# MO417 — Complexidade de Algoritmos I

Cid Carvalho de Souza  Cândida Nunes da Silva
Orlando Lee

7 de outubro de 2008



Buscas em grafos



## Busca em grafos

- Grafos são estruturas mais complicadas do que listas, vetores e árvores (binárias).
  Precisamos de métodos para explorar/percorrer um grafo (orientado ou não-orientado).
- Busca em largura (breadth-first search)
  Busca em profundidade (depth-first search)
- Pode-se obter várias informações sobre a estrutura do grafo que podem ser úteis para projetar algoritmos eficientes para determinados problemas.



## Notação

- Para um grafo $G$ (orientado ou não) denotamos por $V[G]$ seu conjunto de vértices e por $E[G]$ seu conjunto de arestas.
- Para denotar complexidades nas expressões com $O$ ou $\Theta$ usaremos $V$ e $E$ em vez de $|V[G]|$ ou $|E[G]|$. Por exemplo, $\Theta(V+E)$ ou $O(V^2)$.

## Busca em largura

- Dizemos que um vértice $v$ é **alcançável** a partir de um vértice $s$ em um grafo $G$ se existe um caminho de $s$ a $v$ em $G$.
- Definição: a distância de $s$ a $v$ é o comprimento de um caminho mais curto de $s$ a $v$.
- Se $v$ **não é alcançável** a partir de $s$, então dizemos que a distância de $s$ a $v$ é $\infty$ (*infinita*).



## Busca em largura

- Busca em largura recebe um grafo $G = (V, E)$ e um vértice especificado $s$ chamado fonte (*source*).
- Percorre todos os vértices alcançáveis a partir de $s$ em ordem de distância deste. Vértices a mesma distância podem ser percorridos em qualquer ordem.
- Constrói uma Árvore de Busca em Largura com raiz $s$. Cada caminho de $s$ a um vértice $v$ nesta árvore corresponde a um caminho mais curto de $s$ a $v$.



## Busca em largura

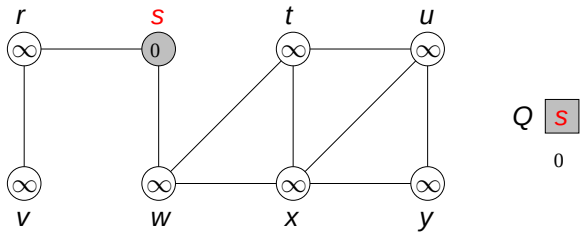
- Inicialmente a Árvore de Busca em Largura contém apenas o vértice fonte $s$.
- Para cada vizinho $v$ de $s$, o vértice $v$ e a aresta $(s, v)$ são acrescentadas à árvore.
- O processo é repetido para os vizinhos dos vizinhos de $s$ e assim por diante, até que todos os vértices atingíveis por $s$ sejam inseridos na árvore.
- Este processo é implementado através de uma fila $Q$.



## Busca em largura

- Busca em largura atribui cores a cada vértice: branco, cinza e preto.
- Cor branca $=$ “não visitado”.
  Inicialmente todos os vértices são brancos.
- Cor cinza $=$ “visitado pela primeira vez”.
- Cor Preta $=$ “teve seus vizinhos visitados”.

## Exemplo (CLRS)

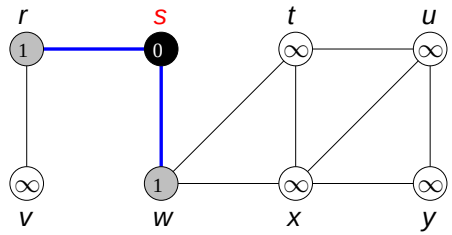



Q | s

0



## Exemplo (CLRS)

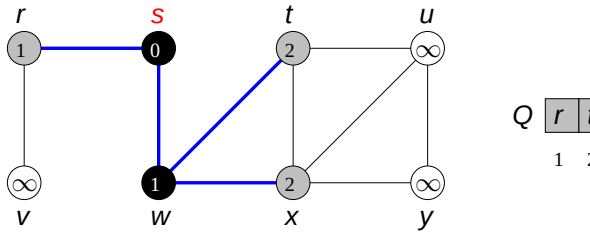



Q | w | r

1 1



## Exemplo (CLRS)

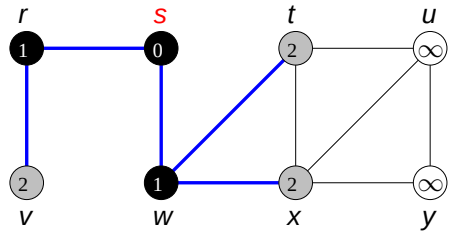



Q | r | t | x

1 2 2



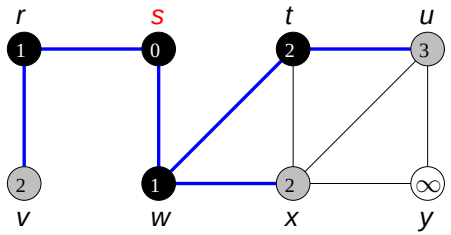
## Exemplo (CLRS)




Q | t | x | v

2 2 2

## Exemplo (CLRS)

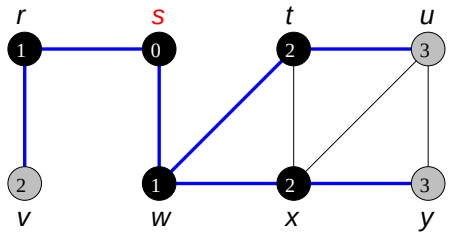



Q | x | v | u

2 2 3



## Exemplo (CLRS)

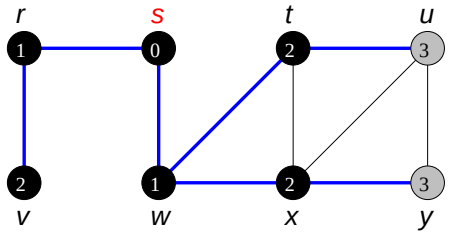



Q | v | u | y

2 3 3



## Exemplo (CLRS)

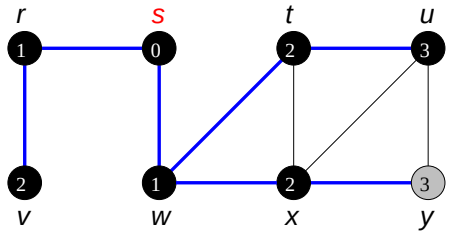



Q | u | y

3 3



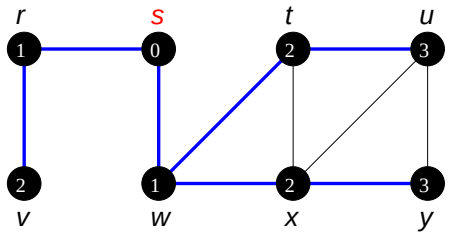
## Exemplo (CLRS)




Q | y

3

## Exemplo (CLRS)




$Q$ $\emptyset$



## Cores

- Para cada vértice $v$ guarda-se sua cor atual cor[$v$] que pode ser branco, cinza ou preto.
- Para efeito de implementação, isto não é realmente necessário, mas facilita o entendimento do algoritmo.



## Representação da árvore e das distâncias

- A raiz da Árvore de Busca em Largura é $s$.
- Cada vértice $v$ (diferente de $s$) possui um pai $\pi[v]$.
- O caminho de $s$ a $v$ na Árvore é dado por:

  $v, \pi[v], \pi[\pi[v]], \pi[\pi[\pi[v]]], \ldots, s$.

- Uma variável $d[v]$ é usada para armazenar a distância de $s$ a $v$ (que será determinada durante a busca).



## Busca em largura

Recebe um grafo $G$ (na forma de listas de adjacências) e um vértice $s \in V[G]$ e devolve
(i) para cada vértice $v$, a distância de $s$ a $v$ em $G$ e
(ii) uma Árvore de Busca em Largura.

BUSCA-EM-LARGURA$(G, s)$
0 ▷ Inicialização
1 **para cada** $u \in V[G] - \{s\}$ **faça**
2 cor$[u] \leftarrow$ branco
3 $d[u] \leftarrow \infty$
4 $\pi[u] \leftarrow$ NIL
5 cor$[s] \leftarrow$ cinza
6 $d[s] \leftarrow 0$
7 $\pi[s] \leftarrow$ NIL

## Busca em largura

```
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  enquanto Q ≠ ∅ faça
11      u ← DEQUEUE(Q)
12      para cada v ∈ Adj[u] faça
13          se cor[v] = branco então
14              cor[v] ← cinza
15              d[v] ← d[u] + 1
16              π[v] ← u
17              ENQUEUE(Q, v)
18      cor[u] ← preto
```



## Consumo de tempo

Método de análise agregado.

- A inicialização consome tempo $\Theta(V)$.
- Depois que um vértice deixa de ser branco, ele não volta a ser branco novamente. Assim, cada vértice é inserido na fila $Q$ no máximo uma vez. Cada operação sobre a fila consome tempo $\Theta(1)$ resultando em um total de $O(V)$.
- Em uma lista de adjacência, cada vértice é percorrido apenas uma vez. A soma dos comprimentos das listas é $\Theta(E)$. Assim, o tempo gasto para percorrer as listas é $O(E)$.



## Complexidade de tempo

Conclusão:

A complexidade de tempo de BUSCA-EM-LARGURA é $O(V + E)$.

