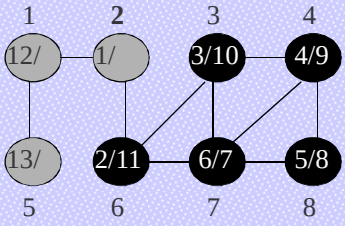
l)

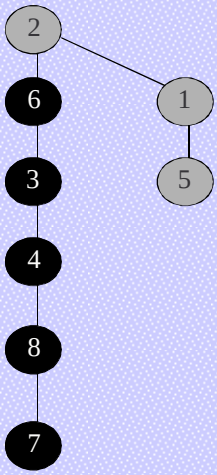




Árvore de Percurso em Profundidade

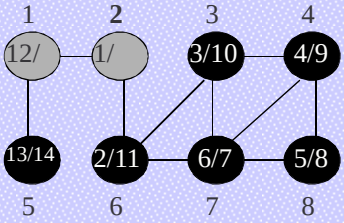
m)

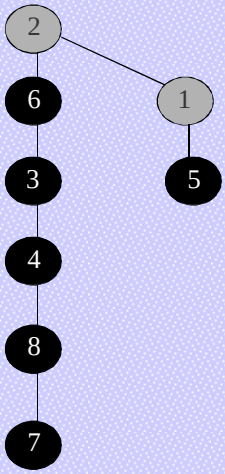




Árvore de Percurso em Profundidade

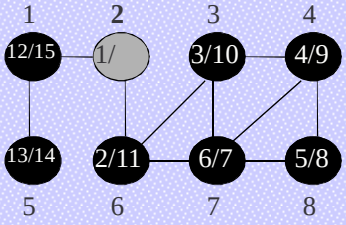
n)

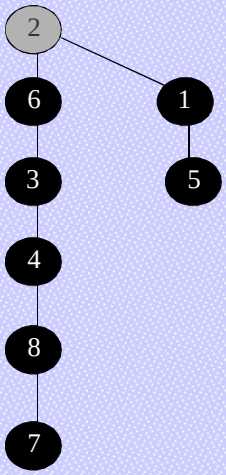




Árvore de Percurso em Profundidade

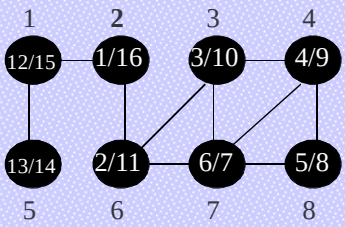
o)

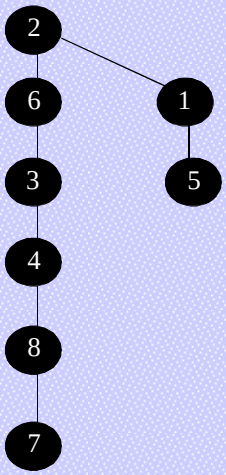




Árvore de Percurso em Profundidade

p)





Classificação de Arestas

As arestas de um grafo podem ser classificadas conforme a sua árvore de busca em profundidade:

– **Arestas de árvore**: arestas que ocorrem na árvore de busca em profundidade;

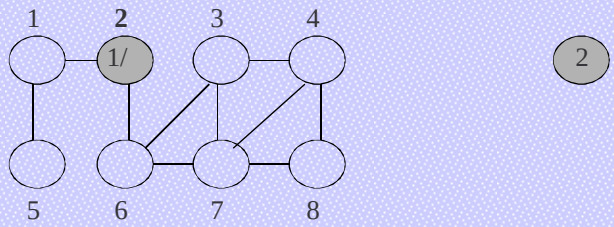
– **Arestas de retorno**: arestas que ligam com um nó antecessor na árvore;

– **Arestas de avanço**: arestas que ligam com um nó descendente na árvore;

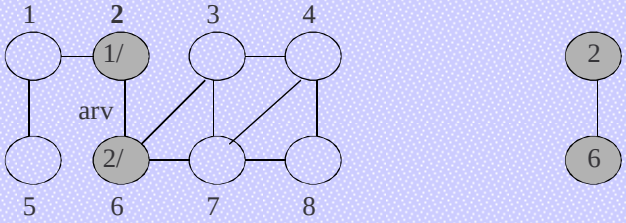
– **Arestas de cruzamento**: demais arestas.

Árvore de Percurso em Profundidade

a)

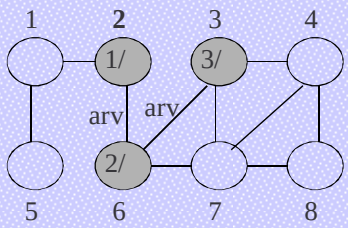


b)

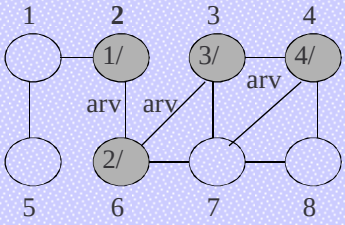


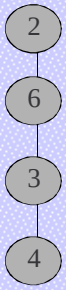
Árvore de Percurso em Profundidade

c)



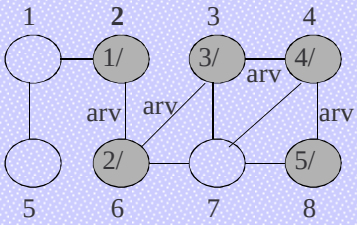
d)





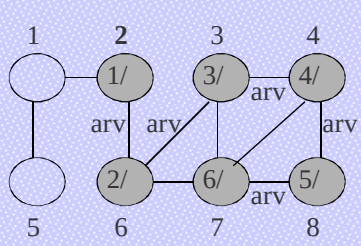
Árvore de Percurso em Profundidade

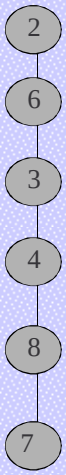
e)



Árvore de Percurso em Profundidade

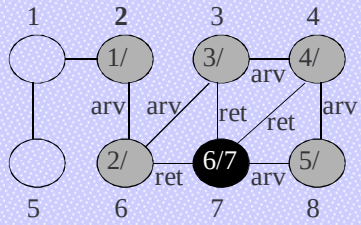
f)

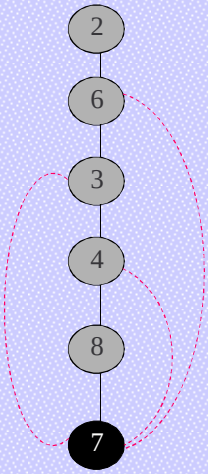




Árvore de Percurso em Profundidade

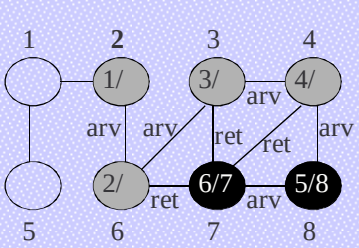
g)

