

GRAFOS E APLICAÇÕES

1. PERCURSO DE GRAFOS

- Frequentemente, 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 árvore por três razões:

- 1) Em geral, não existe um primeiro nó “natural” num grafo a partir do qual o percurso deva começar. Além 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.

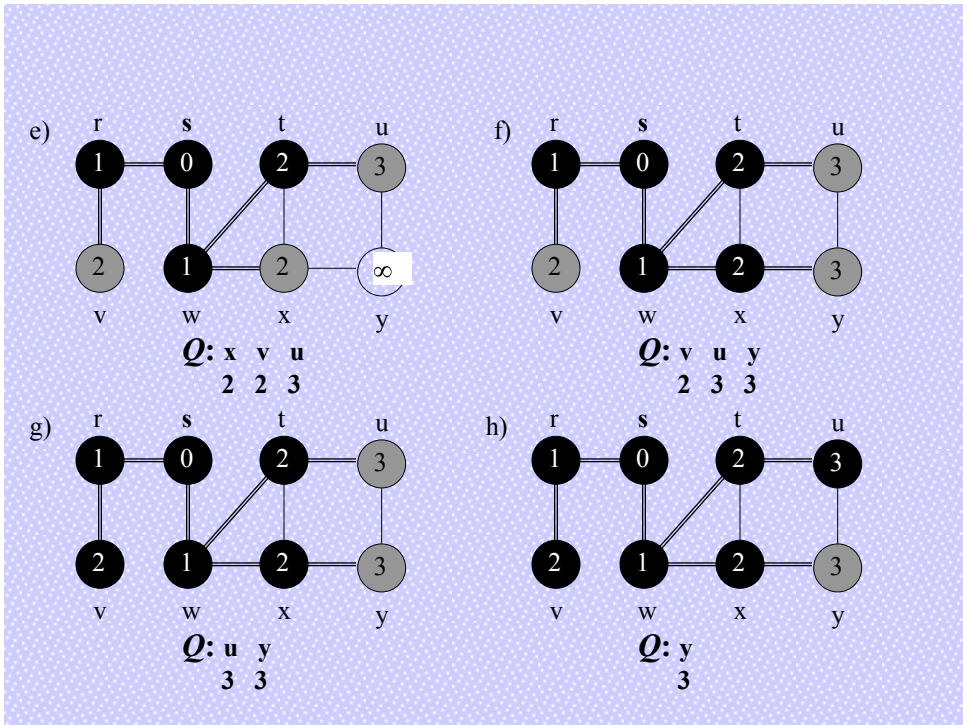
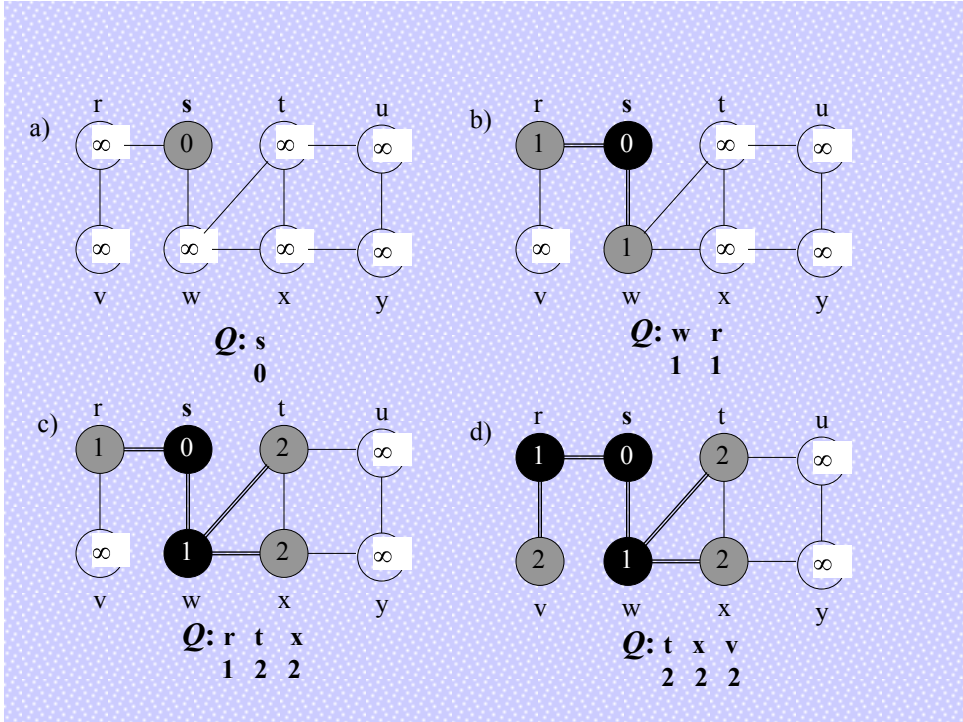
- 3) Um nó de um grafo pode ter mais de um predecessor. Portanto, é possível que um nó seja visitado antes de um de seus predecessores.

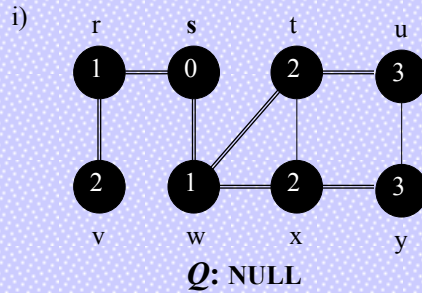
Percurso em Largura

- 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 último 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.





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;

$\pi(u)$: armazena o predecessor de u.

```

BFS(G, s)
1  for (cada vértice  $u \in V[G] - \{s\}$ ) {
2      color[u] = WHITE;
3      d[u] =  $\infty$ ;
4       $\pi[u] = \text{NIL}$ ;
5  }
5  color[s] = GRAY;
6  d[s] = 0;
7   $\pi[s] = \text{NIL}$ ;
8  Q  $\leftarrow$  {s}
9  while (Q  $\neq$  NULL) {
10     u = head[Q];
11     for (cada  $v \in \text{adj}[u]$ )
12         if (color[v] = WHITE) {
13             color[v] = GRAY;
14             d[v]++;
15              $\pi[v] = u$ ;
16             ENQUENE(Q, v);
17         }
17     DEQUENE(Q);
18     color[u] = BLACK;
19 }

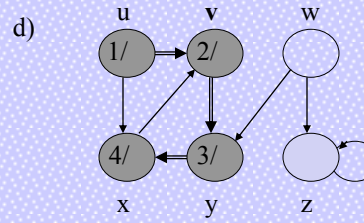
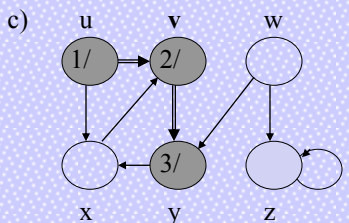