Agora falta mostrar que BUSCA-EM-LARGURA funciona.



## Corretude

Para $u, v \in E[G]$, seja $dist(u, v)$ a distância de $u$ a $v$.

Precisamos mostrar que:

- $d[v] = dist(s, v)$ para todo $v \in V[G]$.
- A função predecessor $\pi[\,]$ define uma Árvore de Busca em Largura com raiz $s$.

## Alguns lemas

Lema 1. Seja $G$ um grafo e $s \in V[G]$.

Então para toda aresta $(u, v)$ temos que

$$dist(s, v) \leq dist(s, u) + 1.$$

Prova:

Imediato.



## Alguns lemas

$d[v]$ é uma estimativa superior de $dist(s, v)$.

Lema 2. Durante a execução do algoritmo vale o seguinte invariante

$$d[v] \geq dist(s, v) \text{ para todo } v \in V[G].$$



## Prova do Lema 2

Indução no número de operações ENQUEUE.

Base: quando $s$ é inserido na fila temos $d[s] = 0 = dist(s, s)$ e $d[v] = \infty \geq dist(s, v)$ para $v \in V - \{s\}$.

Passo de indução: $v$ é descoberto enquanto a busca é feita em $u$ (percorrendo $\mathrm{Adj}[u]$). Então

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq dist(s, u) + 1 \quad \text{(HI)} \\ &\geq dist(s, v). \quad \text{(Lema 1)} \end{aligned}$$

Note que $d[v]$ nunca muda após $v$ ser inserido na fila. Logo, o invariante vale.



## Alguns lemas

Lema 3. Suponha que $\langle v_1, v_2, \ldots, v_r \rangle$ seja a disposição da fila $Q$ na linha 10 em uma iteração qualquer.

Então

$$d[v_r] \leq d[v_1] + 1$$

e

$$d[v_i] \leq d[v_{i+1}] \text{ para } i = 1, 2, \ldots, r - 1.$$

Em outras palavras, os vértices são inseridos na fila em ordem crescente e há no máximo dois valores de $d[v]$ para vértices na fila.

## Prova do Lema 3

Indução no número de operações ENQUEUE e DEQUEUE.

Base: $Q = \{s\}$. O lema vale trivialmente.

Passo de indução: $v_1$ é removido de $Q$. Agora $v_2$ é o primeiro vértice de $Q$. Então

$$d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1.$$

As outras desigualdades se mantêm.

Passo de indução: $v = v_{r+1}$ é inserido em $Q$. Suponha que a busca é feita em $u$ neste momento. Logo $d[v_1] \geq d[u]$. Então

$$d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1.$$

Pela HI $d[v_r] \leq d[u] + 1$. Logo

$$d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}].$$

As outras desigualdades se mantêm.



## Corretude

Teorema. Seja $G$ um grafo e $s \in V[G]$.

Então após a execução de BUSCA-EM-LARGURA,

$$d[v] = dist(s, v) \text{ para todo } v \in V[G].$$

e

$\pi[\,]$ define uma Árvore de Busca em Largura.

Prova:

Note que se $dist(s, v) = \infty$ então $d[v] = \infty$ pelo Lema 3.

Então vamos considerar o caso em que $dist(s, v) < \infty$.

Vamos provar por indução em $dist(s, v)$ que $d[v] = dist(s, v)$.



## Corretude

Base: Se $dist(s, v) = 0$ então $v = s$ e $d[s] = 0$.

Hipótese de indução: Suponha então que $d[u] = dist(s, u)$ para todo vértice $u$ com $dist(s, u) < k$.

Seja $v$ um vértice com $dist(s, v) = k$. Considere um caminho mínimo de $s$ a $v$ em $G$ e chame de $u$ o vértice que antecede $v$ neste caminho. Note que $dist(s, u) = k - 1$.

Considere o instante em que $u$ foi removido da fila $Q$ (linha 11 de BUSCA-EM-LARGURA). Neste instante, $v$ é branco, cinza ou preto.



## Corretude

- se $v$ é branco, então a linha 15 faz com $d[v] = d[u] + 1 = (k - 1) + 1 = k$.
- se $v$ é cinza, então $v$ foi visitado antes por algum vértice $w$ (logo, $v \in \text{Adj}[w]$) e $d[v] = d[w] + 1$. Pelo Lema 3, $d[w] \leq d[u] = k - 1$ e segue que $d[v] = k$.
- se $v$ é preto, então $v$ já passou pela fila $Q$ e pelo Lema 3, $d[v] \leq d[u] = k - 1$. Mas por outro lado, pelo Lema 2, $d[v] \geq dist(s, v) = k$, o que é uma contradição. Este caso não ocorre.

Portanto, em todos os casos temos que $d[v] = dist[s, v]$.

## Caminho mais curto

Imprime um caminho mais curto de $s$ a $v$.

```
Print-Path(G, s, v)
1  se v = s então
2      imprime s
3  senão
4      se π[v] = NIL então
4          imprime não existe caminho de s a v.
5      senão
6          Print-Path(G, s, π[v])
7          imprime v.
```



## Exemplo

Exercício. Mostre que um grafo $G$ é bipartido se e somente se não contém um ciclo de comprimento ímpar.

Projete um algoritmo linear que dado um grafo $G$ devolve

- uma bipartição de $G$, ou
- um ciclo ímpar em $G$.



## Busca em profundidade

Depth First Search = busca em profundidade

- A estratégia consiste em pesquisar o grafo o mais "profundamente" sempre que possível.
- Aplicável tanto a grafos orientados quanto não-orientados.
- Possui um número enorme de aplicações:
  - determinar os componentes de um grafo
  - ordenação topológica
  - determinar componentes fortemente conexos
  - subrotina para outros algoritmos



## Busca em profundidade

Recebe um grafo $G = (V, E)$ (representado por listas de adjacências). A busca inicia-se em um vértice qualquer. Busca em profundidade é um método recursivo. A idéia básica consiste no seguinte:

- Suponha que a busca atingiu um vértice $u$.
- Escolhe-se um vizinho não visitado $v$ de $u$ para prosseguir a busca.
- "Recursivamente" a busca em profundidade prossegue a partir de $v$.
- Quando esta busca termina, tenta-se prosseguir a busca a partir de outro vizinho de $u$. Se não for possível, ela retorna (*backtracking*) ao nível anterior da recursão.

## Busca em profundidade

Outra forma de entender Busca em Profundidade é imaginar que os vértices são armazenados em uma pilha à medida que são visitados. Compare isto com Busca em Largura onde os vértices são colocados em uma fila.

- Suponha que a busca atingiu um vértice $u$.
- Escolhe-se um vizinho não visitado $v$ de $u$ para prosseguir a busca.
- Empilhe $v$ e repete-se o passo anterior com $v$.
- Se nenhum vértice não visitado foi encontrado, então desempilhe um vértice da pilha, digamos $u$, e volte ao primeiro passo.



## Floresta de Busca em Profundidade

- A busca em profundidade associa a cada vértice $x$ um predecessor $\pi[x]$.
- O subgrafo induzido pelas arestas

  $\{(\pi[x], x) : x \in V[G] \text{ e } \pi[x] \neq \text{NIL}\}$

  é a Floresta de Busca em Profundidade.
- Cada componente desta floresta é uma Árvore de Busca em Profundidade.



## Cores dos vértices

A medida que o grafo é percorrido, os vértices visitados vão sendo coloridos.

Cada vértice tem uma das seguintes cores:

- Cor branca = "vértice ainda não visitado"'.
- Cor cinza = "vértice visitado mas ainda não finalizado".
- Cor preta = "vértice visitado e finalizado".



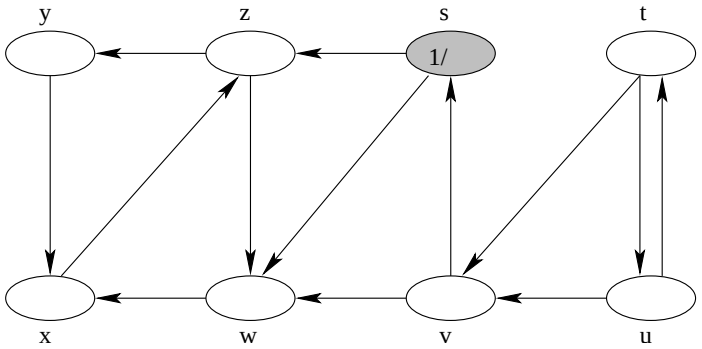
## Estampas/rótulos

A busca em profundidade associa a cada vértice $x$ dois rótulos:

- $d[x]$: instante de descoberta de $x$.
  Neste instante $x$ torna-se cinza.
- $f[x]$: instante de finalização de $x$.
  Neste instante $x$ torna-se preto.

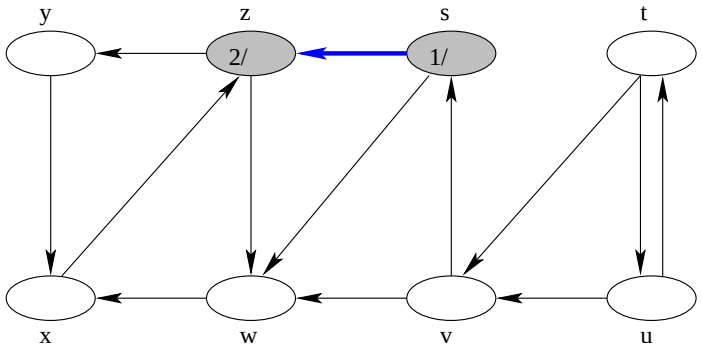
Os rótulos são inteiros entre 1 e $2|V|$.