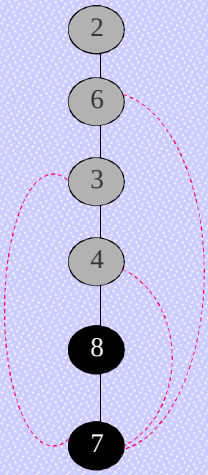




Árvore de Percurso em Profundidade

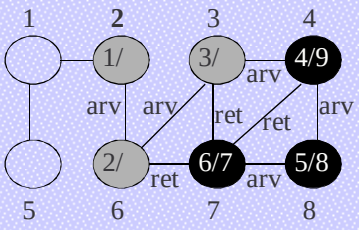
h)

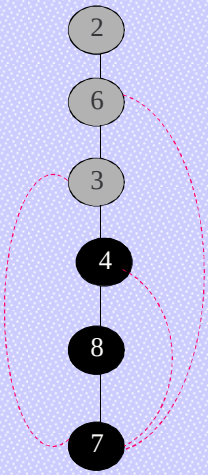




Árvore de Percurso em Profundidade

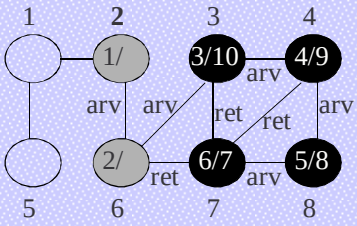
i)

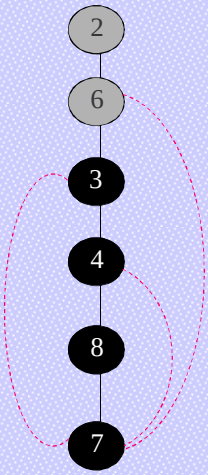




Árvore de Percurso em Profundidade

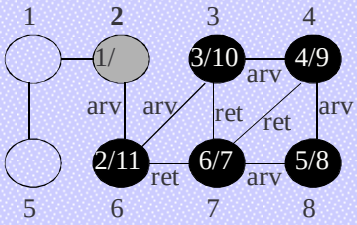
j)

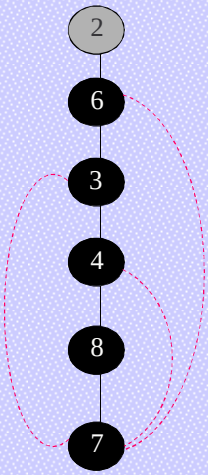




Árvore de Percurso em Profundidade

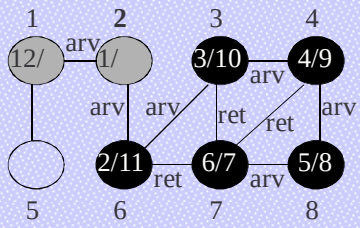
k)

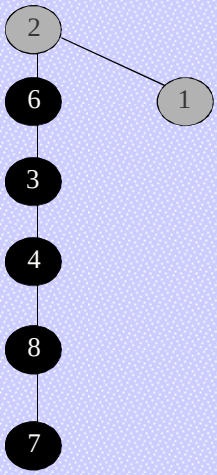




Árvore de Percurso em Profundidade

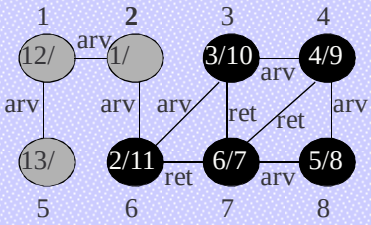
l)

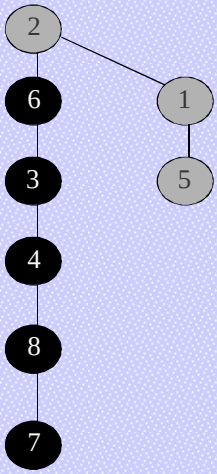




Árvore de Percurso em Profundidade

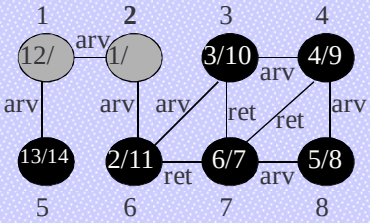
m)

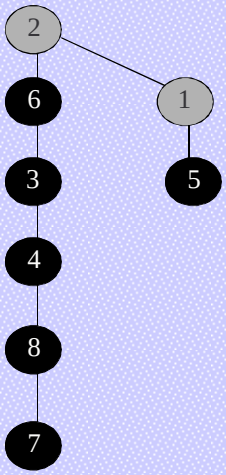




Árvore de Percurso em Profundidade

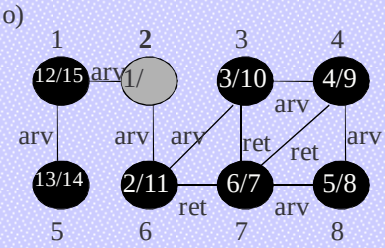
n)

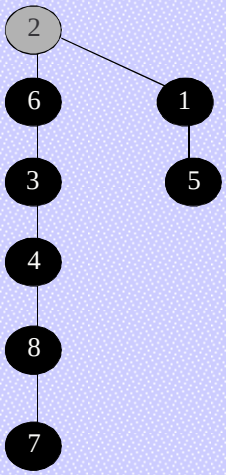




Árvore de Percurso em Profundidade

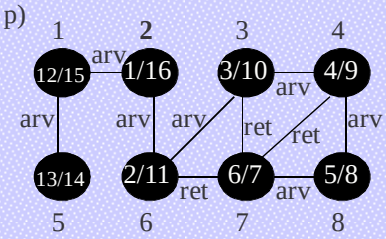
o)

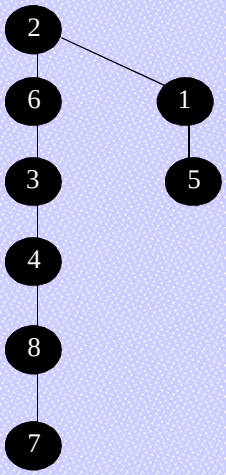




Árvore de Percurso em Profundidade

p)

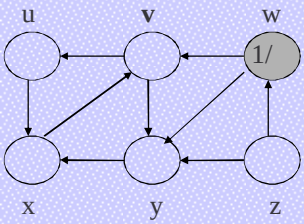




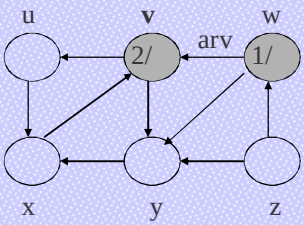
Em um grafo não orientado, todas as arestas são de árvore ou de retorno.