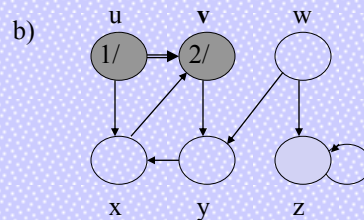
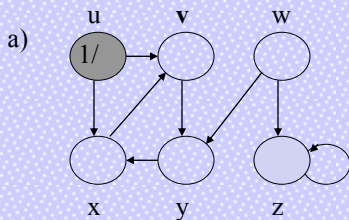
```

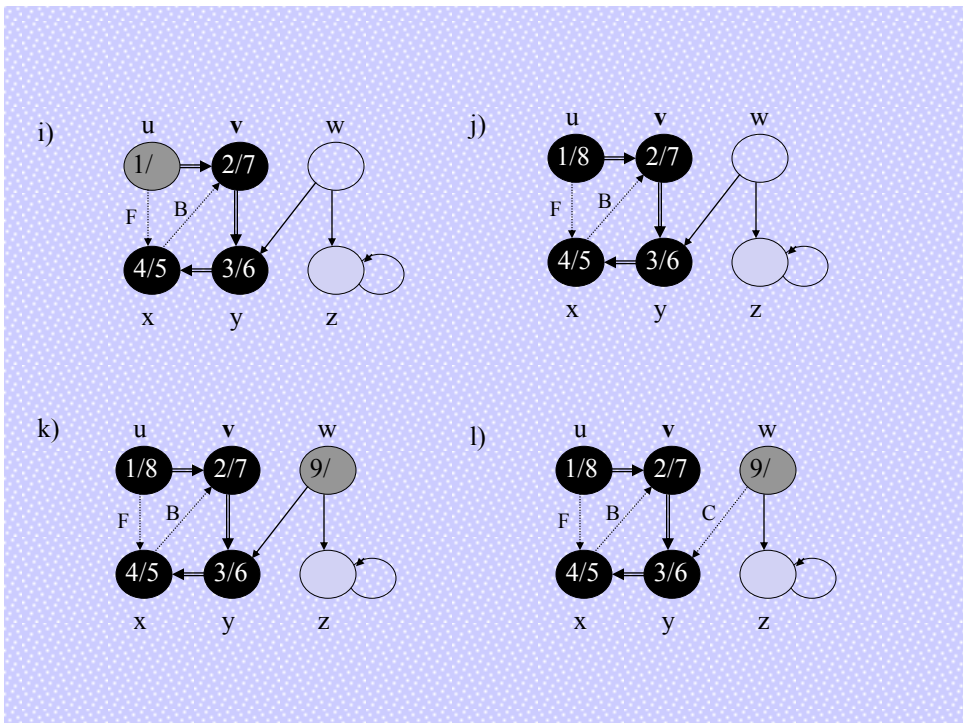
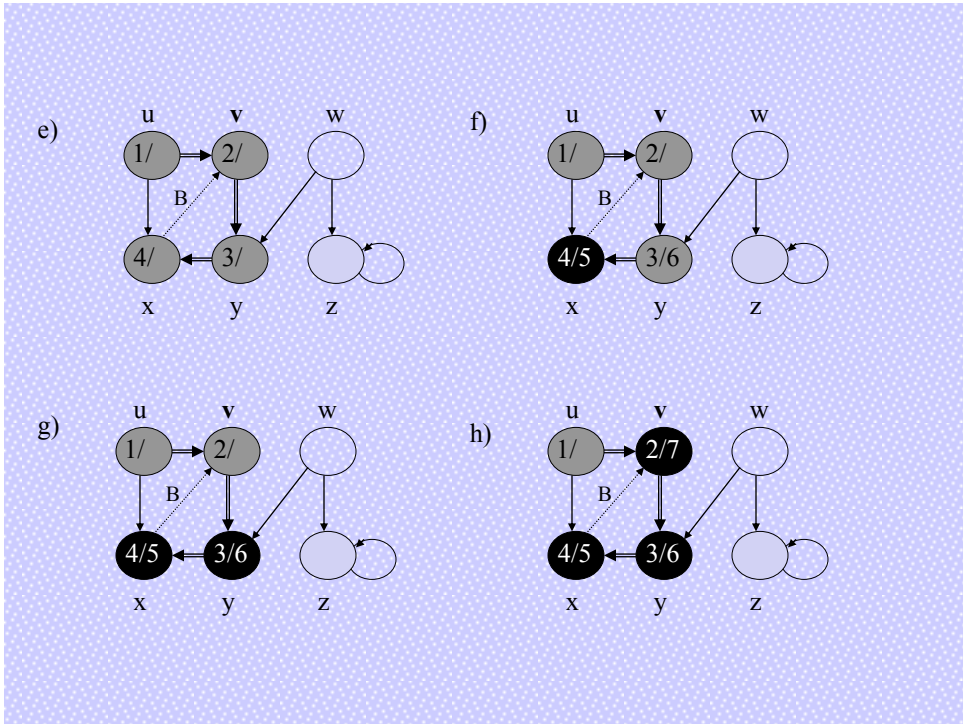
Percurso em Profundidade

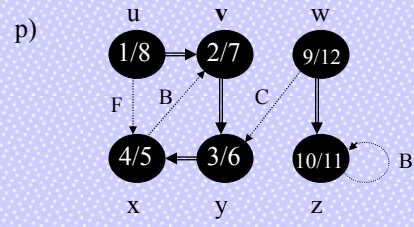
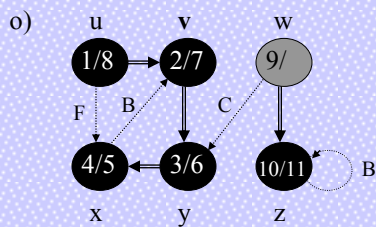
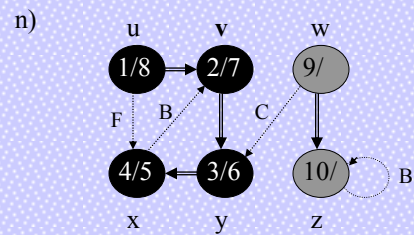
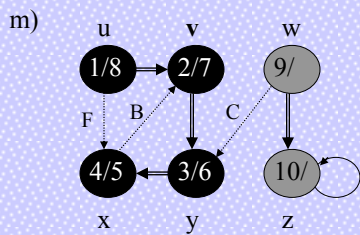
- Um percurso em profundidade, como o próprio nome indica, percorre um único caminho do grafo até onde ele possa chegar, isto é, até visitar um nó sem sucessores ou um nó cujos sucessores já tenham sido visitados. Em seguida, ele continua no último nó no caminho recém-percorrido que tenha um sucessor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







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  $u \in V[G]$  {
2      color[u] = WHITE;
3       $\pi[u] = \text{NIL}$ ;
4  }
4  time = 0;
5  for cada vértice  $u \in V[G] - \{s\}$ 
6      if (color[u] == WHITE)
7          DFS_VISIT(u);

```

```

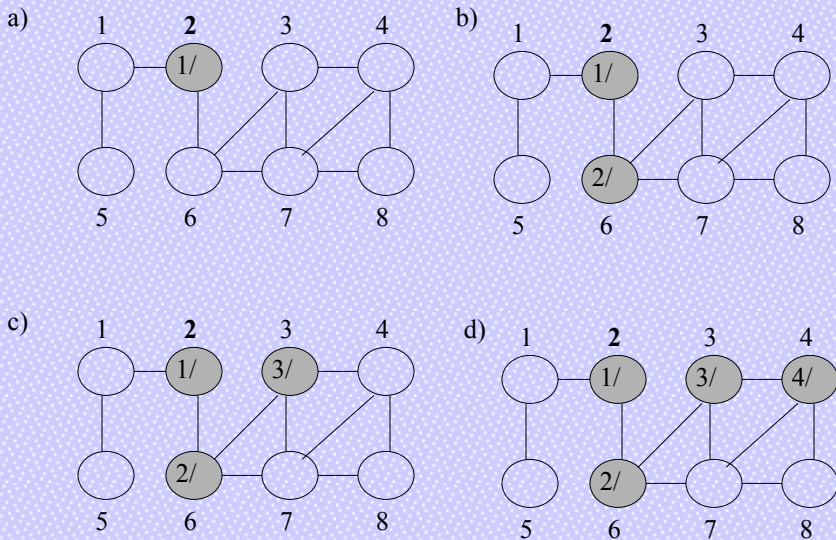
DFS_VISIT(u)
1  color[u] = GRAY;
2  d[u] = time = time++;
3  for cada  $v \in \text{adj}[u]$ 
4      if (color[v] == WHITE) {
5           $\pi[v] = u$ ;
6          DFS_VISIT(v);
7      }
7  color[u] = BLACK;
8  f[u] = time = time++;

```

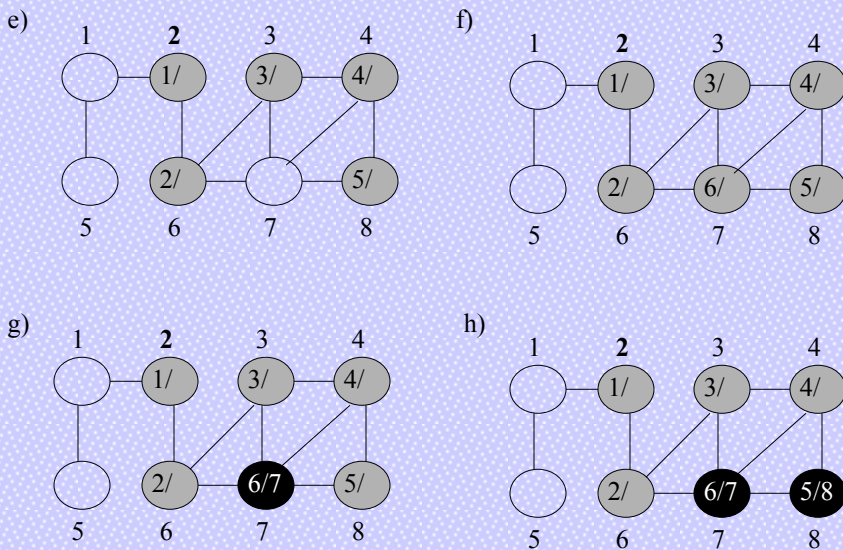
Eficiência do Algoritmo:

- 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 número de arestas no grafo.