## Exemplo



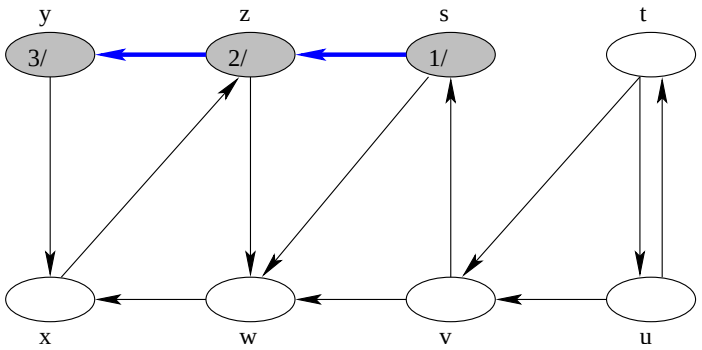



## Exemplo



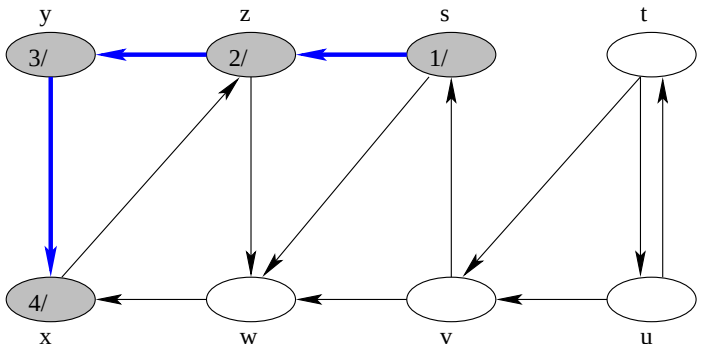



## Exemplo

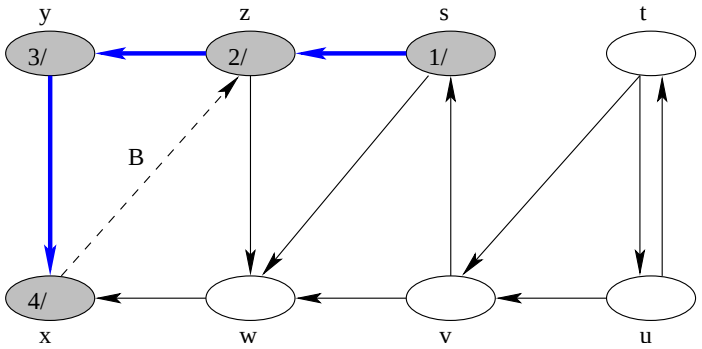





## Exemplo

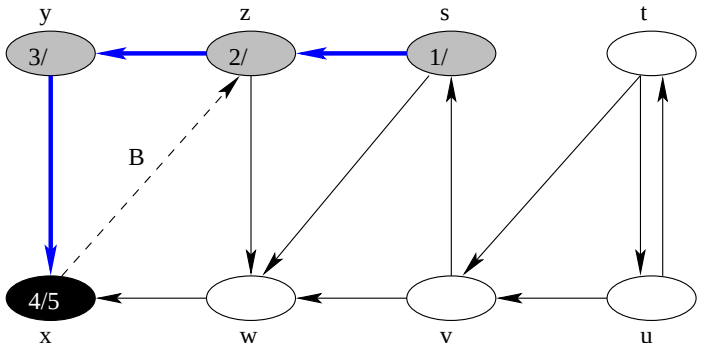


## Exemplo



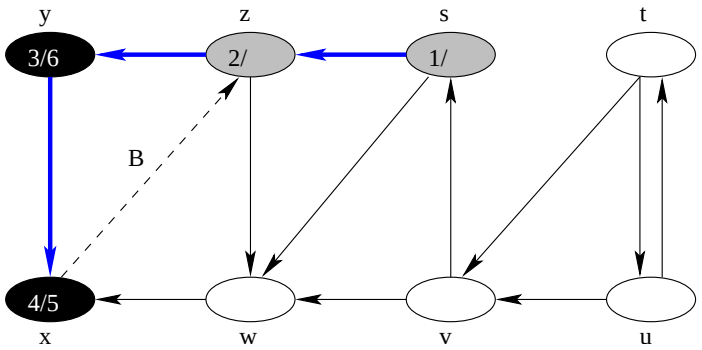



## Exemplo



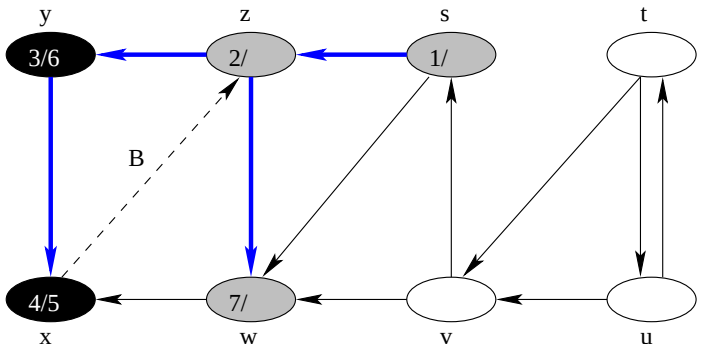



## Exemplo

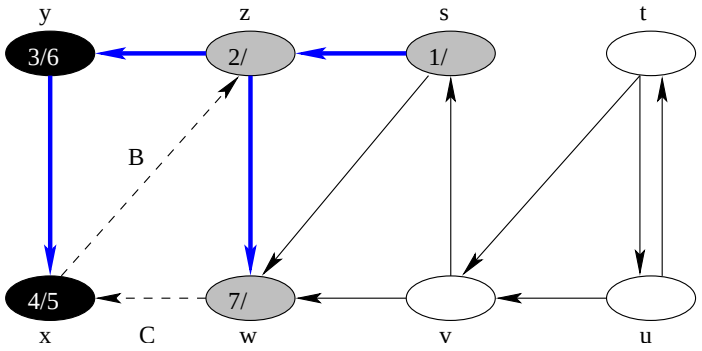





## Exemplo

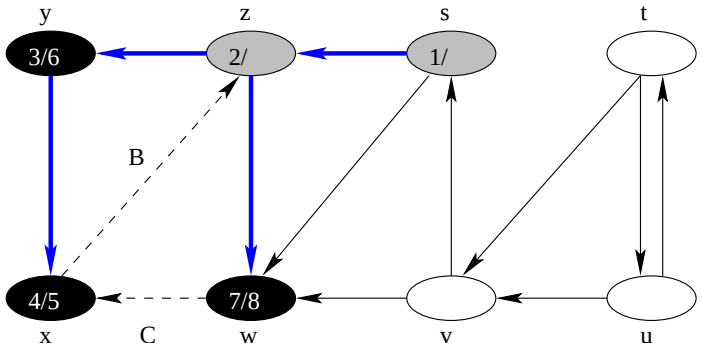


## Exemplo



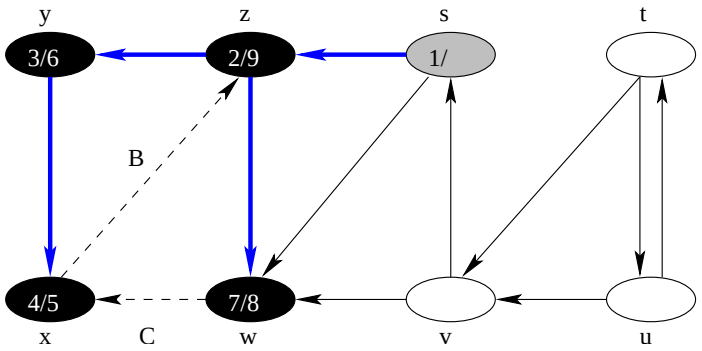



## Exemplo



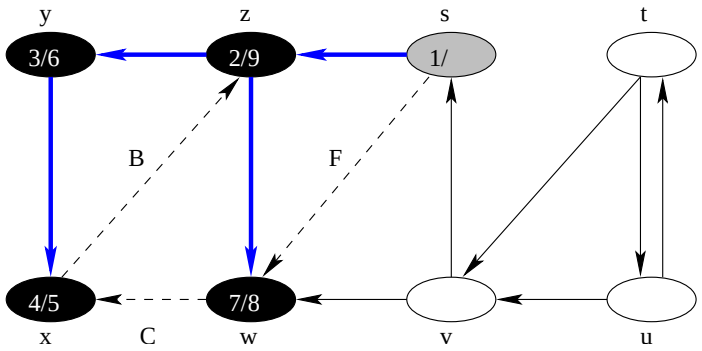



## Exemplo

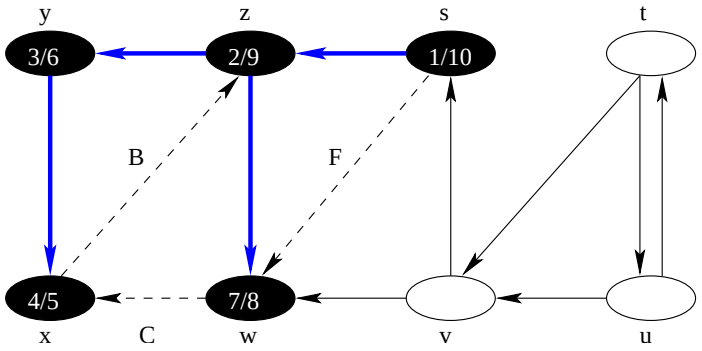





## Exemplo

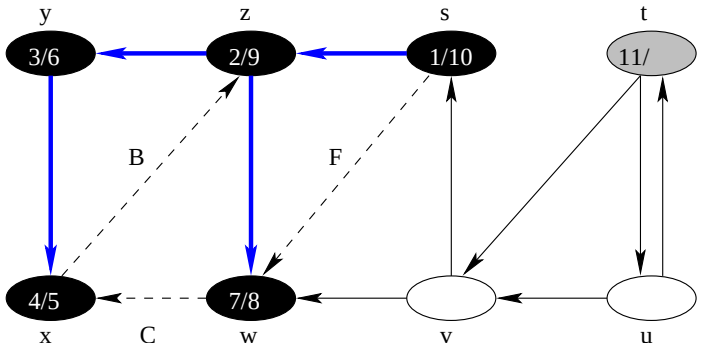


## Exemplo



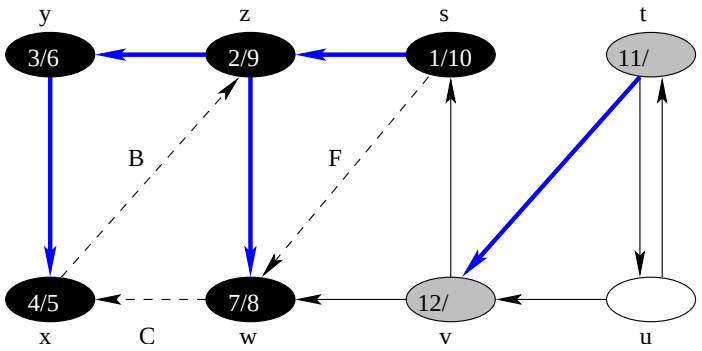