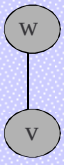
Árvore de Percurso em Profundidade - dígrafo

a)



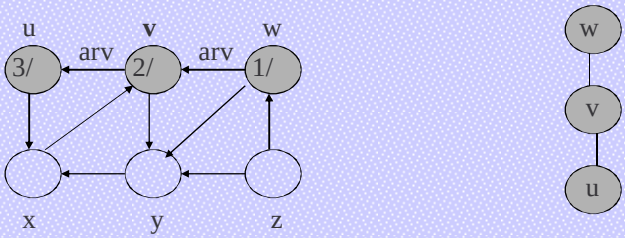
b)



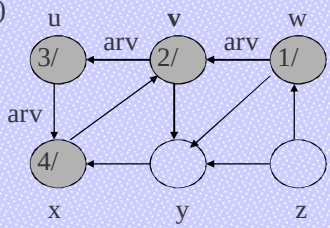


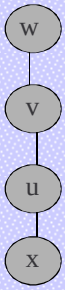
Árvore de Percurso em Profundidade - dígrafo

c)



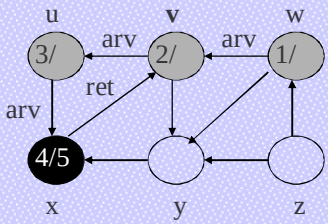
d)

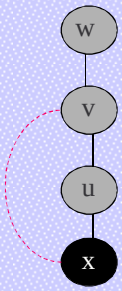




Árvore de Percurso em Profundidade - dígrafo

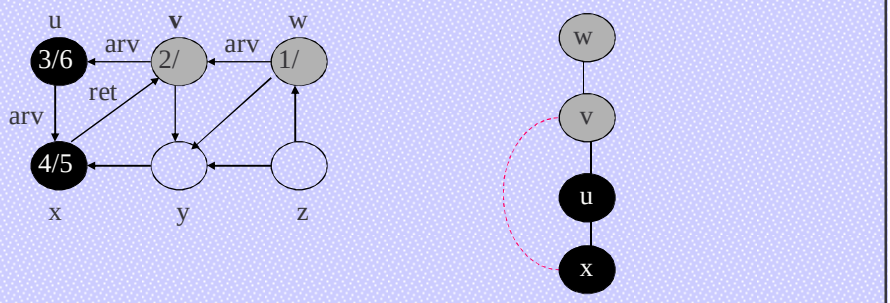
e)





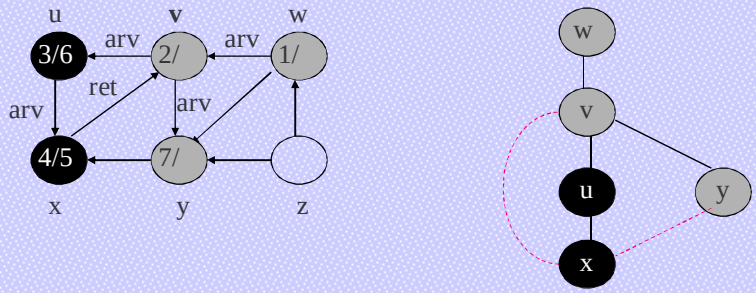
Árvore de Percurso em Profundidade - dígrafo

f)



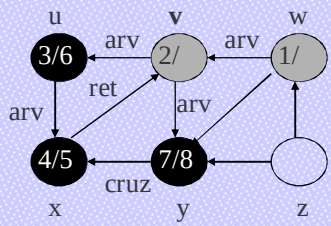
Árvore de Percurso em Profundidade - dígrafo

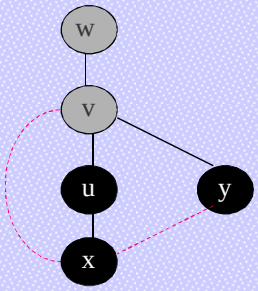
g)



Árvore de Percurso em Profundidade - dígrafo

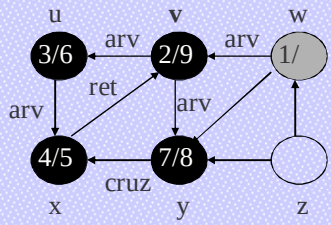
h)

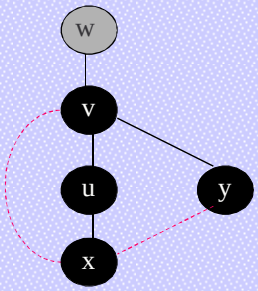




Árvore de Percurso em Profundidade - dígrafo

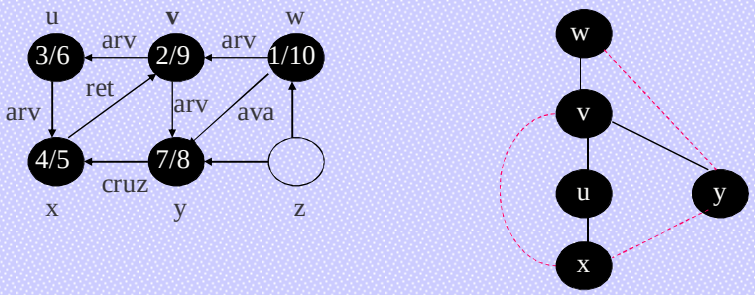
i)





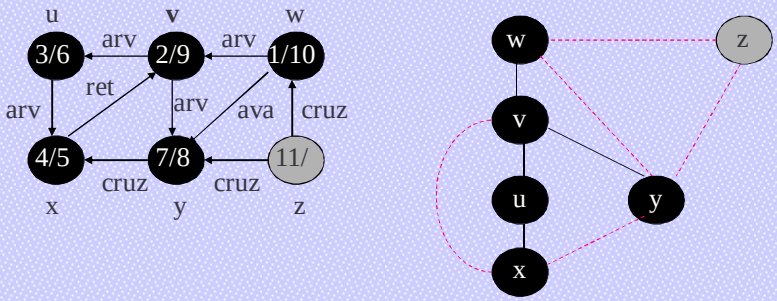
Árvore de Percurso em Profundidade - dígrafo

j)



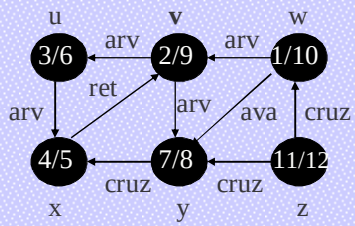
Árvore de Percurso em Profundidade - dígrafo

