MC448 — Análise de Algoritmos I

Cid Carvalho de Souza Cândida Nunes da Silva
Orlando Lee

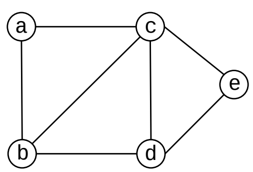
10 de novembro de 2009

Grafos: Noções Básicas e Representação

Definição de Grafo

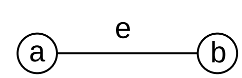
Um *grafo* é um par $G = (V, E)$ onde

- V é um conjunto finito de elementos chamados *vértices* e
- E é um conjunto finito de pares não-ordenados de vértices chamados *arestas*.
- **Exemplo:**
  $V = \{a, b, c, d, e\}$
  $E = \{(a,b), (a,c), (b,c), (b,d), (c,d), (c,e), (d,e)\}$



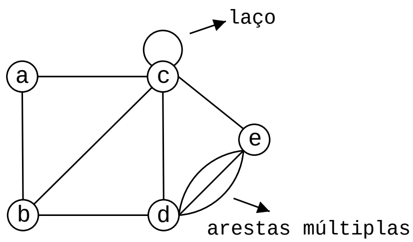
Definição de Grafo

- Dada uma aresta $e = (a, b)$, dizemos que os vértices a e b são os *extremos* da aresta e e que a e b são vértices *adjacentes*.
- Dizemos também que a aresta e é *incidente* aos vértices a e b, e que os vértices a e b são incidentes à aresta e.



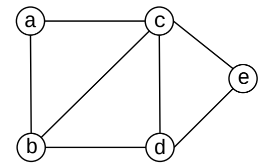
Grafo Simples

- Dizemos que um grafo é *simples* quando não possui laços ou arestas múltiplas.
- Um *laço* é uma aresta com extremos idêntico e *arestas múltiplas* são duas ou mais arestas com o mesmo par de vértices como extremos.
- **Exemplo:**



Tamanho do Grafo

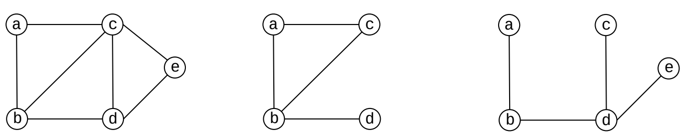
- Denotamos por $|V|$ e $|E|$ a cardinalidade dos conjuntos de vértices e arestas de um grafo G, respectivamente.
- No exemplo abaixo temos $|V| = 5$ e $|E| = 7$.



O *tamanho* do grafo G é dado por $|V| + |E|$.

Subgrafo e Subgrafo Gerador

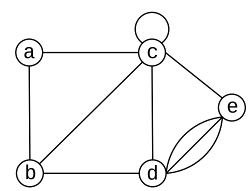
- Um *subgrafo* $H = (V', E')$ de um grafo $G = (V, E)$ é um grafo tal que $V' \subseteq V$, $E' \subseteq E$.
- Um *subgrafo gerador* de G é um subgrafo H com $V' = V$.
- **Exemplo:**



Grafo G — Subgrafo não gerador — Subgrafo gerador

Grau de um vértice

- O *grau* (*degree*) de um vértice v, denotado por $d(v)$ é o número de arestas incidentes a v, com laços contados duas vezes.
- **Exemplo:**



d(a)=2
d(b)=3
d(c)=6
d(d)=5
d(e)=4

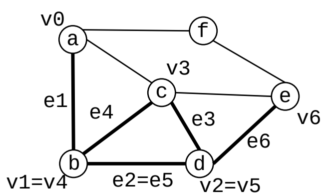
Teorema (*Handshaking lemma*)

Para todo grafo $G = (V, E)$ temos:

$$\sum_{v \in V} d(v) = 2|E|.$$

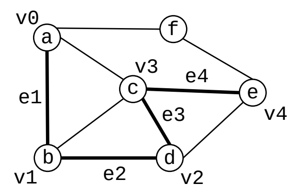
Caminhos em Grafos

- Um *caminho* P de v_0 a v_n no grafo G é uma seqüência finita e não vazia $(v_0, e_1, v_1, \ldots, e_n, v_n)$ cujos elementos são alternadamente vértices e arestas e tal que, para todo $1 \leq i \leq n$, v_{i-1} e v_i são os extremos de e_i.
- O *comprimento* do caminho P é dado pelo seu número de arestas, ou seja, n.
- **Exemplo:**