Percurso em Profundidade – Grafo não direcionado

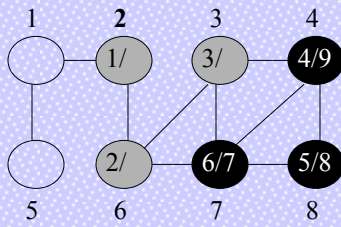


Percurso em Profundidade – Grafo não direcionado

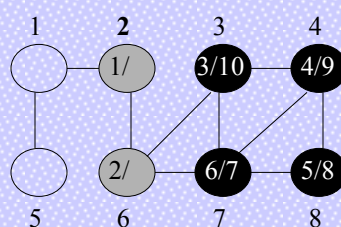


Percurso em Profundidade – Grafo não direcionado

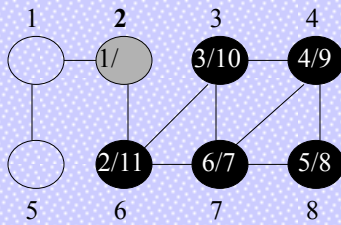
i)



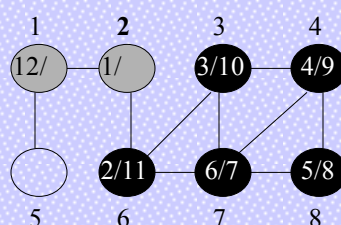
j)



k)

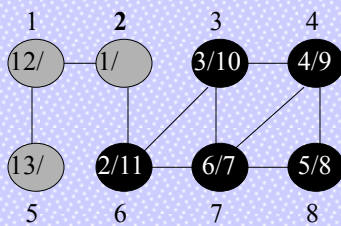


l)

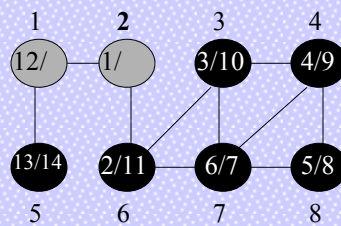


Percurso em Profundidade – Grafo não direcionado

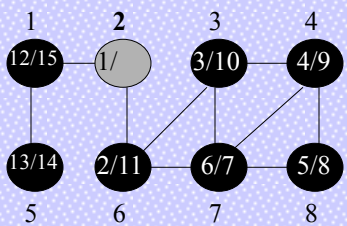
m)



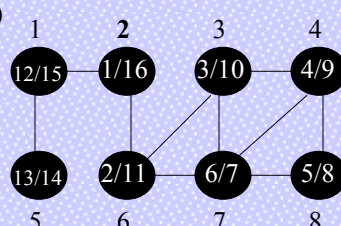
n)



o)

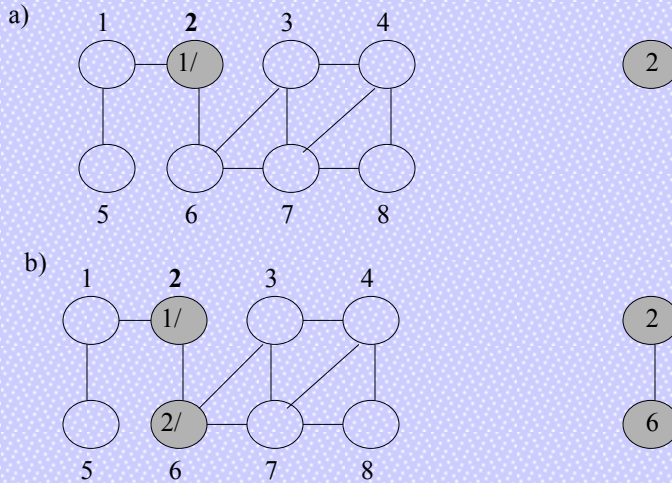


p)

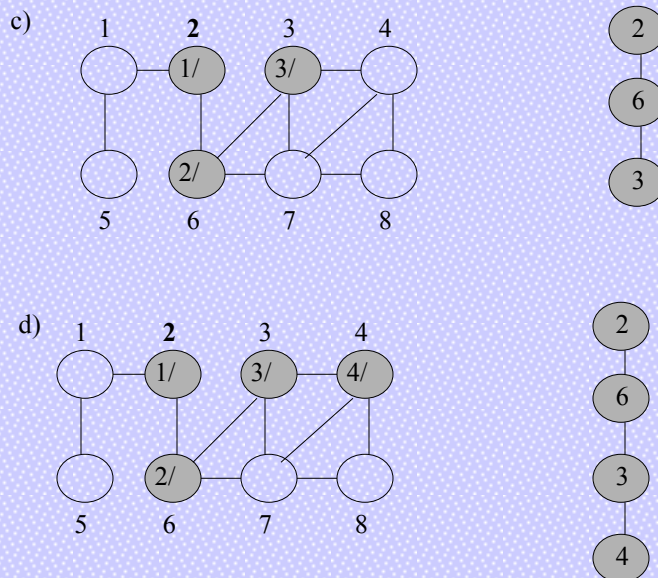


Árvore de Percurso em Profundidade

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

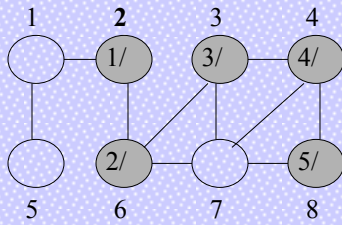


Árvore de Percurso em Profundidade



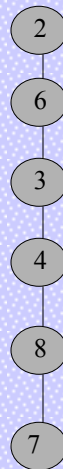
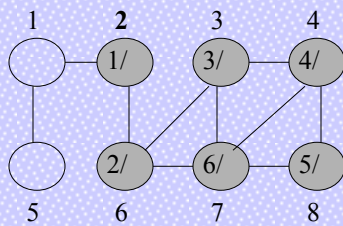
Árvore de Percurso em Profundidade

e)



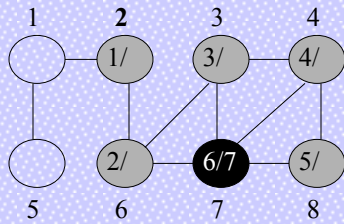
Árvore de Percurso em Profundidade

f)



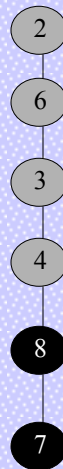
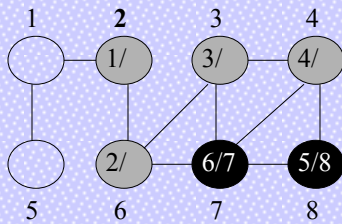
Árvore de Percurso em Profundidade

g)



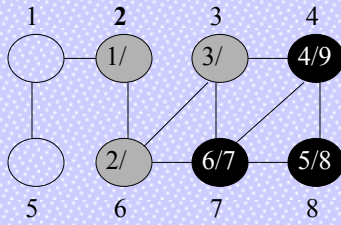
Árvore de Percurso em Profundidade