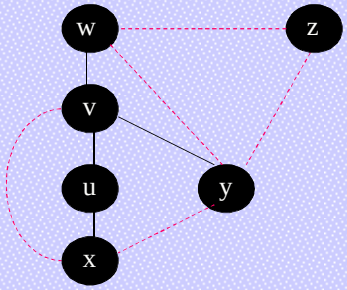
k)



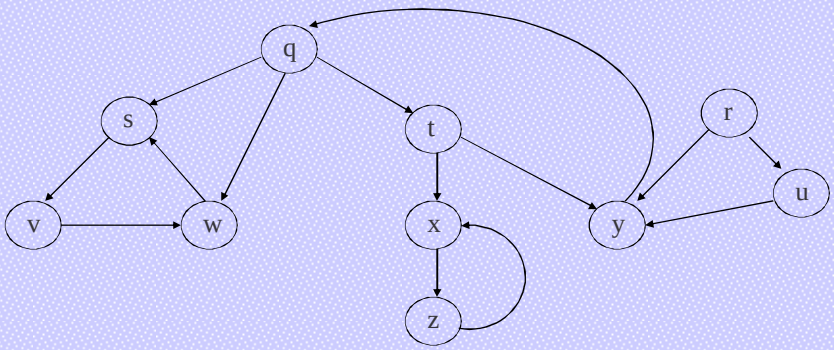
Árvore de Percurso em Profundidade - dígrafo

l)





Exercício



Mostre como a busca em profundidade funciona sobre o grafo acima. Suponha que o loop for das linhas 5-7 do procedimento DFS considere os vértices em ordem alfabética, e suponha que cada lista de adjacências esteja em ordem alfabética. Mostre os tempos de descoberto e termino para cada vértice, e mostre também a classificação de cada aresta.

Aplicações de Percurso em Profundidade

Teste para Verificar se Grafo é Acíclico

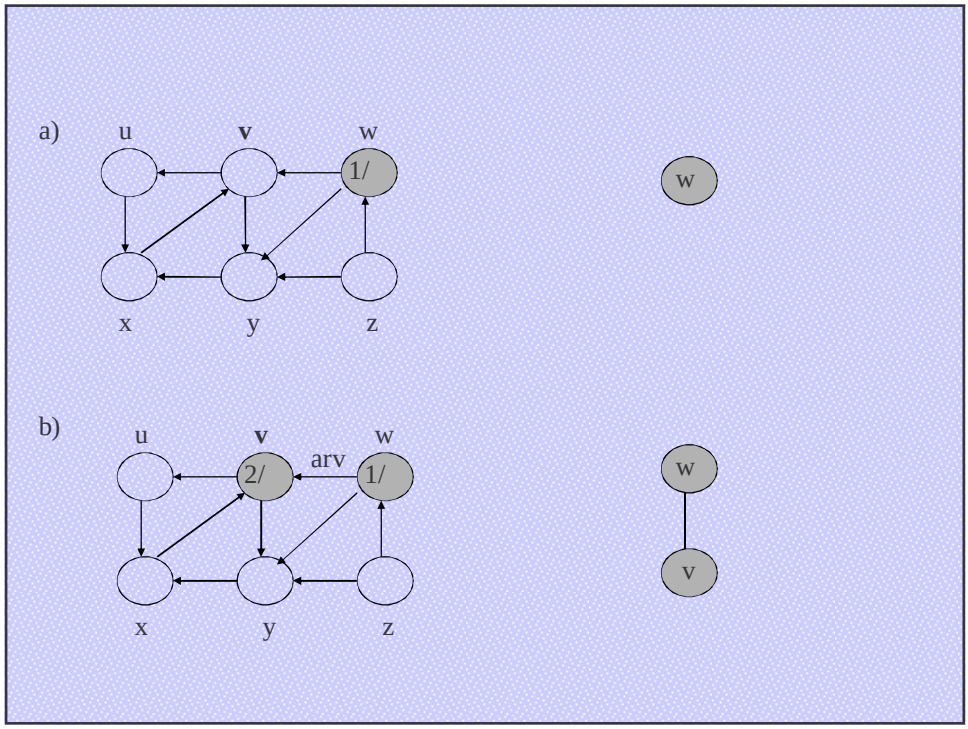
• Percurso em profundidade pode ser usada para verificar se um grafo é acíclico ou contém um ou mais ciclos.

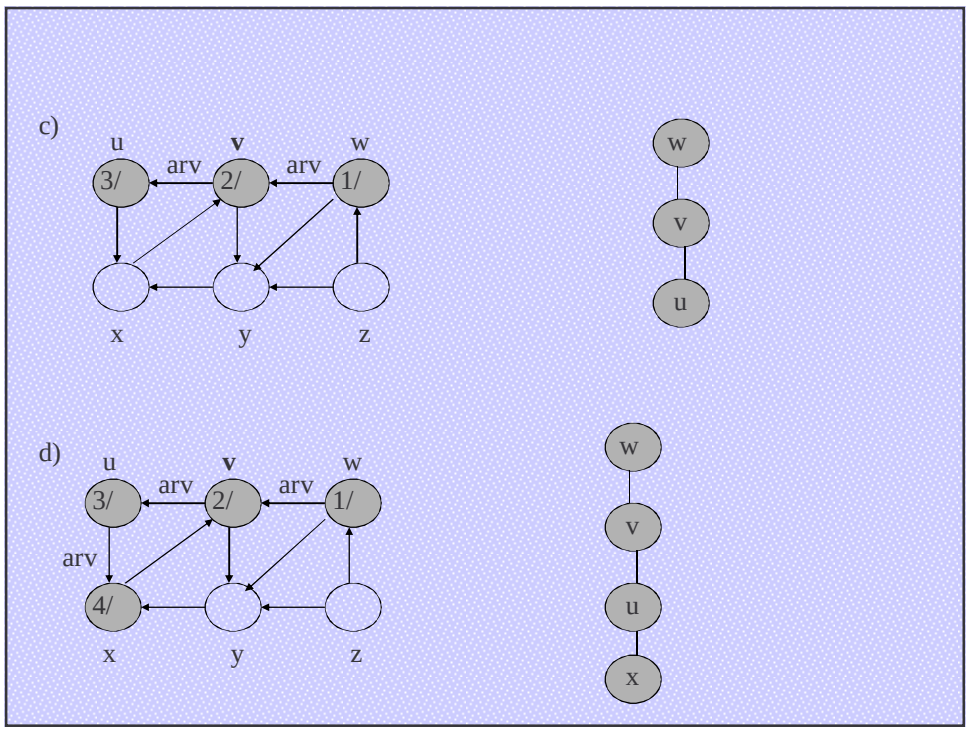
• Se uma aresta de retorno é encontrada durante o percurso em profundidade em G, então o grafo tem ciclo.