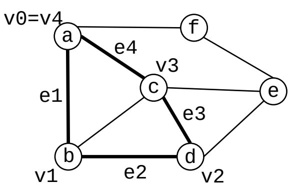


Caminhos Simples e Ciclos

- Um *caminho simples* é um caminho em que não há repetição de vértices e nem de arestas na sequência.
- Um *ciclo* ou *caminho fechado* é um caminho em que $v_0 = v_n$.
- **Exemplo:**



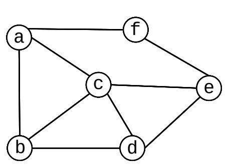
Caminho Simples



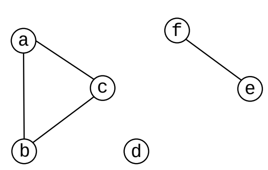
Ciclo

Grafo Conexo

- Dizemos que um grafo é *conexo* se, para qualquer par de vértices u e v de G, existe um caminho de u a v em G.
- Quando o grafo G não é conexo, podemos particionar em *componentes conexos*. Dois vértices u e v de G estão no mesmo componente conexo de G se há caminho de u a v em G.
- **Exemplo:**



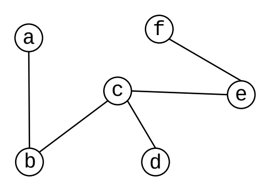
Conexo



Não-conexo com
3 componentes conexos

Árvore

- Um grafo G é uma *árvore* se é conexo e não possui ciclos (acíclico). As seguintes afirmações são equivalentes:
  - G é uma árvore.
  - G é conexo e possui exatamente $|V| - 1$ arestas.
  - G é conexo e a remoção de qualquer aresta desconecta o grafo (minimal conexo).
  - Para todo par de vértices u, v de G, existe um único caminho de u a v em G.
- **Exemplo:**



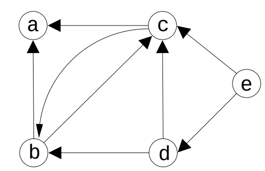
- Uma folha é um vértice de grau 1.

Alguns exemplos de grafos

- Floresta: grafo acíclico (não precisa ser conexo). Cada componente é uma árvore!
- Grafo completo: para todo par de vértices u, v a aresta (u, v) pertence ao grafo.
- Grafo bipartido: possui uma bipartição (A, B) do conjunto de vértices tal que toda aresta tem um extremo em A e outro em B.
- Grafo planar: pode ser desenhado no plano de modo que arestas se interceptam apenas nos extremos.

Grafo Orientado

- As definições que vimos até agora são para grafos *não orientados*.
- Um *grafo orientado* é definido de forma semelhante, com a diferença que as arestas (às vezes chamadas de *arcos*) consistem de pares ordenados de vértices.
- **Exemplo:**



- Às vezes, para enfatizar, dizemos grafo não-orientado em vez de simplesmente grafo.

Grafo orientado

- Se $e = (u, v)$ é uma aresta de um grafo orientado G, então dizemos que e sai de u e entra em v.
- O *grau de saída* $d^+(v)$ de um vértice v é o número de arestas que saem de v. O *grau de entrada* $d^-(v)$ de v é o número de arestas que entram em v.

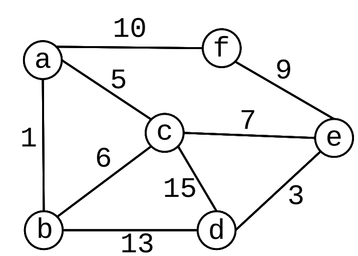
Teorema.
Para todo grafo orientado $G = (V, E)$ temos:

$$\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v) = |E|.$$

- Em geral considera-se que caminhos e ciclos de grafos orientados são orientados (todas as arestas "vão na mesma direção").
- Há um conceito de conexidade para grafos orientados que veremos mais tarde.