h)



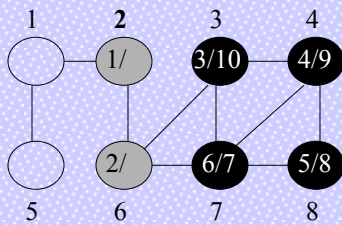
Árvore de Percurso em Profundidade

i)



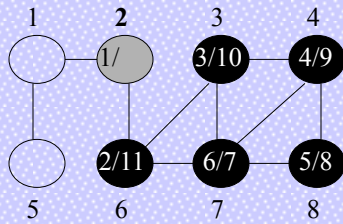
Árvore de Percurso em Profundidade

j)



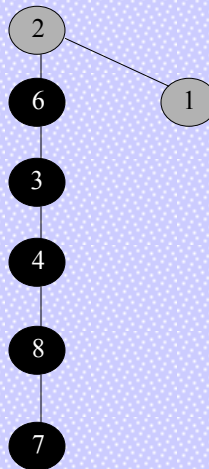
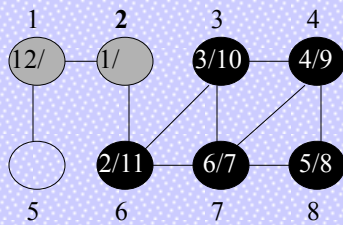
Árvore de Percurso em Profundidade

k)



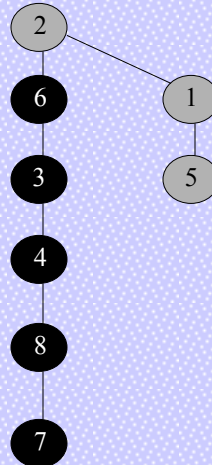
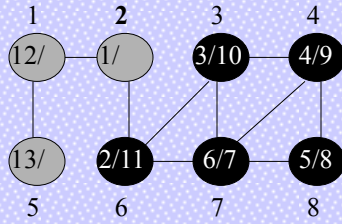
Árvore de Percurso em Profundidade

l)



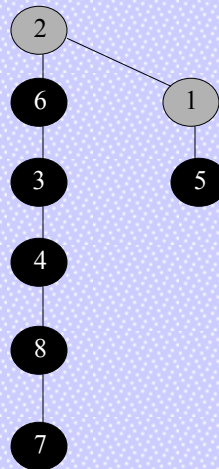
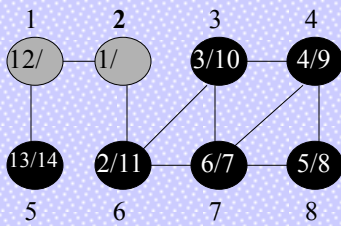
Árvore de Percurso em Profundidade

m)



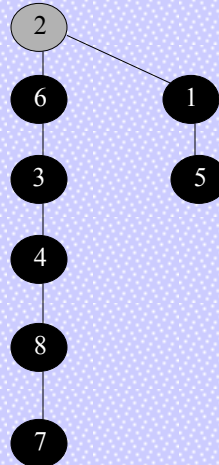
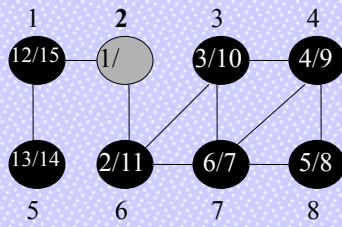
Árvore de Percurso em Profundidade

n)



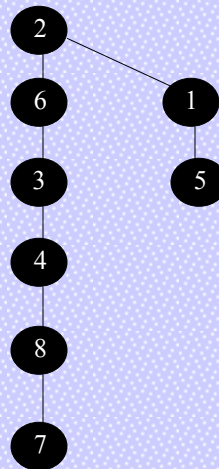
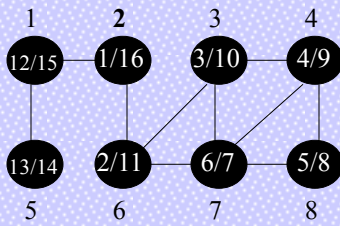
Árvore de Percurso em Profundidade

o)



Árvore de Percurso em Profundidade

p)

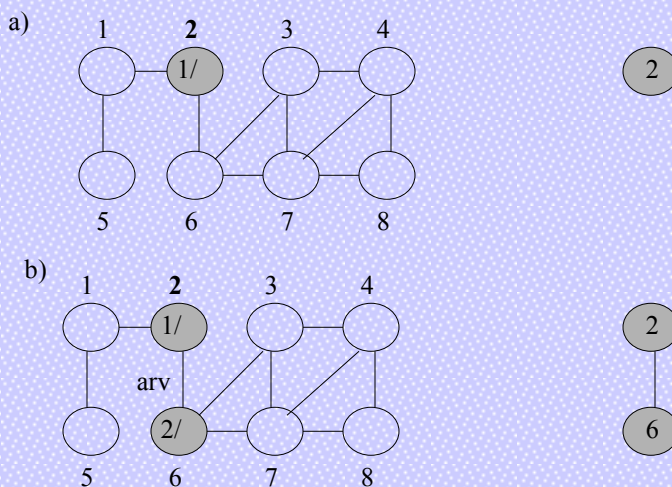


Classificação de Arestas

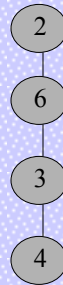
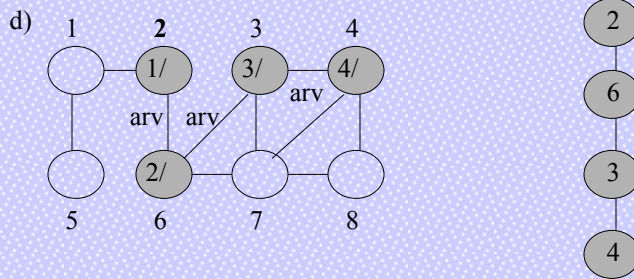
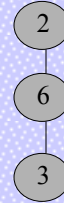
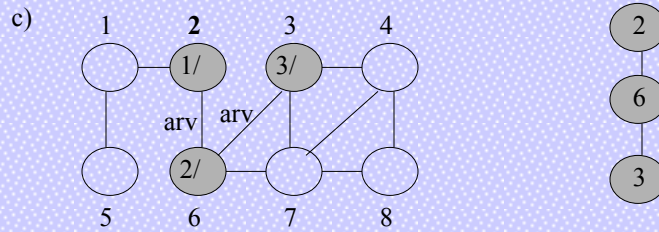
As arestas de um grafo podem ser classificadas conforme a sua árvore de busca em profundidade:

- **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.