Teste para Verificar se Grafo é Acíclico

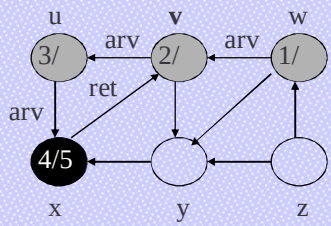
•Um grafo direcionado G é acíclico se e somente se o percurso em profundidade em G não apresentar arestas de retorno.

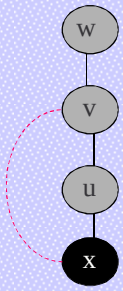
• O algoritmo de percurso em profundidade pode ser modificado para detectar ciclos em grafos orientados simplesmente verificando se um vértice w adjacente a v possui cor cinza na primeira vez que a aresta (v, w) é percorrida.


a)
u
v
w
1/
x
y
z
w
b)
u
v
w
arv
2/
1/
x
y
z
w
v


c)
u
v
w
arv
arv
3/
2/
1/
x
y
z
w
v
u
d)
u
v
w
arv
arv
3/
2/
1/
arv
4/
x
y
z
w
v
u
x

e)





O grafo é cíclico

Ordenação Topológico

- Um grafo direcionado acíclico é também chamado de **dag** (directed acyclic graph).
- Um dag é diferente de uma árvore, uma vez que as árvores são não direcionadas.
- Dags podem ser utilizados, por exemplo, para indicar precedências entre eventos.

Ordenação Topológico

- A ordenação topológica é uma ordenação linear de todos os vértices, tal que se G contém uma aresta (u,v) então u aparece antes de v.
- Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita.

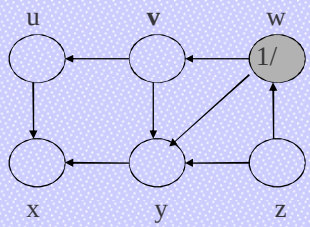
Ordenação Topológico

TOPOLOGICAL-SORT(G)

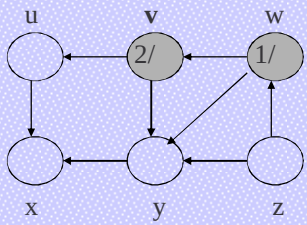
1. Chamar DFS(G) para calcular o tempo de término f[v] para cada vértice v;
2. À medida que cada vértice é terminado, inserir o vértice à frente de uma lista ligada;
3. Retorna a lista ligada de vértices.

Ordenação Topológico

a)

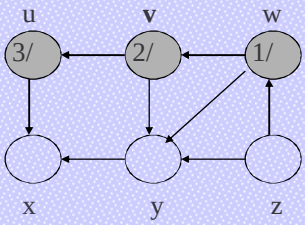


b)

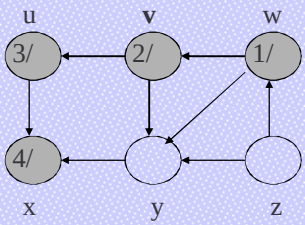


Ordenação Topológico

c)

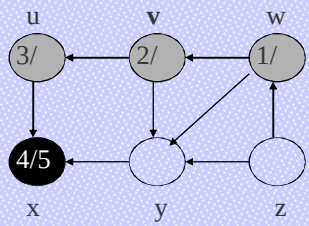


d)



Ordenação Topológico

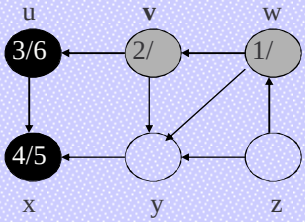
e)



x

Ordenação Topológico

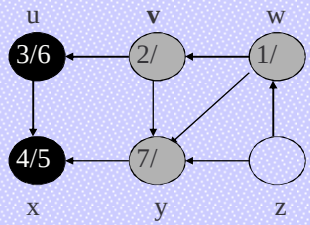
f)



u x

Ordenação Topológico

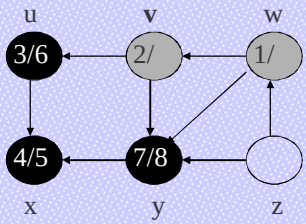
g)



u x

Ordenação Topológico

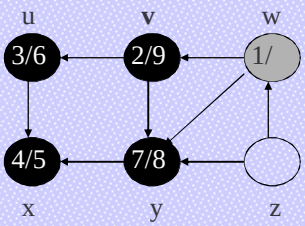
h)



y u x

Ordenação Topológico

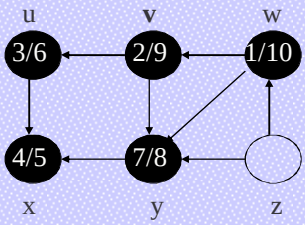
i)



v y u x

Ordenação Topológico

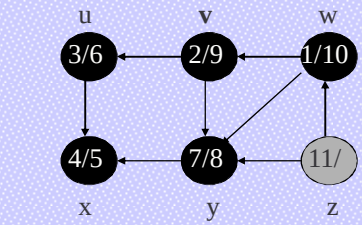
j)



w v y u x

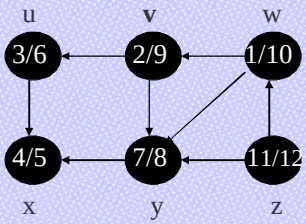
Ordenação Topológico

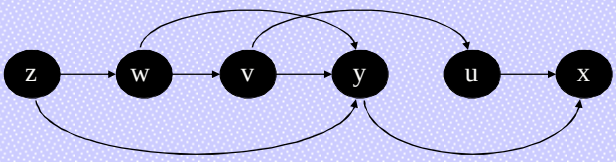
k)



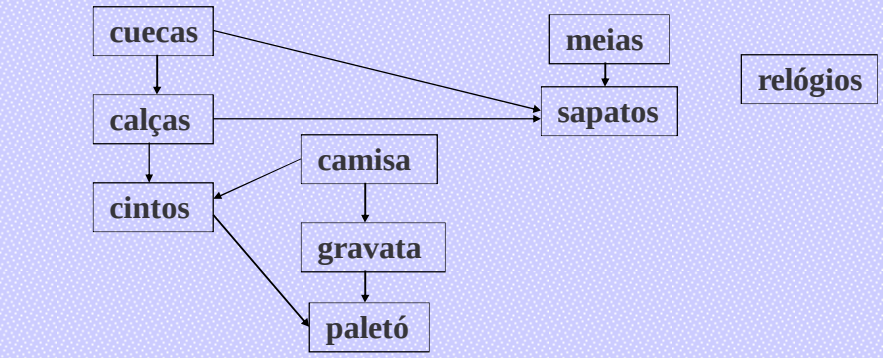
Ordenação Topológico

l)