## Exemplo



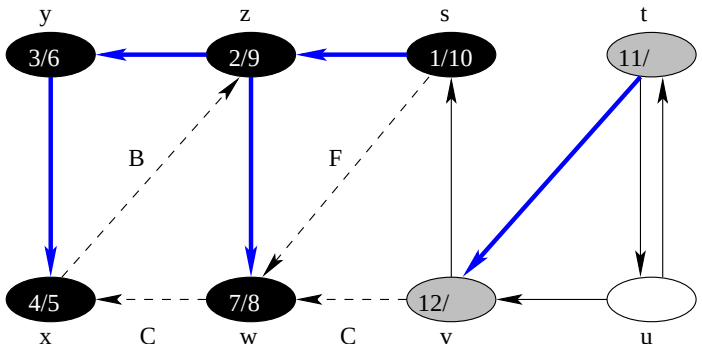



## Exemplo

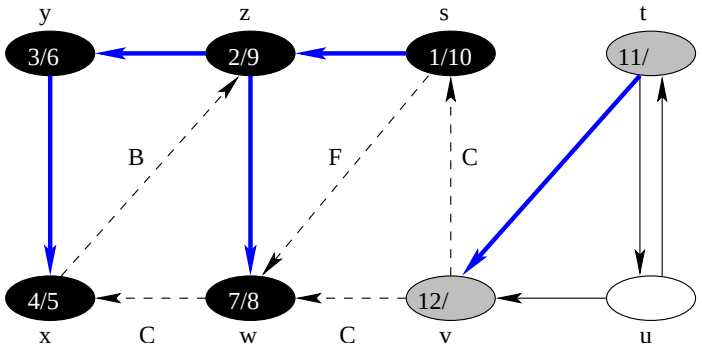





## Exemplo

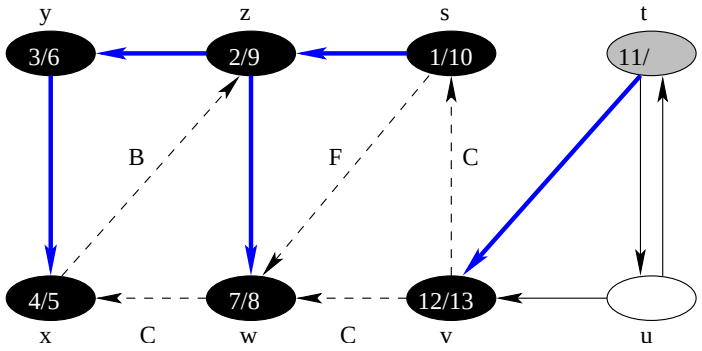


## Exemplo



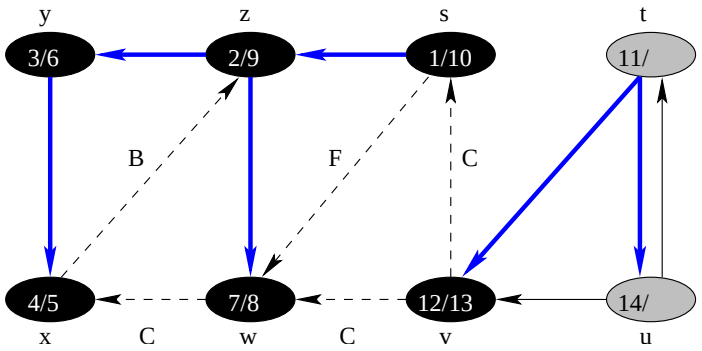



## Exemplo



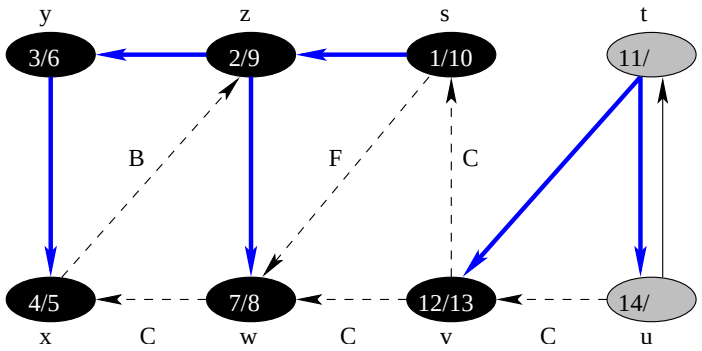



## Exemplo

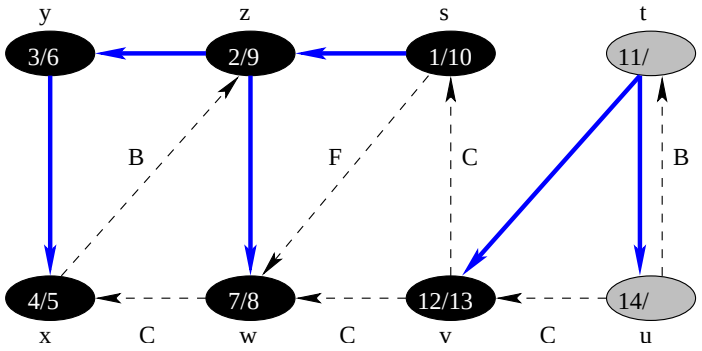





## Exemplo

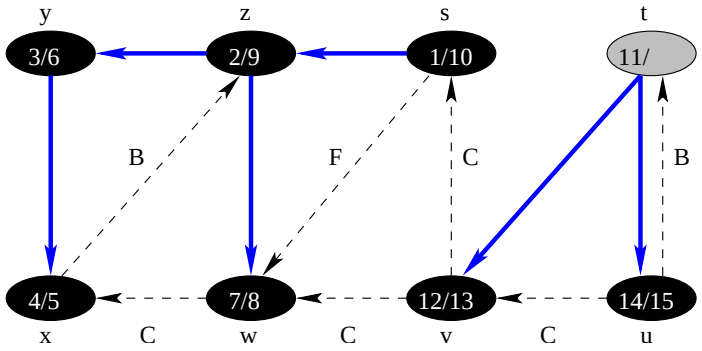


## Exemplo



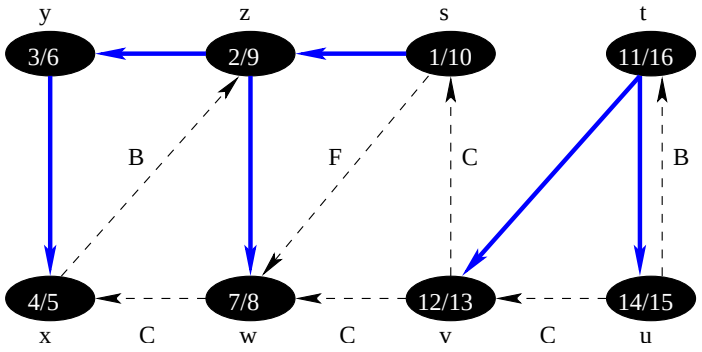



## Exemplo






## Exemplo






## Rótulos versus cores

Para todo $x \in V[G]$ vale que $d[x] < f[x]$.

Além disso

- $x$ é branco antes do instante $d[x]$.
- $x$ é cinza entre os instantes $d[x]$ e $f[x]$.
- $x$ é preto após o instante $f[x]$.