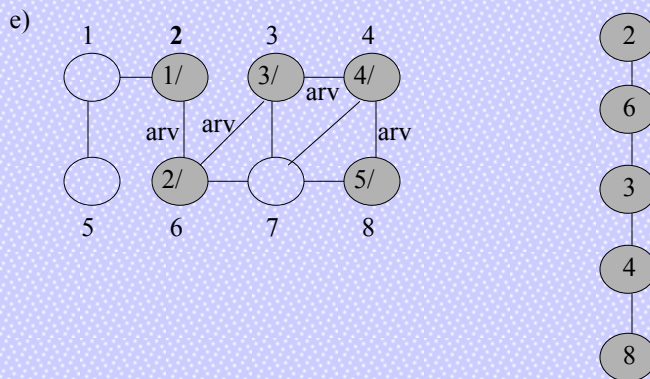
Árvore de Percurso em Profundidade



Árvore de Percurso em Profundidade

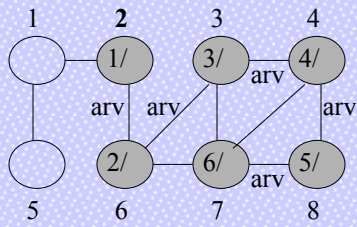


Árvore de Percurso em Profundidade



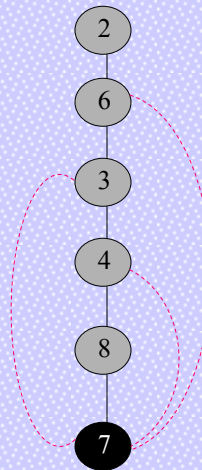
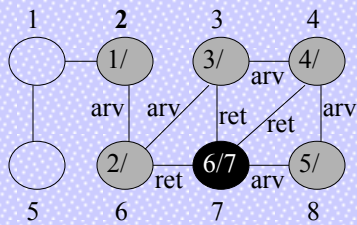
Árvore de Percurso em Profundidade

f)



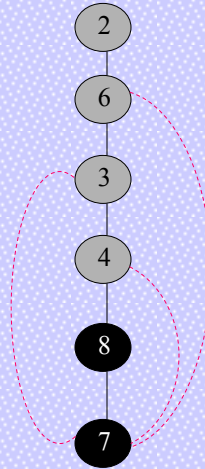
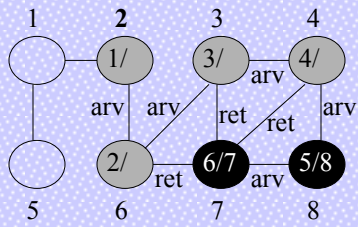
Árvore de Percurso em Profundidade

g)



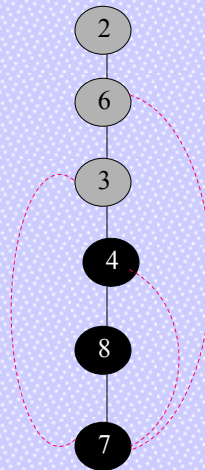
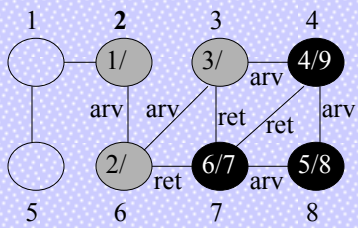
Árvore de Percurso em Profundidade

h)



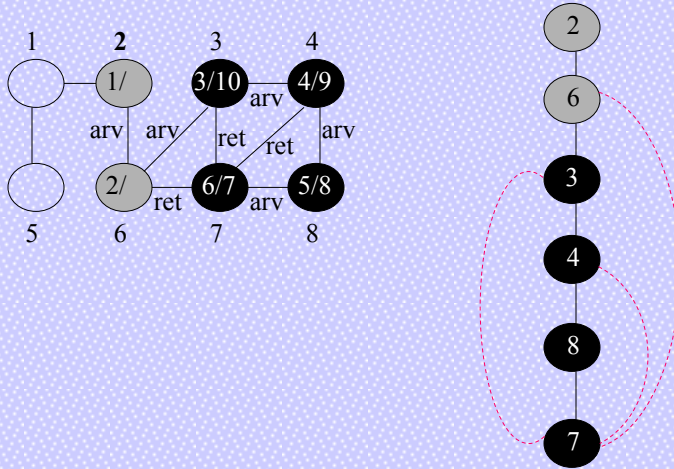
Árvore de Percurso em Profundidade

i)



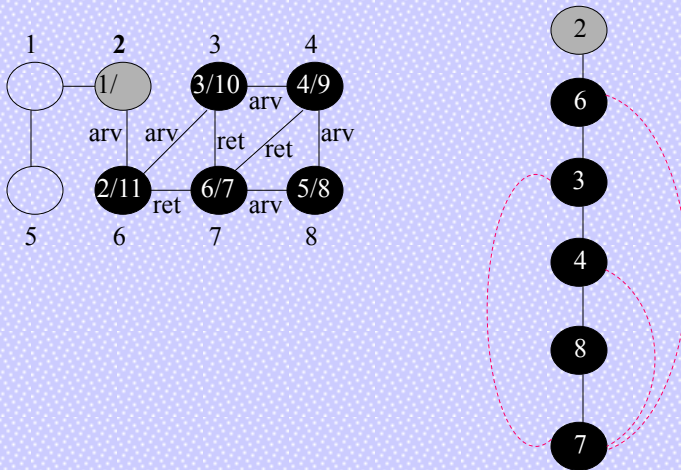
Árvore de Percurso em Profundidade

j)

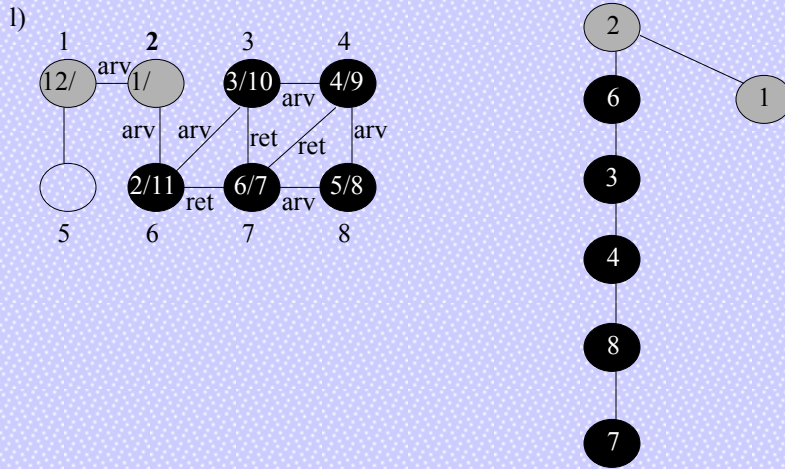


Árvore de Percurso em Profundidade

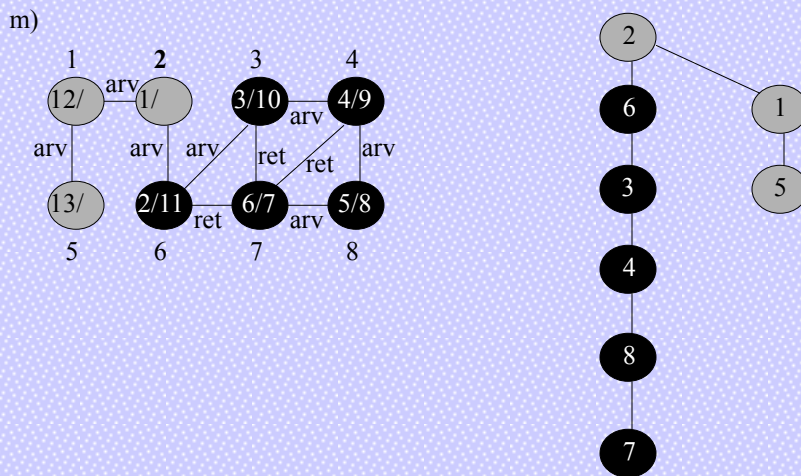
k)



Árvore de Percurso em Profundidade

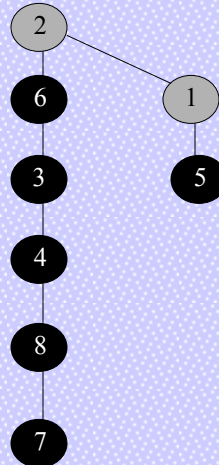
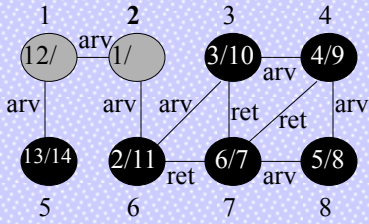


Árvore de Percurso em Profundidade



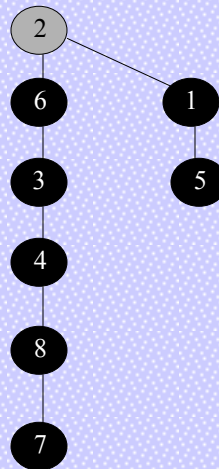
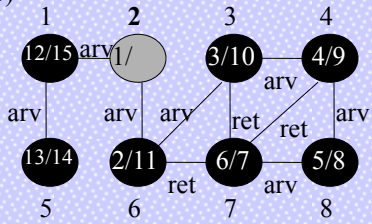
Árvore de Percurso em Profundidade

n)

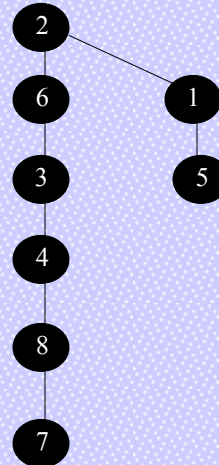
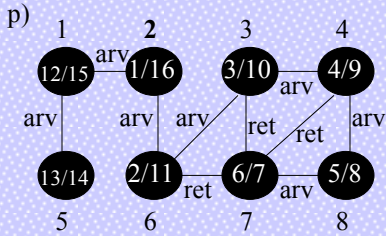


Árvore de Percurso em Profundidade

o)

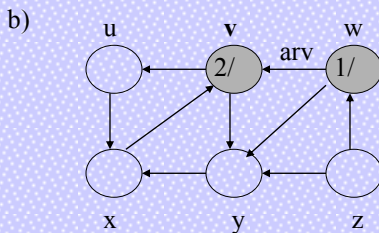
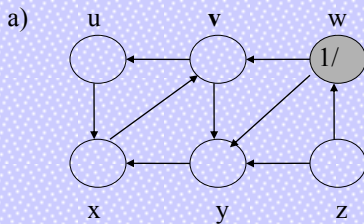


Árvore de Percurso em Profundidade

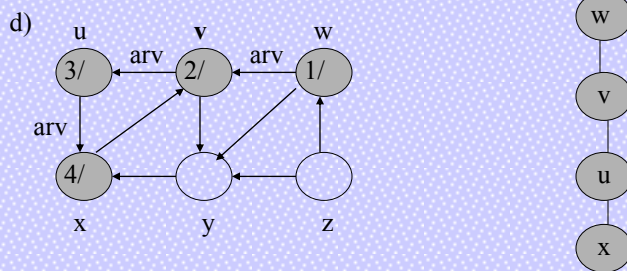
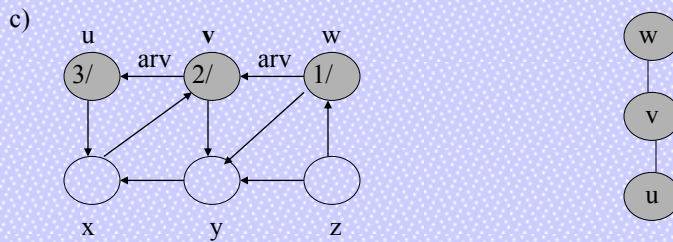


Em um grafo não orientado, todas as arestas são de árvore ou de retorno.