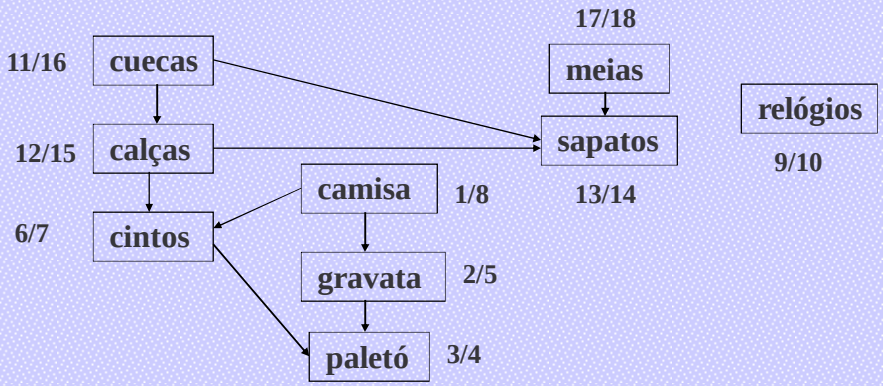




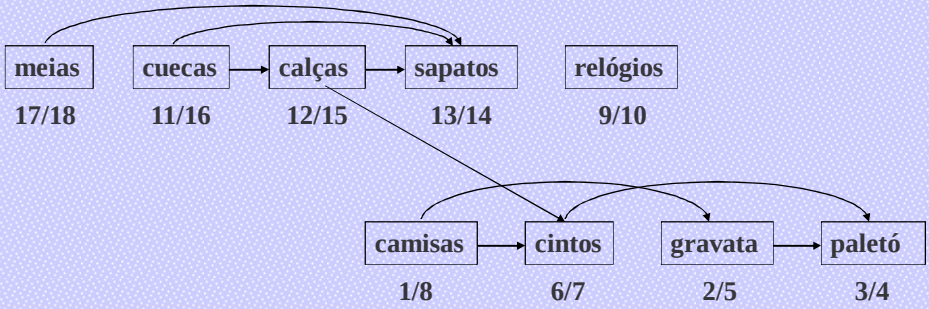
Ordenação Topológico



Ordenação Topológico



Ordenação Topológico



Ordenação Topológico – outra técnica

Descrição do Problema

Suponha que um cozinheiro receba um pedido para prepara um ovo frito. A tarefa de fritar um ovo pode ser decomposta em varias subtarefas distintas:

pegar o ovo	estalar o ovo	pegar o óleo
untar a frigideira	esquentar o óleo	por o ovo na frigideira
esperar o ovo fritar	retirar o ovo	

Algumas dessas tarefas precisam ser feitas antes de outras, e.x., “pegar o ovo” deve preceder à “quebrar o ovo”. Outras podem ser feitas simultaneamente. E.x., “pegar o ovo” e “esquentar o óleo”.

Objetivo

O cozinheiro quer oferecer o mais rápido serviço possível e presume-se que ele tenha à sua disposição uma grande quantidade de auxiliares. O problema resume-se em atribuir tarefas aos auxiliares de modo a finalizar o serviço no menor intervalo de tempo possível.

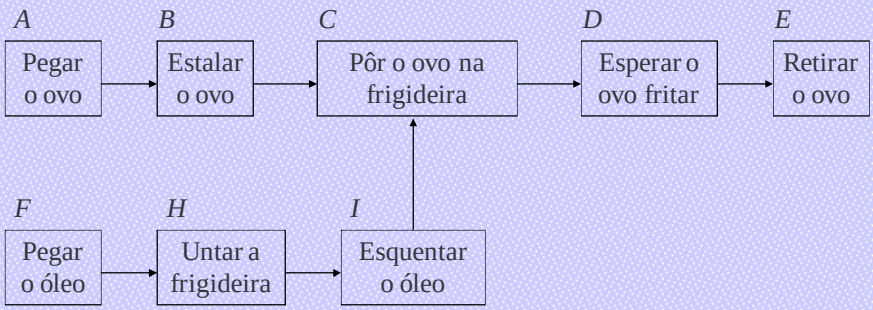
Aplicações Relevantes

Embora esse exemplo possa parecer trivial, ele é típico de vários problemas de escalonamento do mundo real. É possível que um sistema de computador precisa escalonar operações para minimizar o tempo desempenho; o compilador pode precisar escalonar operações em linguagem de maquina para reduzir o tempo de execução; ou o gerente de uma fabrica precisa organizar uma linha de de montagem para diminuir o tempo de produção, etc.

Formulação do Problema

Representamos o problema como um grafo. Cada nó do grafo representa uma subtarefa e cada arco <*x*, *y*> representa a exigência de que a subtarefa *y* não pode ser executada antes do termino da subtarefa *x*. Observe que o grafo não pode conter um ciclo do nó *x* até ele mesmo, a subtarefa *x* não poderia ser iniciada até que a subtarefa *x* tivesse sido terminada. Então, o grafo é acíclico orientado.

Formulação do Problema (cont.)



Solução do Problema

- Como *G* não contem um ciclo, deve existir pelo menos um nó em G sem antecessores. No grafo anterior, os nós *A* e *F* não tem predecessores;
- Assim, as subtarefas que eles representam podem ser executadas imediata e simultaneamente, sem esperar o termino de nenhuma outra tarefa. Toda tarefa adicional precisara esperar até que pelo menos uma dessas tarefas termine.

Solução do Problema (cont.)

- Assim que esses duas tarefas forem executadas, seus nós poderão ser removidos do grafo, o grafo resultante deve também conter pelo menos um nó sem predecessor. No exemplo, esses dois nós são *B* e *H*. Assim, as subtarefas *B* e *H* podem ser executadas simultaneamente no segundo período de tempo.
- Continuando dessa maneira, descobrimos que o menor intervalo de tempo em que o ovo pode ser frito é seis períodos de tempo e que um máximo de dois auxiliares precisa ser utilizados, como segue:

Período de tempo	auxiliar 1	auxiliar 2
1	pegar o ovo	pegar o óleo
2	estalar o ovo	untar a frigideira
3	esquentar o óleo	
4	por o ovo na frigideira	
5	esperar o ovo fritar	
6	retirar o ovo	

Algoritmo

processo anterior pode ser descrito da seguinte maneira:

1. Ler as precedências e construir o grafo;
2. Usar o grafo para determinar as subtarefas que podem ser feitas simultaneamente.