## Algoritmo DFS

Recebe um grafo $G$ (na forma de listas de adjacências) e devolve
(i) os instantes $d[v], f[v]$ para cada $v \in V$ e
(ii) uma Floresta de Busca em Profundidade.

DFS($G$)
1 **para cada** $u \in V[G]$ **faça**
2 $\quad$ cor$[u] \leftarrow$ branco
3 $\quad$ $\pi[u] \leftarrow$ NIL
4 tempo $\leftarrow 0$
5 **para cada** $u \in V[G]$ **faça**
6 $\quad$ **se** cor$[u] =$ branco
7 $\quad\quad$ **então** DFS-VISIT($u$)



## Algoritmo DFS-VISIT

Constrói recursivamente uma Árvore de Busca em Profundidade com raiz $u$.

DFS-VISIT($u$)
1 cor$[u] \leftarrow$ cinza
2 tempo $\leftarrow$ tempo $+ 1$
3 $d[u] \leftarrow$ tempo
4 **para cada** $v \in$ Adj$[u]$ **faça**
5 $\quad$ **se** cor$[v] =$ branco
6 $\quad\quad$ **então** $\pi[v] \leftarrow u$
7 $\quad\quad\quad$ DFS-VISIT($v$)
8 cor$[u] \leftarrow$ preto
9 $f[u] \leftarrow$ tempo $\leftarrow$ tempo $+ 1$



## Algoritmo DFS

DFS($G$)
1 **para cada** $u \in V[G]$ **faça**
2 $\quad$ cor$[u] \leftarrow$ branco
3 $\quad$ $\pi[u] \leftarrow$ NIL
4 tempo $\leftarrow 0$
5 **para cada** $u \in V[G]$ **faça**
6 $\quad$ **se** cor$[u] =$ branco
7 $\quad\quad$ **então** DFS-VISIT($u$)

Consumo de tempo

$O(V) + V$ chamadas a DFS-VISIT().



## Algoritmo DFS-VISIT

DFS-VISIT($u$)
1 cor$[u] \leftarrow$ cinza
2 tempo $\leftarrow$ tempo $+ 1$
3 $d[u] \leftarrow$ tempo
4 **para cada** $v \in$ Adj$[u]$ **faça**
5 $\quad$ **se** cor$[v] =$ branco
6 $\quad\quad$ **então** $\pi[v] \leftarrow u$
7 $\quad\quad\quad$ DFS-VISIT($v$)
8 cor$[u] \leftarrow$ preto
9 $f[u] \leftarrow$ tempo $\leftarrow$ tempo $+ 1$

Consumo de tempo
linhas 4-7: executado $|\text{Adj}[u]|$ vezes.

## Complexidade de DFS

- DFS-VISIT($v$) é executado exatamente uma vez para cada $v \in V$.
- Em uma execução de DFS-VISIT($v$), o laço das linhas 4-7 é executado $|\mathrm{Adj}[u]|$ vezes.
  Assim, o custo total de todas as chamadas é
  $\sum_{v \in V} |\mathrm{Adj}(v)| = \Theta(E)$.

Conclusão: A complexidade de tempo de DFS é $O(V + E)$.

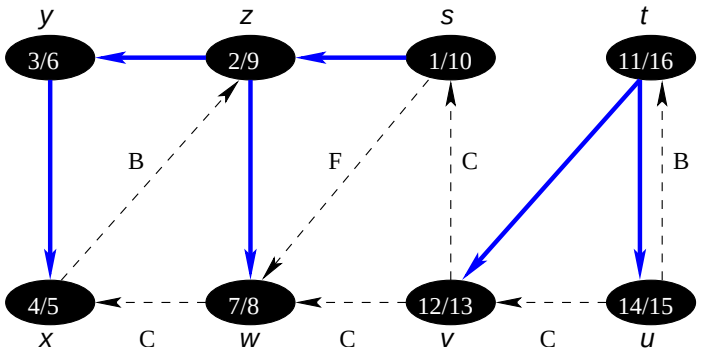


## Estrutura de parênteses

- Os rótulos $d[x], f[x]$ têm propriedades muito úteis para serem usadas em outros algoritmos.
- Eles refletem a ordem em que a busca em profundidade foi executada.
- Eles fornecem informação de como é a "cara" (estrutura) do grafo.

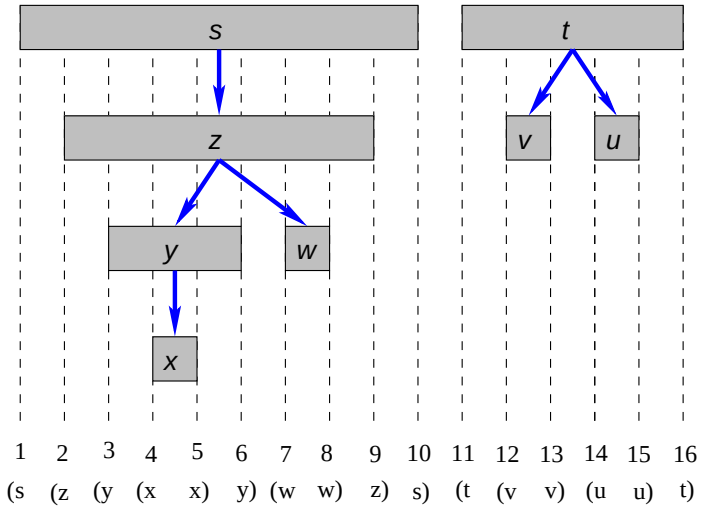


## Estrutura de parênteses






## Estrutura de parênteses



## Estrutura de parênteses

Teorema (Teorema dos Parênteses)

Em uma busca em profundidade sobre um grafo $G = (V, E)$, para quaisquer vértices $u$ e $v$, ocorre exatamente uma das situções abaixo:

- $[d[u], f[u]]$ e $[d[v], f[v]]$ são disjuntos.
- $[d[u], f[u]]$ está contido em $[d[v], f[v]]$ e $u$ é descendente de $v$ na Árvore de BP.
- $[d[v], f[v]]$ está contido em $[d[u], f[u]]$ e $v$ é descendente de $u$ na Árvore de BP.



## Estrutura de parênteses

Corolário. (Intervalos encaixantes para descendentes)

Um vértice $v$ é um descendente próprio de $u$ na Floresta de BP se e somente se $d[u] < d[v] < f[v] < f[u]$.

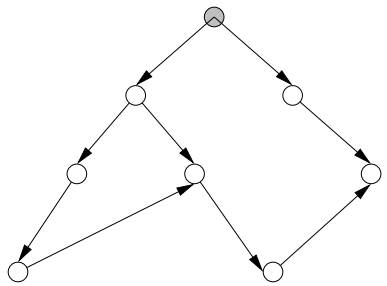
Equivalentemente, $v$ é um descendente próprio de $u$ se e somente se $[d[v], f[v]]$ está contido em $[d[u], f[u]]$.



## Teorema do Caminho Branco

Teorema. (Teorema do Caminho Branco)

Em uma Floresta de BP, um vértice $v$ é descendente de $u$ se e somente se no instante $d[u]$ (quando $u$ foi descoberto), existia um caminho de $u$ a $v$ formado apenas por vértices brancos.





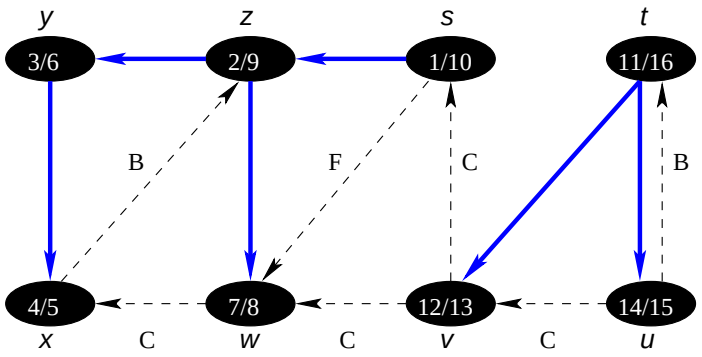
## Classificação de arestas

Busca em profundidade pode ser usada para classificar arestas de um grafo $G = (V, E)$.

Ela classifica as arestas em quatro tipos:

- Arestas da árvore: arestas que pertencem à Floresta de BP.
- Arestas de retorno: arestas $(u, v)$ ligando um vértice $u$ a um ancestral $v$ na Árvore de BP.
- Arestas de avanço: arestas $(u, v)$ ligando um vértice $u$ a um descendente próprio $v$ na Árvore de BP.
- Arestas de cruzamento: todas as outras arestas.