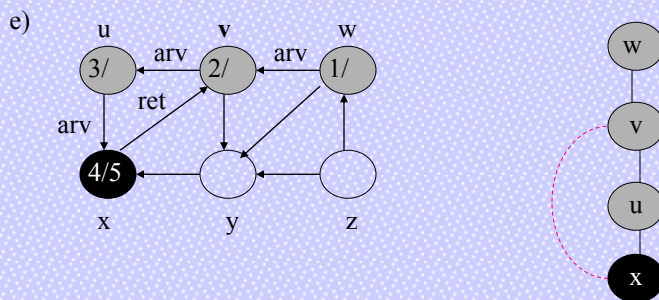
Árvore de Percurso em Profundidade - dígrafo



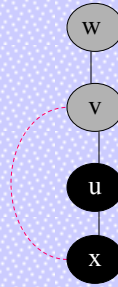
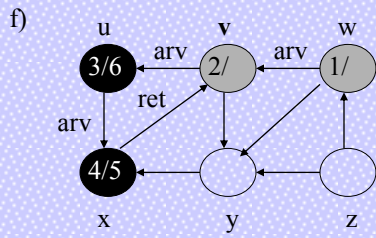
Árvore de Percurso em Profundidade - dígrafo



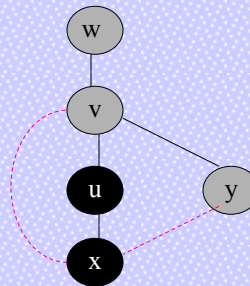
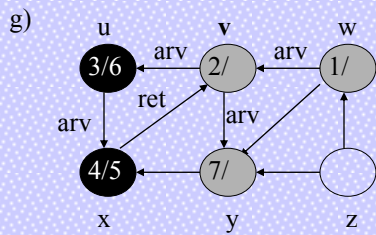
Árvore de Percurso em Profundidade - dígrafo



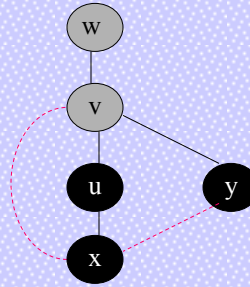
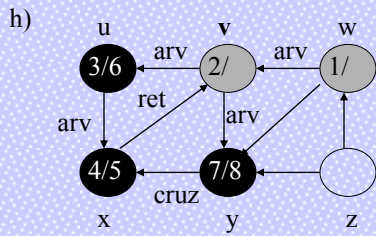
Árvore de Percurso em Profundidade - dígrafo



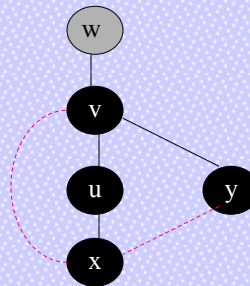
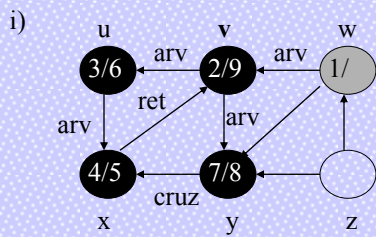
Árvore de Percurso em Profundidade - dígrafo



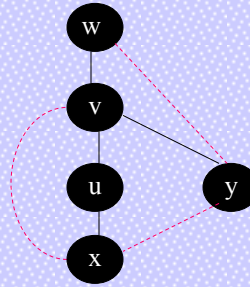
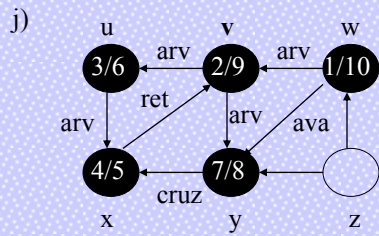
Árvore de Percurso em Profundidade - dígrafo



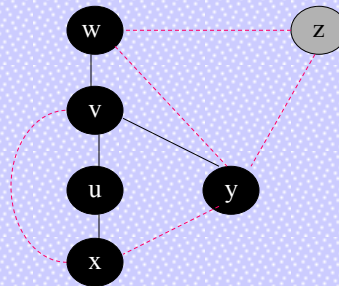
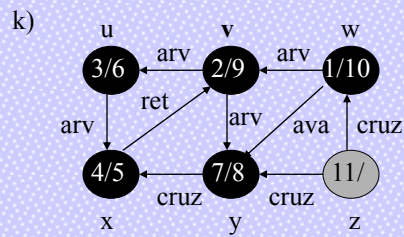
Árvore de Percurso em Profundidade - dígrafo



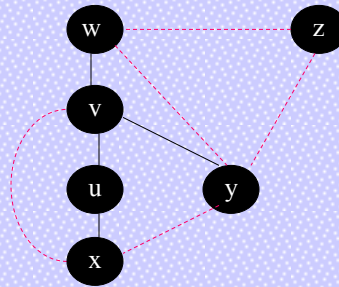
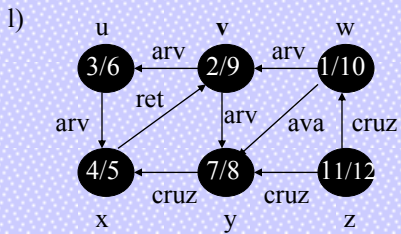
Árvore de Percurso em Profundidade - dígrafo



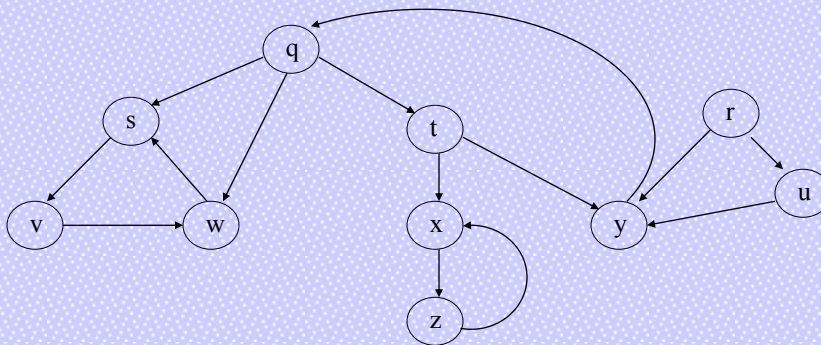
Árvore de Percurso em Profundidade - dígrafo



Árvore de Percurso em Profundidade - dígrafo



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.