Grafo Ponderado

- Um grafo (orientado ou não) é *ponderado* se a cada aresta e do grafo está associado um valor real $c(e)$, o qual denominamos *custo (ou peso)* da aresta.
- **Exemplo:**



Algoritmos em Grafos - Motivação

- Grafos são estruturas abstratas que podem modelar diversos problemas do mundo real.
- Por exemplo, um grafo pode representar conexões entre cidades por estradas ou uma rede de computadores.
- O interesse em estudar algoritmos para problemas em grafos é que conhecer um algoritmo para um determinado problema em grafos pode significar conhecer algoritmos para diversos problemas reais.

Aplicações

- Caminho mínimo: dado um conjunto de cidades, as distâncias entre elas e duas cidades A e B, determinar um caminho (trajeto) mais curto de A até B.
- Árvore Geradora de Peso Mínimo: dado um conjunto de computadores, onde cada par de computadores pode ser ligado usando uma quantidade de fibra ótica, encontrar uma rede interconectando-os que use a menor quantidade de fibra ótica possível.
- Emparelhamento máximo: dado um conjunto de pessoas e um conjunto de vagas para diferentes empregos, onde cada pessoa é qualificada para certos empregos e cada vaga deve ser ocupada por exatamente uma pessoa, encontrar um modo de empregar o maior número possível de pessoas.

Aplicações

- Problema do Caixeiro Viajante: dado um conjunto de cidades, encontrar um ciclo que sai de uma cidade, passa por todas as cidades e volta para a cidade inicial tal que a distância total a ser percorrida seja menor possível.
- Problema Chinês do Correio: dado o conjunto das ruas de um bairro, encontrar um caminho que passa por todas as ruas voltando ao ponto inicial tal que a distância total a ser percorrida seja menor possível.

Representação Interna de Grafos

- A complexidade dos algoritmos para solução de problemas modelados por grafos depende fortemente da sua representação interna.
- Existem duas representações canônicas: matriz de adjacência e listas de adjacência.
- O uso de uma ou outra num determinado algoritmo depende da natureza das operações que ditam a complexidade do algoritmo.

Matriz de adjacência

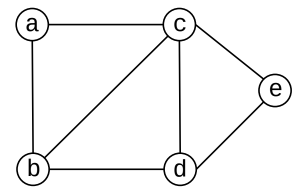
- Seja $G = (V, E)$ um grafo simples (orientado ou não).
- A *matriz de adjacência* de G é uma matriz quadrada A de ordem $|V|$, cujas linhas e colunas são indexadas pelos vértices em V, e tal que:

$$A[i,j] = \begin{cases} 1 & \text{se } (i,j) \in E, \\ 0 & \text{caso contrário.} \end{cases}$$

- Note que se G é não-orientado, então a matriz A correspondente é simétrica.

Matriz de adjacência

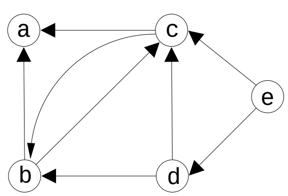
- Exemplo de um grafo e a matriz de adjacência correspondente.



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	1	0
c	1	1	0	1	1
d	0	1	1	0	1
e	0	0	1	1	0

Matriz de adjacência

- Exemplo de um grafo orientado e a matriz de adjacência correspondente.



	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

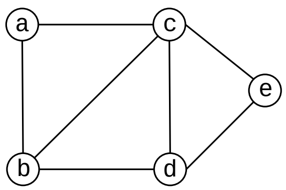
Listas de adjacência

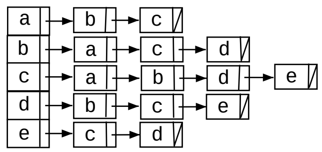
- Seja $G = (V, E)$ um grafo simples (orientado ou não).
- A representação de G por uma *lista de adjacências* consiste no seguinte.