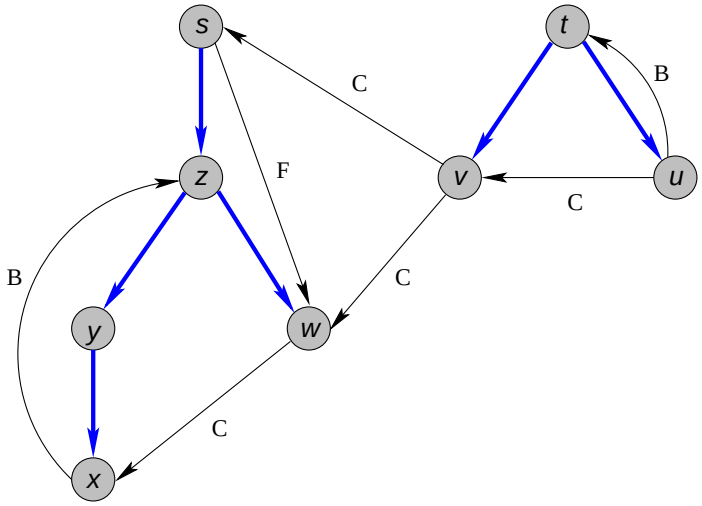
## Classificação de arestas




É fácil modificar o algoritmo DFS($G$) para que ele também classifique as arestas de $G$. (Exercício)



## Classificação de arestas






## Grafos não-orientados

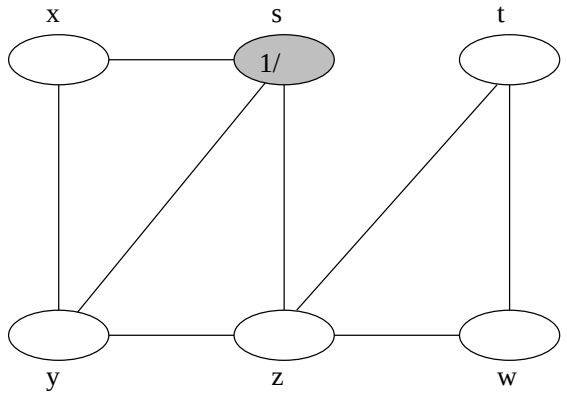
Em grafos não-orientados $(u, v)$ e $(v, u)$ indicam a mesma aresta. A sua classificação depende de quem foi visitado primeiro: $u$ ou $v$.

Para grafos não-orientados, existem apenas dois tipos de arestas.

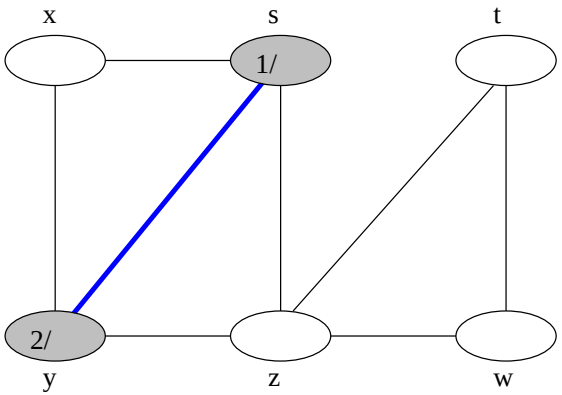
Teorema.
Em uma busca em profundidade sobre um grafo não-orientado $G$, cada aresta de $G$ ou é aresta da árvore ou é aresta de retorno.