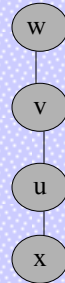
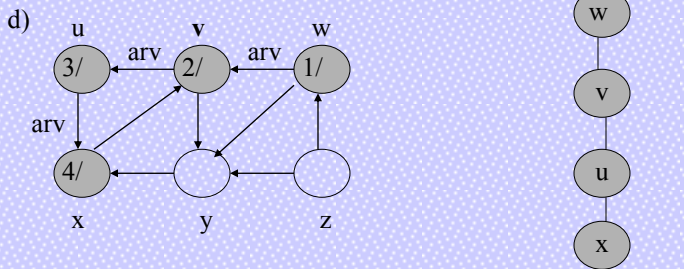
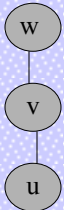
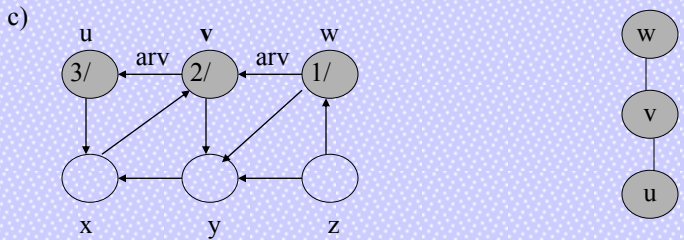
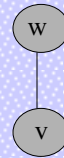
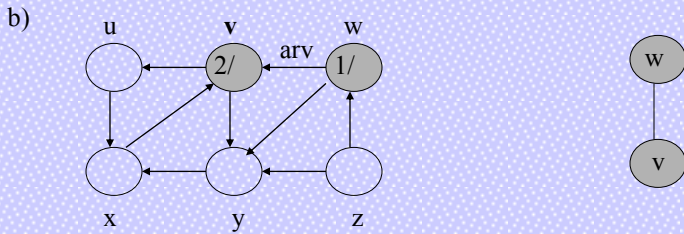
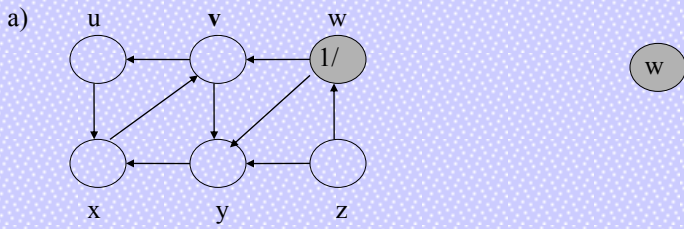
Aplicações de Percurso em Profundidade

Teste para Verificar se Grafo é Acíclico

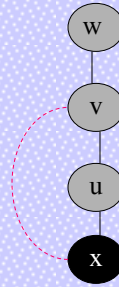
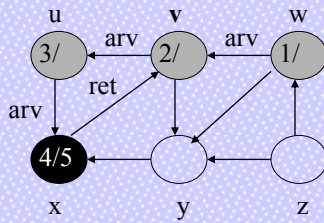
- Percurso em profundidade pode ser usada para verificar se um grafo é acíclico ou contém um ou mais ciclos.
- Se uma aresta de retorno é encontrada durante o percurso em profundidade em G , então o grafo tem ciclo.

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.



e)



O grafo é cíclico

Ordenação Topológica

- 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

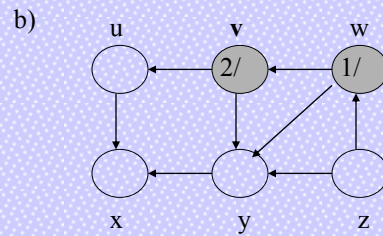
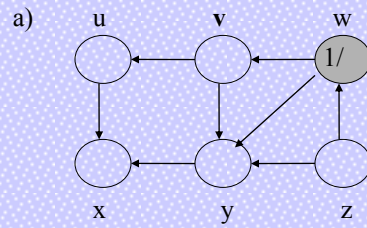
- A ordenação topológica é uma ordenação linear de todos os vértices, tal que se G contém uma aresta (u,v) então u aparece antes de v .
- Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita.

Ordenação Topológico

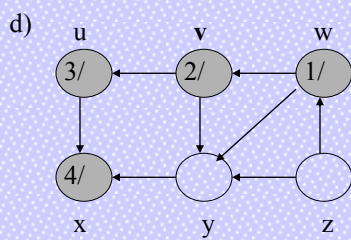
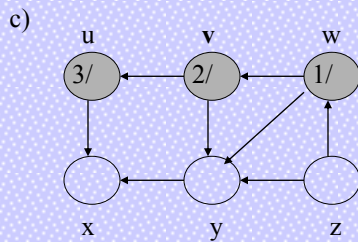
TOPOLOGICAL-SORT(G)

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.

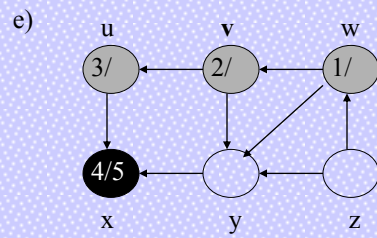
Ordenação Topológica



Ordenação Topológica

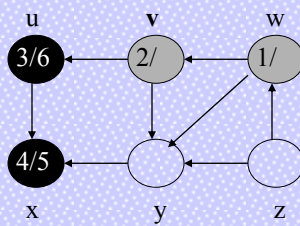


Ordenação Topológica



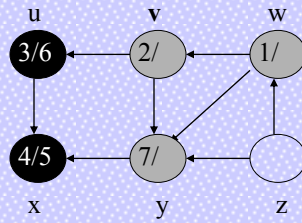
Ordenação Topológica

f)



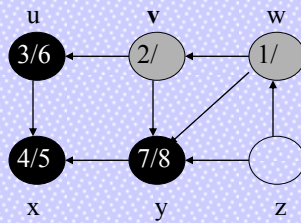
Ordenação Topológica

g)



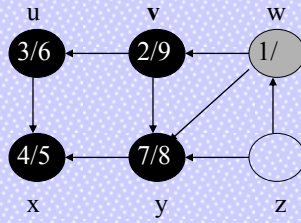
Ordenação Topológica

h)



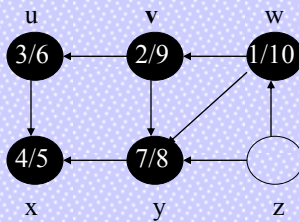
Ordenação Topológica

i)



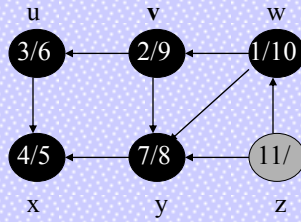
Ordenação Topológica

j)



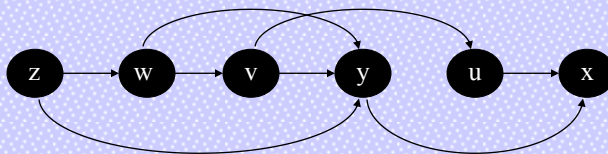
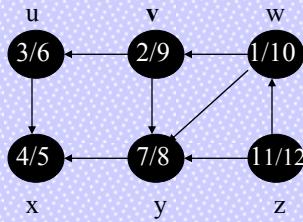
Ordenação Topológica

k)

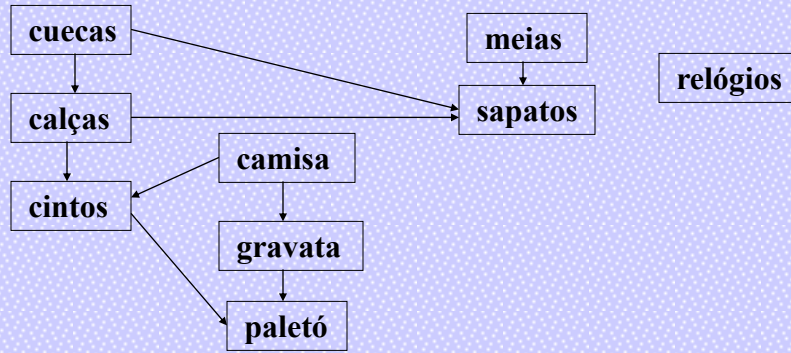


Ordenação Topológica

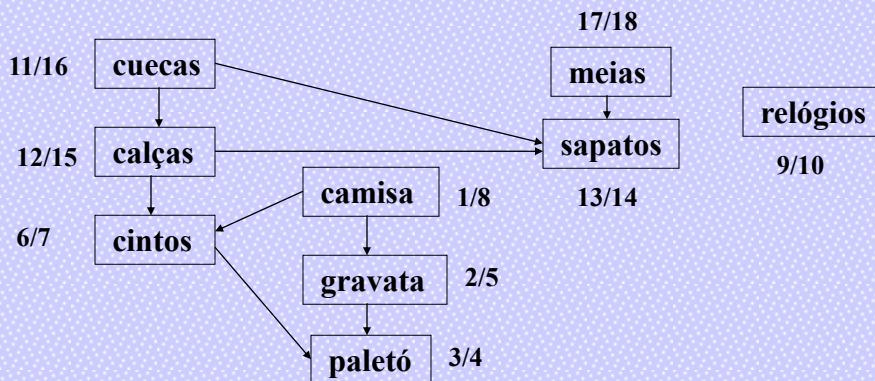
l)



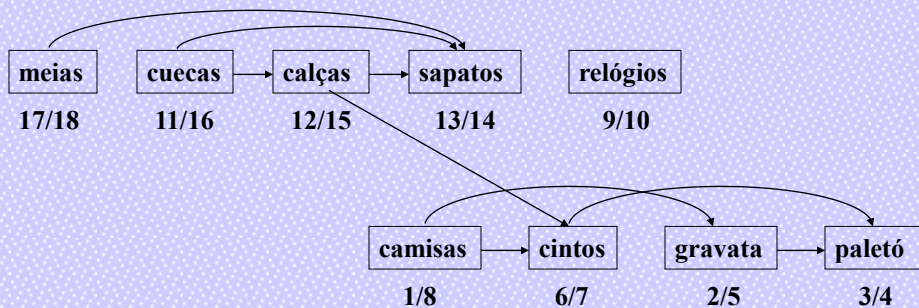
Ordenação Topológica



Ordenação Topológica



Ordenação Topológica



Ordenação Topológica – 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.