Algoritmo

O passo 2 pode ser detalhado por meio do seguinte algoritmo:

while (o grafo não está vazio) {

i) determina os nós sem predecessores;

ii) dá saída nesse grupo de nós com indicação de que eles podem ser executados simultaneamente no próximo período de tempo;

iii) remover estes nós e seus arcos incidentes do grafo;

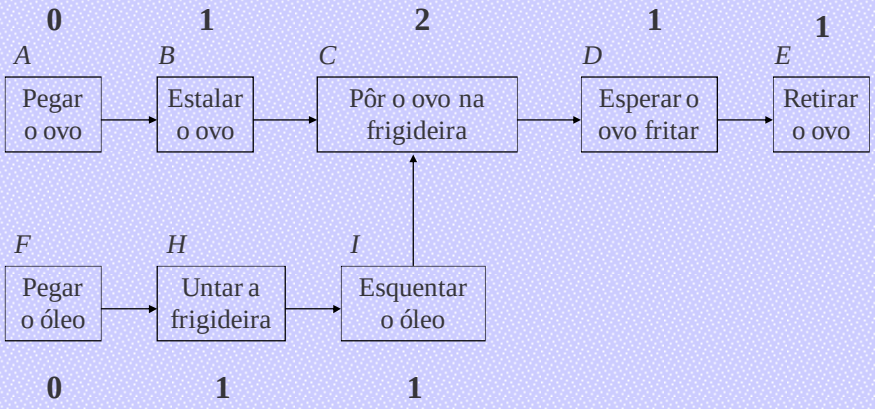
}

Como Determinar os Nós Sem Predecessores?

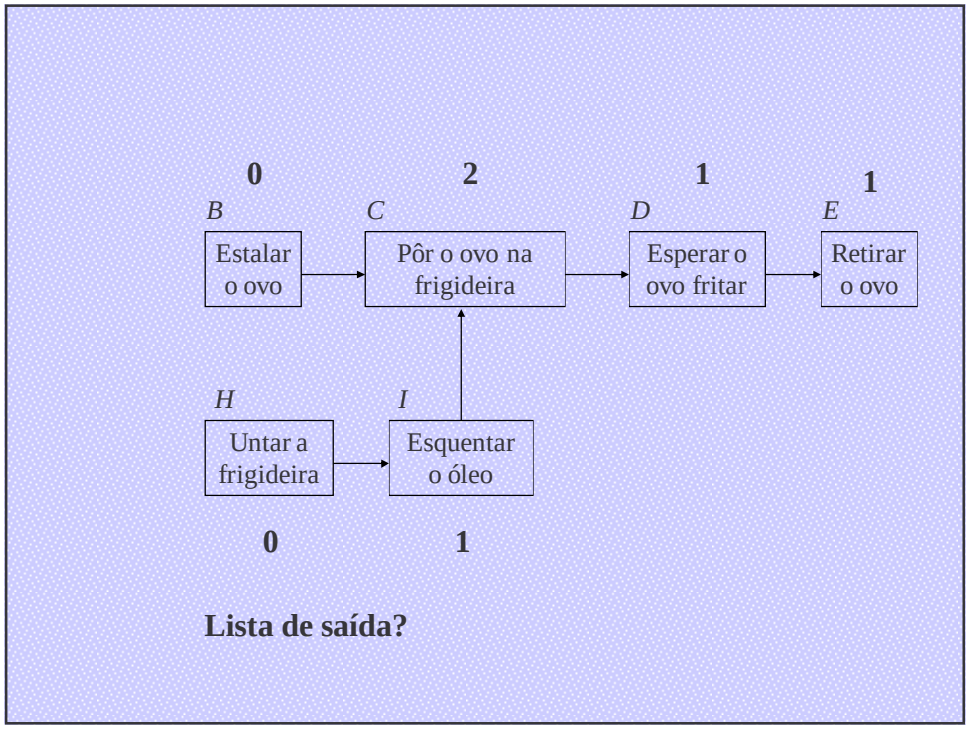
- Um método seria manter um campo *count* em cada nó contendo o numero de nós que o precedem.
- Inicialmente, depois que o grafo for construído, examinamos todos os nós do grafo e colocamos os nós com contagem 0 numa lista de saída.

Como Determinar os Nós Sem Predecessores? (cont.)

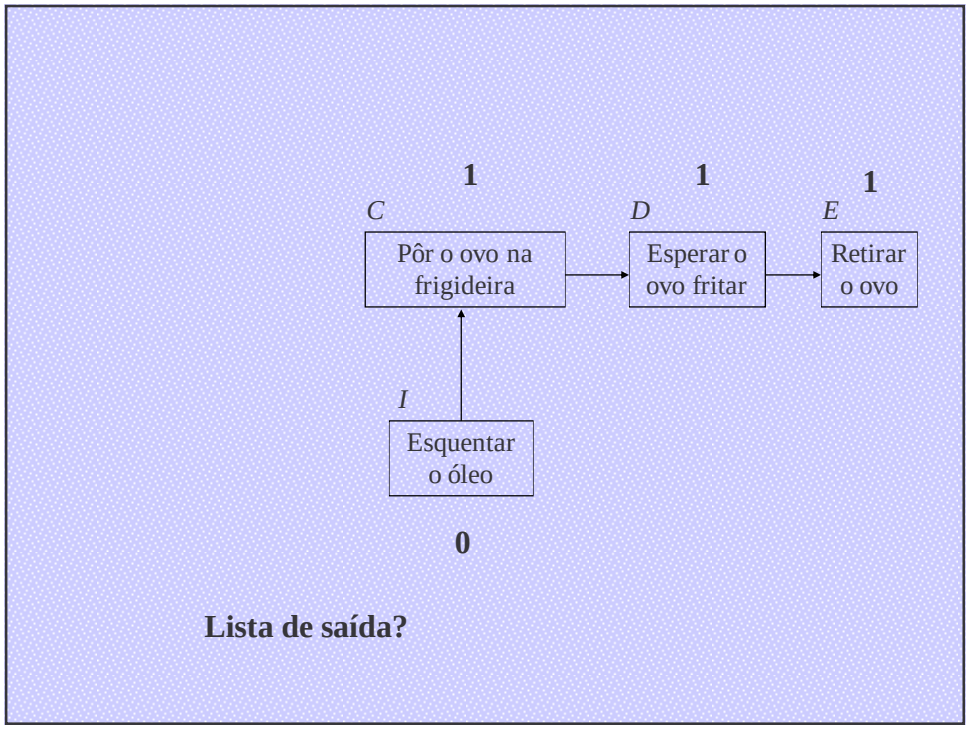
- Em seguida, durante cada período de tempo simulado, a lista de saída é percorrida, cada nó na lista é removido e a lista de adjacência de arcos emanando desse nó de grafo é percorrida.
- Para cada arco, a contagem no nó de grafo que encerra o arco é reduzida em 1 e, quando a contagem chegar a 0, o nó será colocado na lista de saída do próximo período de tempo. Ao mesmo tempo, o nó do arco é liberado.