## Exemplo

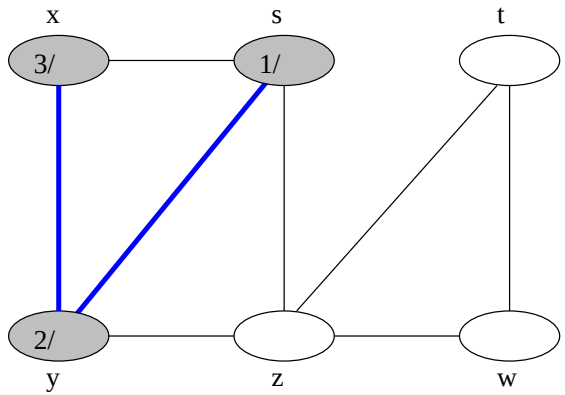


## Exemplo



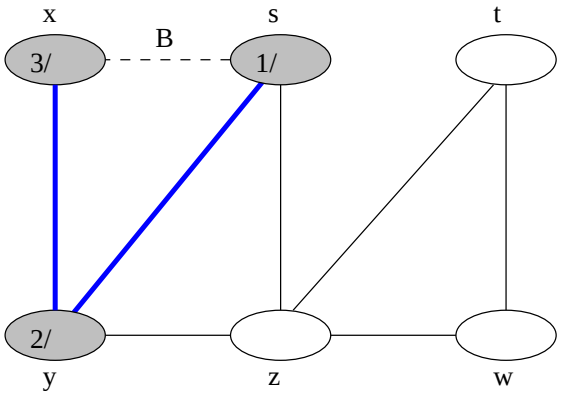



## Exemplo



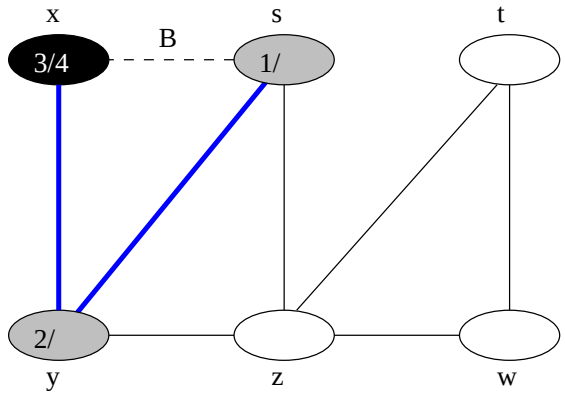



## Exemplo

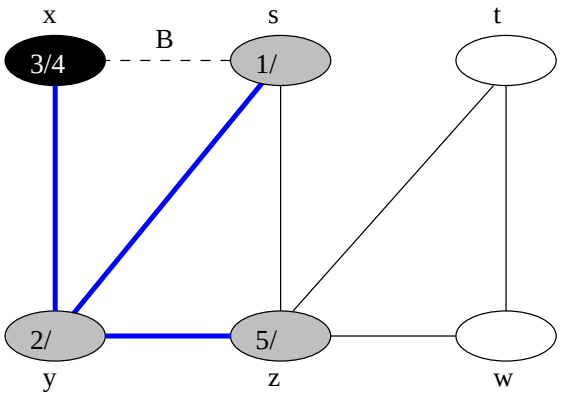





## Exemplo

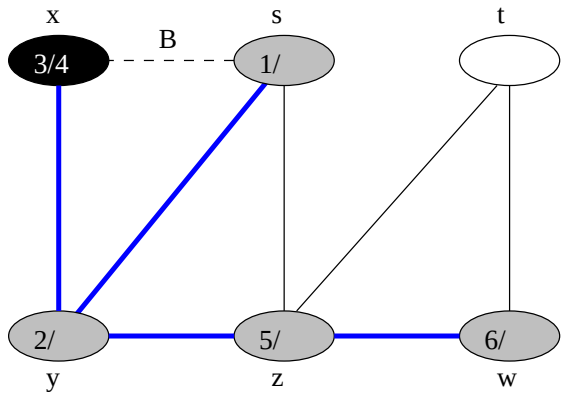


## Exemplo



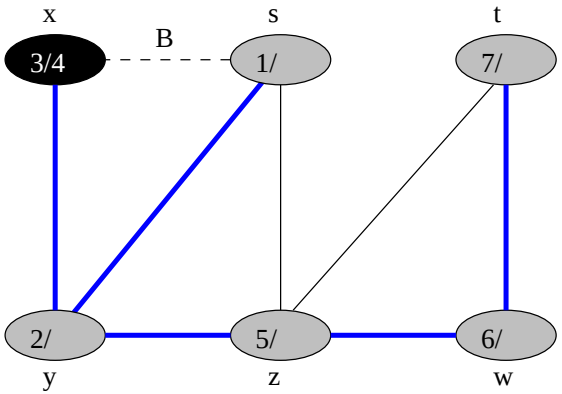



## Exemplo



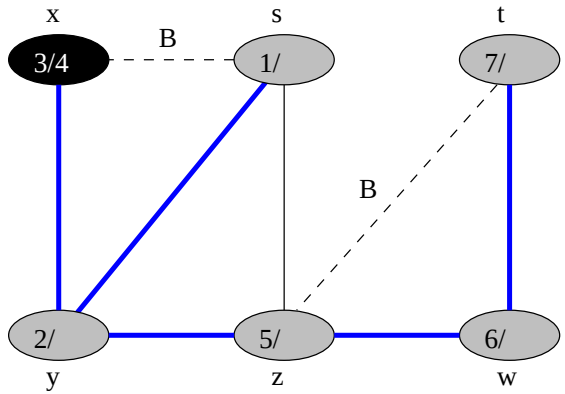



## Exemplo

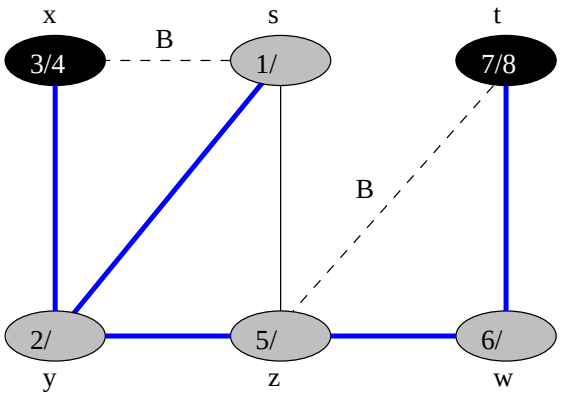





## Exemplo

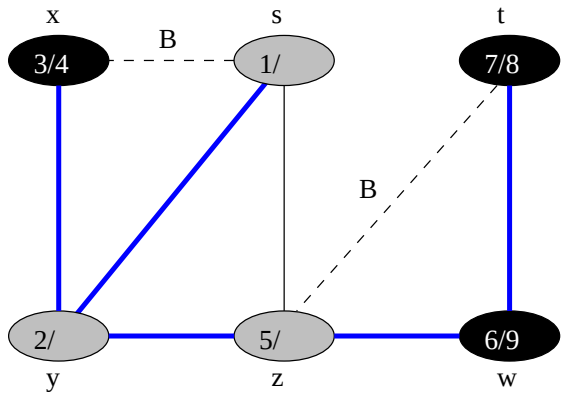


## Exemplo



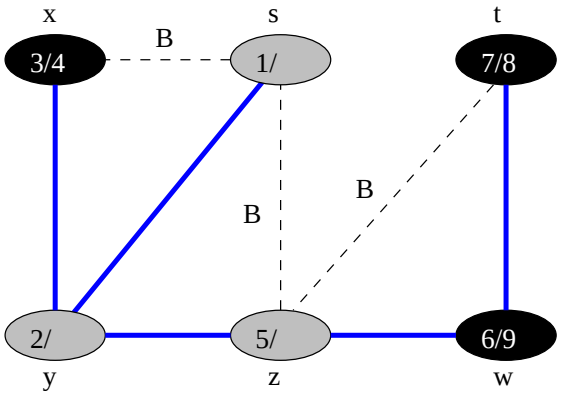



## Exemplo



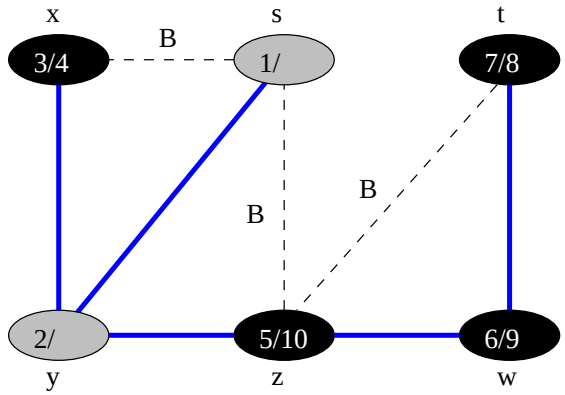



## Exemplo

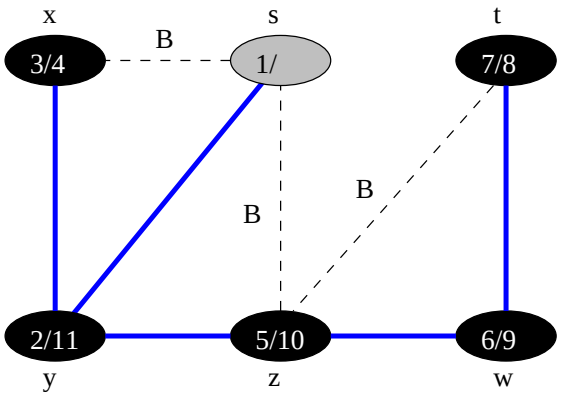





## Exemplo

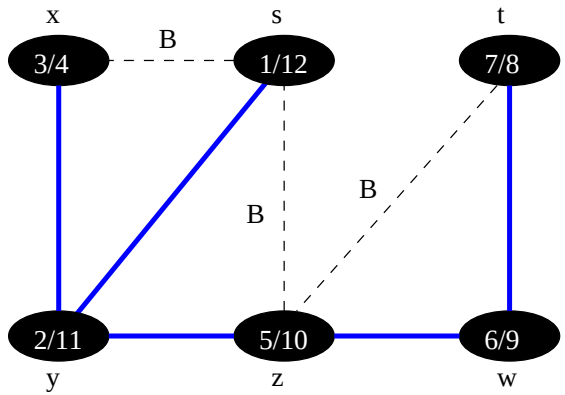


## Exemplo






## Exemplo



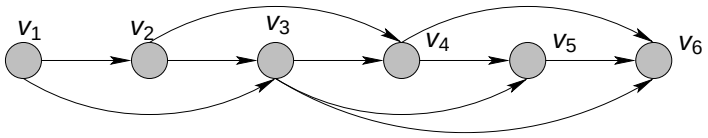



## Ordenação Topológica

Uma ordenação topológica de um grafo orientado $G = (V, E)$ é um arranjo linear dos vértices de $G$

$v_1 \quad v_2 \quad v_3 \quad \ldots \quad v_{n-2} \quad v_{n-1} \quad v_n$

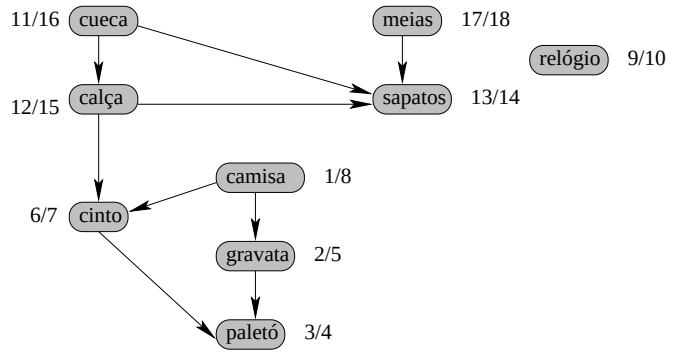
tal que se $(v_i, v_j)$ é uma aresta de $G$, então $i < j$.



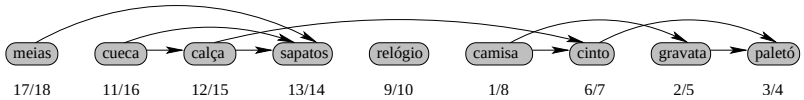



## Ordenação Topológica

Ordenação topológica é usada em aplicações onde eventos ou tarefas têm precedência sobre outras.



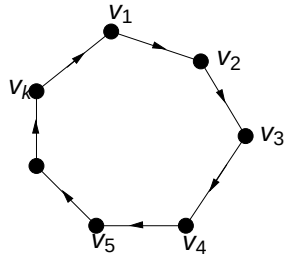
## Ordenação Topológica






## Ordenação Topológica

- Nem todo grafo orientado possui uma ordenação topológica.
  Por exemplo, um ciclo orientado não possui uma ordenação topológica.




- Um grafo orientado $G = (V, E)$ é acíclico se **não** contém um ciclo orientado.



## Grafo Orientado Acíclico

Teorema. Um grafo orientado $G$ é acíclico se e somente se possui uma ordenação topológica.

Prova.

Obviamente, se $G$ possui uma ordenação topológica então $G$ é acíclico.

Vamos mostrar a recíproca.

**Definição**

Uma fonte é um vértice com grau de entrada igual a zero.
Um sorvedouro é um vértice com grau de saída igual a zero.



## Grafo Orientado Acíclico

Lema. Todo grafo orientado acíclico possui uma fonte e um sorvedouro.

Baseado no resultado acima pode-se projetar um algoritmo para obter uma ordenação topológica de um grafo orientado acíclico $G$.

- Encontre uma fonte $v_1$ de $G$.
- Recursivamente encontre uma ordenação topológica $v_2, \ldots, v_n$ de $G - v_1$.
- Devolva $v_1, v_2, \ldots, v_n$.

Complexidade: $O(V^2)$ (análise grosseira)
Pode-se fazer melhor: O(V+E) (CLRS 22.4-5)

## Ordenação Topológica

Recebe um grafo orientado acíclico $G$ e
devolve uma ordenação topológica de $G$.

TOPOLOGICAL-SORT($G$)
1 Execute DFS($G$) para calcular $f[v]$ para cada vértice $v$
2 À medida que cada vértice for finalizado, coloque-o no início de uma lista ligada
3 Devolva a lista ligada resultante

Outro modo de ver a linha 2 é:
Imprima os vértices em ordem decrescente de $f[v]$.



## Complexidade de tempo

Conclusão

A complexidade de tempo de TOPOLOGICAL-SORT é $O(V+E)$.

Agora falta mostrar que TOPOLOGICAL-SORT funciona.



## Corretude

Lema.
Um grafo orientado $G$ é acíclico se e somente se em uma busca em profundidade de $G$ não aparecem arestas de retorno.

Prova:

Suponha que $(u,v)$ é uma aresta de retorno.

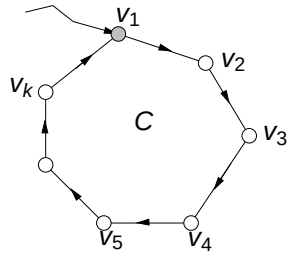
Então $v$ é um ancestral de $u$ na Floresta de BP.

Portanto, existe um caminho de $v$ a $u$ que juntamente com $(u,v)$ forma um ciclo orientado. Logo, $G$ não é acíclico.



## Corretude

Agora suponha que $G$ contém um ciclo orientado $C$.




Suponha que $v_1$ é o primeiro vértice de $C$ a ser descoberto.
Então no instante $d[v_1]$ existe um caminho branco de $v_1$ a $v_k$.
Pelo Teorema do Caminho Branco, $v_k$ torna-se um descendente de $v_1$ e portanto, $(v_k, v_1)$ torna-se uma aresta de retorno.

## Corretude

Lembre que TOPOLOGICAL-SORT imprime os vértices em ordem decrescente de $f[\,]$.

Para mostrar que o algoritmo funciona, basta então mostrar que se $(u, v)$ é uma aresta de $G$, então $f[u] > f[v]$.

Considere o instante em que $(u, v)$ é examinada.