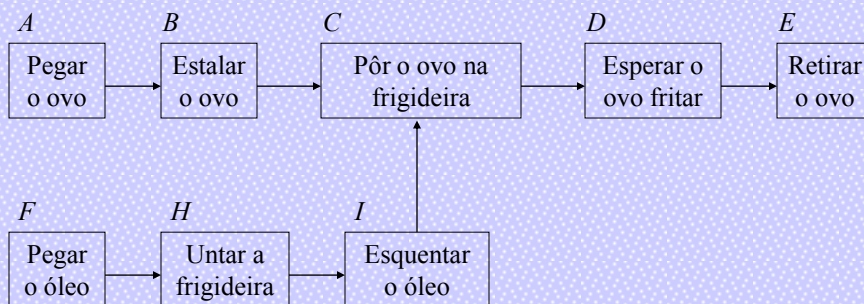
Aplicações Relevantes

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 de desempenho; o compilador pode precisar escalonar operações em linguagem de máquina para reduzir o tempo de execução; ou o gerente de uma fábrica precisa organizar uma linha 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 $\langle x, y \rangle$ 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 contém um ciclo, deve existir pelo menos um nó em G sem antecessores. No grafo anterior, os nós A e F não têm predecessores;
- Assim, as subtarefas que eles representam podem ser executadas imediatamente e simultaneamente, sem esperar o término de nenhuma outra tarefa. Toda tarefa adicional precisará esperar até que pelo menos uma dessas tarefas termine.

Solução do Problema (cont.)

- Assim que essas 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:

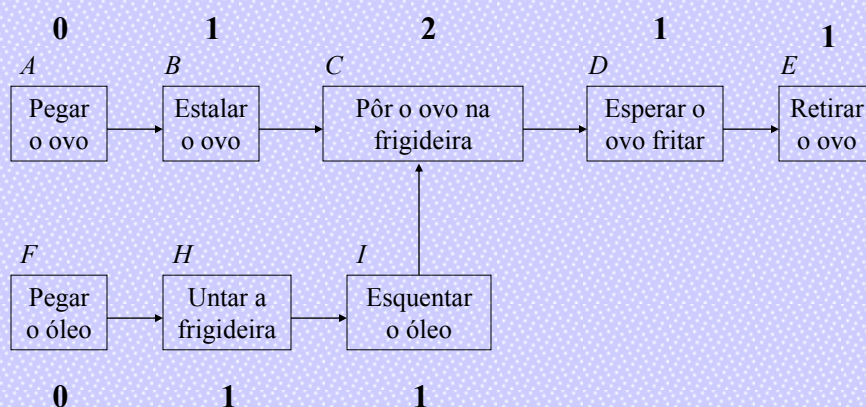
```
while (o grafo não está vazio) {  
  i)   determina os nós sem predecessores;  
  ii)  dá saída nesse grupo de nós com indicação de que eles  
        podem ser executados simultaneamente no próximo  
        período de tempo;  
  iii) remover estes nós e seus arcos incidentes do grafo;  
}
```

Como Determinar os Nós Sem Predecessores?

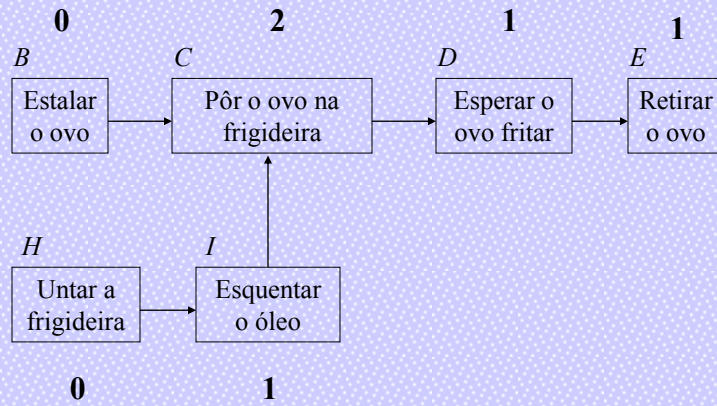
- Um método seria manter um campo *count* em cada nó contendo o número de nós que o precedem.
- Inicialmente, depois que o grafo for construído, examinamos todos os nós do grafo e colocamos os nós com contagem 0 numa lista de saída.

Como Determinar os Nós Sem Predecessores? (cont.)

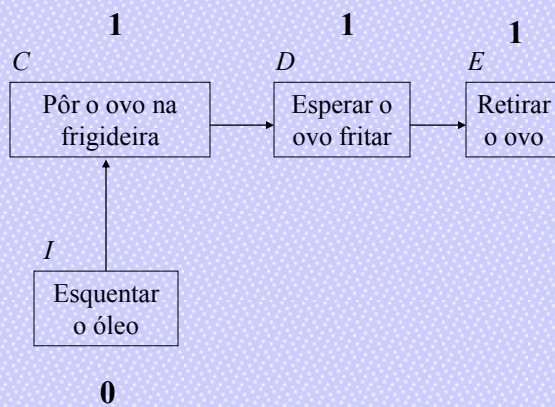
- Em seguida, durante cada período de tempo simulado, a lista de saída é percorrida, cada nó na lista é removido e a lista de adjacência de arcos emanando desse nó de grafo é percorrida.
- Para cada arco, a contagem no nó de grafo que encerra o arco é reduzida em 1 e, quando a contagem chegar a 0, o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



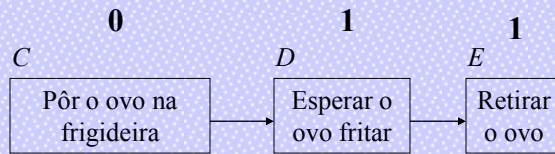
Lista de saída?



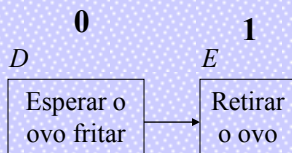
Lista de saída?



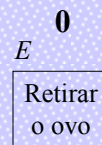
Lista de saída?



Lista de saída?



Lista de saída?



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