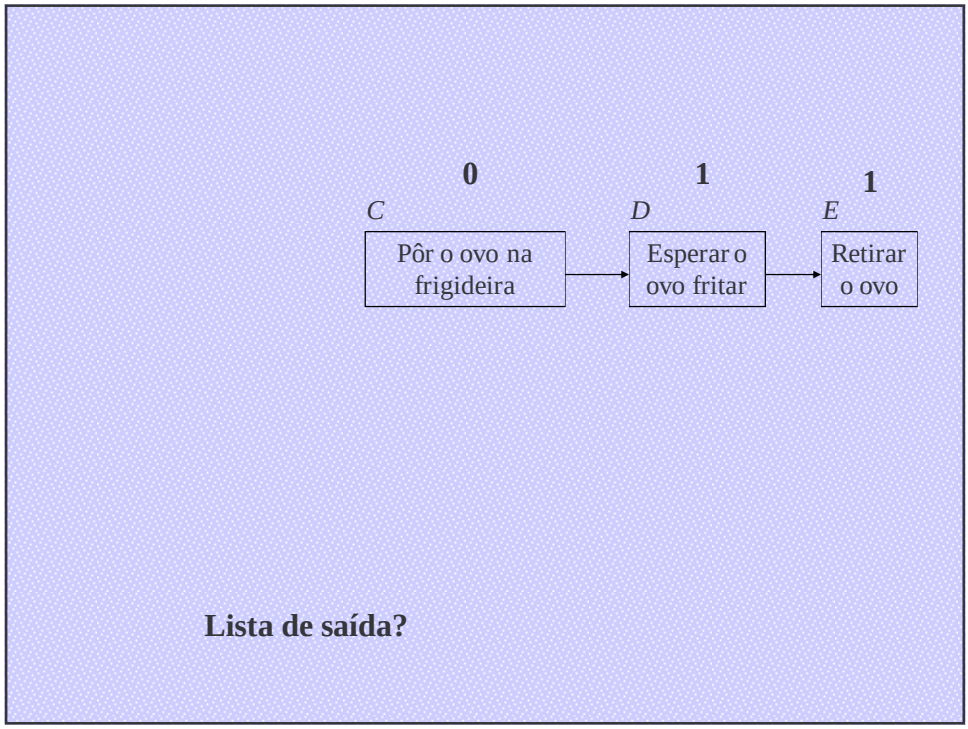
Lista de saída?

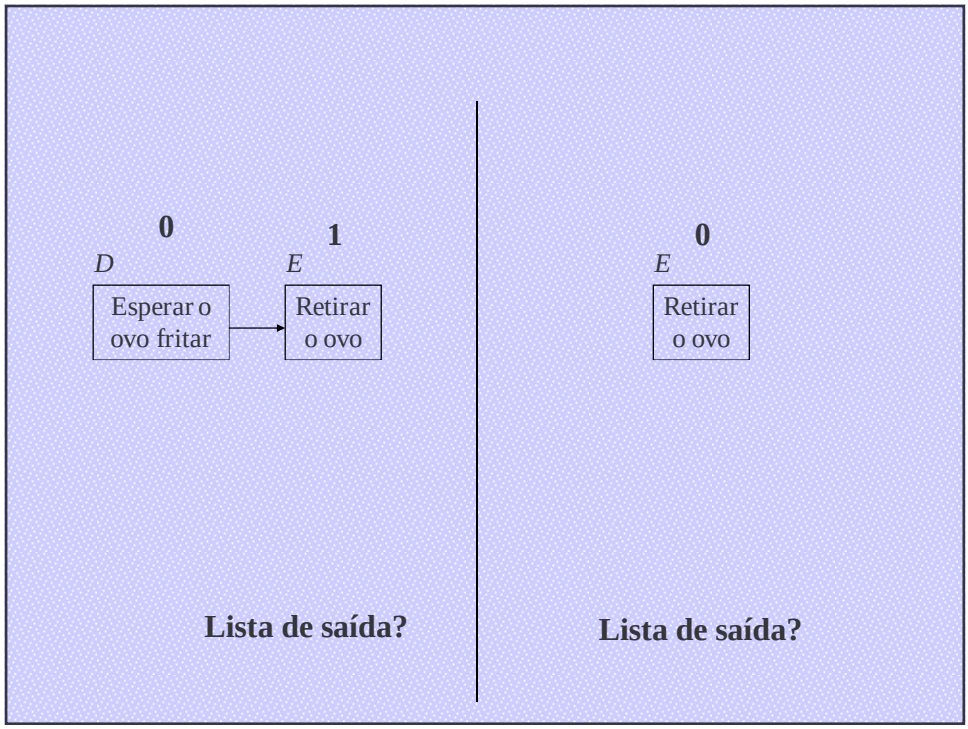


Lista de saída?



Lista de saída?


0
1
1
C
D
E
Pôr o ovo na frigideira
Esperar o ovo fritar
Retirar o ovo
Lista de saída?


0
1
D
E
Esperar o ovo fritar
Retirar o ovo
Lista de saída?
0
E
Retirar o ovo
Lista de saída?

O Detalhamento do Passo 2 Pode Ser Rescrito Assim:

```
/* atravessa o conj. de nós e coloca todos os nós com contagem 0 na
   lista de saída*/
outp = NULL;
for (todo node(p) no grafo)
    if (count(p) == 0) {
        remove node(p) do grafo;
        coloca node(p) na lista de saída;
    }
```

Cont.

```
/* simula os períodos de tempo */
period = 0;
while (outp != NULL) {
    ++period;
    printf("%d\n", period);
    imprime as tarefas executadas;
    /* inicializa lista de saída do prox. período de tempo */
    nextout = NULL;
    /* percorre a lista de saída */
    p = outp;
```

```
        while (p != NULL) {
              printf("%s", info(p));
              for (todos os arcos a emanando de node(p)) {
                    /* reduz count no nó final */
                    t = ponteiro p/ o nó que encerra a;
                    count(t)--;
                    if (count(t) == 0) {
                          remove node(t) do grafo;
                          inclui node(t) na lista de nextout;
                    }
                    free arc(a);
              }
              q = next(p);
              free(p);
              p = q;
        }  /*p != NULL */
        outp = nextout;
  }
  if (restar algum nó no grafo )
        error - existe um ciclo no grafo
```

Com Círculo