Neste instante, $v$ não pode ser cinza pois senão $(u, v)$ seria uma aresta de retorno.

Logo, $v$ é branco ou preto.



## Corretude

- Se $v$ é branco, então $v$ é descendente de $u$ e portanto $f[v] < f[u]$.
- Se $v$ é preto, então $v$ já foi finalizado e $f[v]$ foi definido. Por outro lado $u$ ainda não foi finalizado. Logo, $f[v] < f[u]$.

Portanto, TOPOLOGICAL-SORT funciona corretamente.



Componentes fortemente conexos



## Componentes fortemente conexos (CFC)

- Uma aplicação clássica de busca em profundidade:

  decompor um grafo orientado em seus componentes fortemente conexos.
- Muitos algoritmos em grafos começam com tal decomposição.
- O algoritmo opera separadamente em cada componente fortemente conexo.
- As soluções são combinadas de alguma forma.

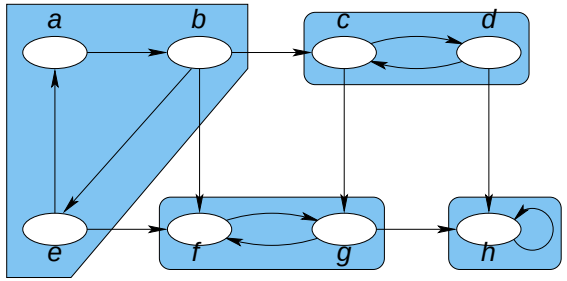
## Componentes fortemente conexos

Um componente fortemente conexo de um grafo orientado $G = (V, E)$ é um subconjunto de vértices $C \subseteq V$ tal que:

1. Para todo par de vértices $u$ e $v$ em $C$, existe um caminho (orientado) de $u$ a $v$ e vice-versa.
2. $C$ é maximal com respeito à propriedade (1).



## Componentes fortemente conexos




Um grafo orientado e seus componentes fortemente conexos.



## Grafo transposto

Seja $G = (V, E)$ um grafo orientado.

O grafo transposto de $G$ é o grafo $G^T = (V^T, E^T)$ tal que

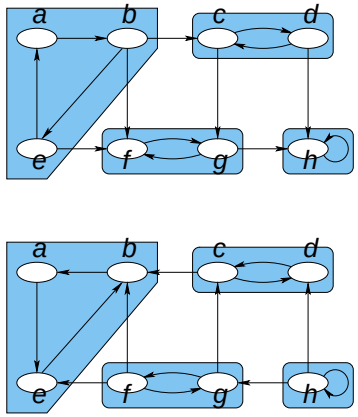
- $V^T = V$ e
- $E^T = \{(u, v) : (v, u) \in E\}$.

Ou seja, $G^T$ é obtido a partir de $G$ invertendo as orientações das arestas.

Dada uma representação de listas de adjacências de $G$ é possível obter a representação de listas de adjacências de $G^T$ em tempo $\Theta(V + E)$.



## Grafo transposto




Um grafo orientado e o grafo transposto. Note que eles têm os mesmos componentes fortemente conexos.

## Algoritmo

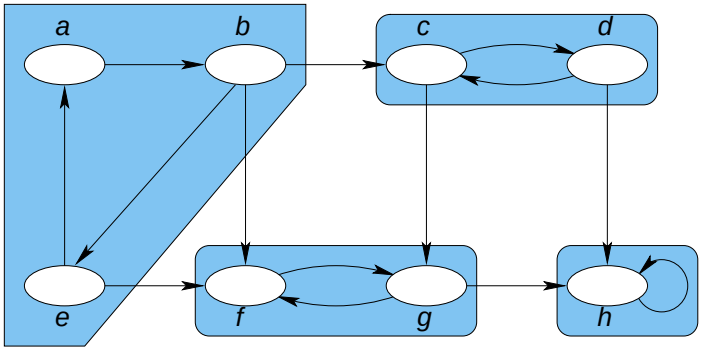
COMPONENTES-FORTEMENTE-CONEXOS($G$)

1 Execute DFS($G$) para obter $f[v]$ para $v \in V$.
2 Execute DFS($G^T$) considerando os vértices em ordem decrescente de $f[v]$.
3 Devolva os conjuntos de vértices de cada árvore da Floresta de Busca em Profundidade obtida.

Veremos que os conjuntos devolvidos são exatamente os componentes fortemente conexos de $G$.

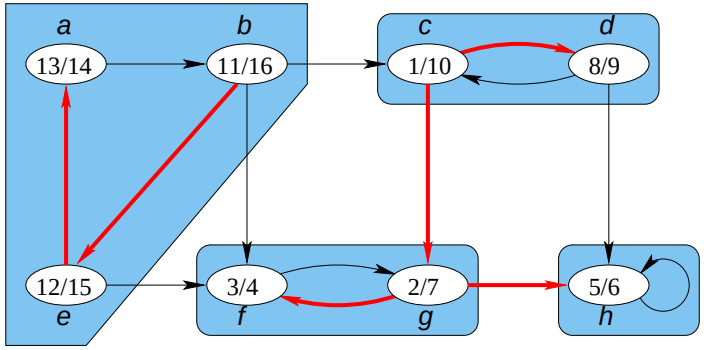


## Exemplo CLRS





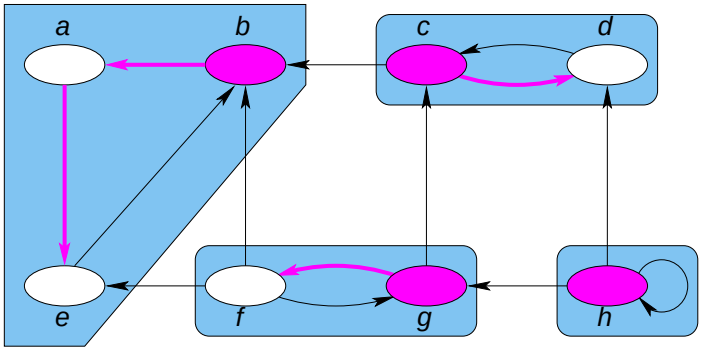

## Exemplo CLRS




1 Execute DFS($G$) para obter $f[v]$ para $v \in V$.



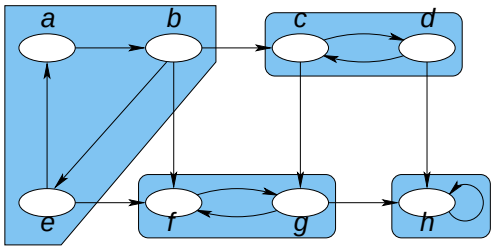
## Exemplo CLRS




2 Execute DFS($G^T$) considerando os vértices em ordem decrescente de $f[v]$.
3 Os componentes fortemente conexos correspondem aos vértices de cada árvore da Floresta de Busca em Profundida.

## Grafo Componente

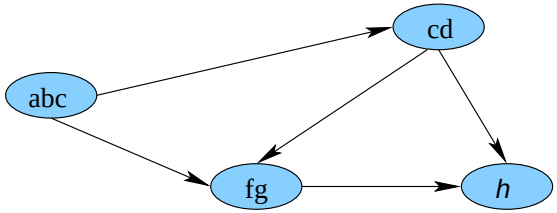
A idéia por trás de COMPONENTES-FORTEMENTE-CONEXOS segue de uma propriedade do grafo componente $G^{CFC}$ obtido a partir de $G$ **contraindo-se** seus componentes fortemente conexos.






## Grafo Componente

A idéia por trás de COMPONENTES-FORTEMENTE-CONEXOS segue de uma propriedade do grafo componente $G^{CFC}$ obtido a partir de $G$ **contraindo-se** seus componentes fortemente conexos.




$G^{CFC}$ é acíclico.

Os componentes fortementes conexos são visitados em ordem topológica com relação a $G^{CFC}$!



## Corretude

**Lema 22.13 (CLRS)**

Sejam $C$ e $C'$ dois componentes f.c. de $G$.
Sejam $u, v \in C$ e $u', v' \in C'$.
Suponha que existe um caminho $u \rightsquigarrow u'$ em $G$.
Então **não existe** um caminho $v' \rightsquigarrow v$ em $G$.

O lema acima mostra que $G^{CFC}$ é acíclico.

Agora vamos mostrar porque COMPONENTES-FORTEMENTE-CONEXOS funciona.



## Corretude

Daqui pra frente $d, f$ referem-se à busca em profundidade em $G$ feita no passo 1 do algoritmo.

**Definição**:

Para todo subconjunto $U$ de vértices denote

$$d(U) := \min_{u \in U}\{d[u]\} \quad \text{e} \quad f(U) := \max_{u \in U}\{f[u]\}.$$

**Lema 22.14 (CLRS):**

Sejam $C$ e $C'$ dois componentes f.c. de $G$.
Suponha que existe $(u, v)$ em $E$ onde $u \in C$ e $v \in C'$.
Então $f(C) > f(C')$.

# Corretude

**Corolário 22.15 (CLRS):**

Sejam $C$ e $C'$ dois componentes f.c. de $G$.
Suponha que existe $(u, v)$ está em $E^T$ onde $u \in C$ e $v \in C'$.
Então $f(C) < f(C')$.

**Teorema 22.16 (CLRS):**

O algoritmo COMPONENTES-FORTEMENTE-CONEXOS determina corretamente os componentes fortemente conexos de $G$.