  Para cada vértice v, temos uma lista ligada $\mathrm{Adj}[v]$ dos vértices adjacentes a v, ou seja, w aparece em $\mathrm{Adj}[v]$ se (v, w) é uma aresta de G. Os vértices podem estar em qualquer ordem em uma lista.

Listas de adjacência

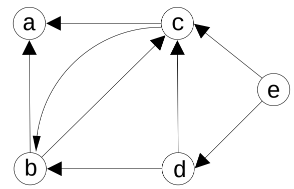
- Exemplo de um grafo não-orientado e a listas de adjacência correspondente.

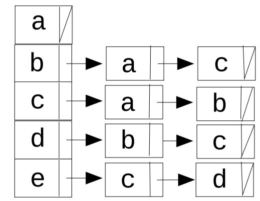




Lista de adjacências

Exemplo de um grafo orientado e a lista de adjacências correspondente.





Matriz $\times$ Lista de adjacência

- Matriz de adjacência: é fácil verificar se (i, j) é uma aresta de G.
- Lista de adjacência: é fácil descobrir os vértices adjacentes a um dado vértice v (ou seja, listar $\mathrm{Adj}[v]$).
- Matriz de adjacência: espaço $\Theta(|V|^2)$. Adequada a grafos densos ($|E| = \Theta(|V|^2)$).
- Lista de adjacência: espaço $\Theta(|V| + |E|)$. Adequada a grafos esparsos ($|E| = \Theta(|V|)$).

Extensões

- Há outras alternativas para representar grafos, mas matrizes e listas de adjacência são as mais usadas.
- Elas podem ser adaptadas para representar grafos ponderados, grafos com laços e arestas múltiplas, grafos com pesos nos vértices etc.
- Para determinados problemas é essencial ter estruturas de dados adicionais para melhorar a eficiência dos algoritmos.

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 – BFS)
  Busca em profundidade (depth-first search – DFS)
- 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 Θ 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 ∞ (*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.

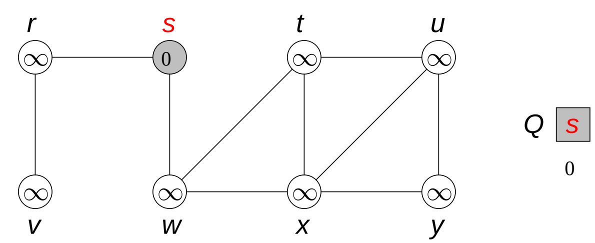
Busca em largura

- 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.

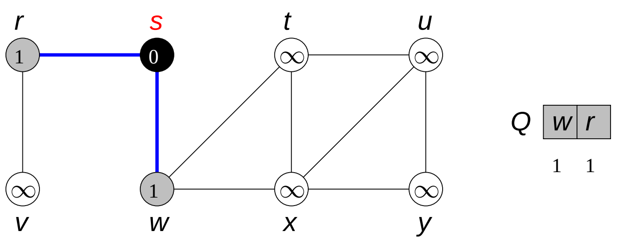
Busca em largura

- 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".

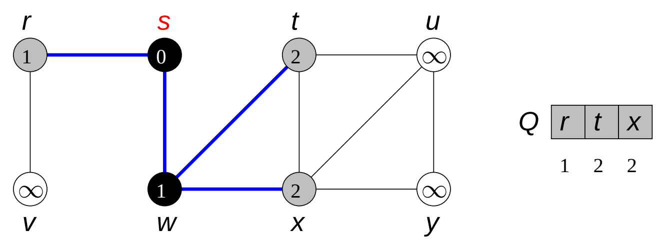
Exemplo (CLRS)



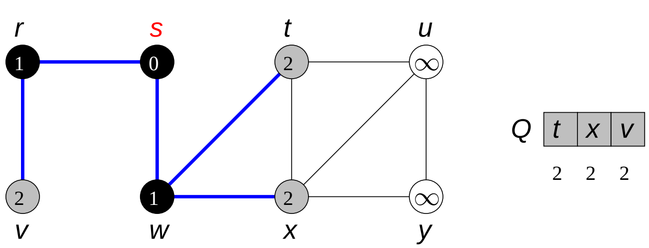
Exemplo (CLRS)



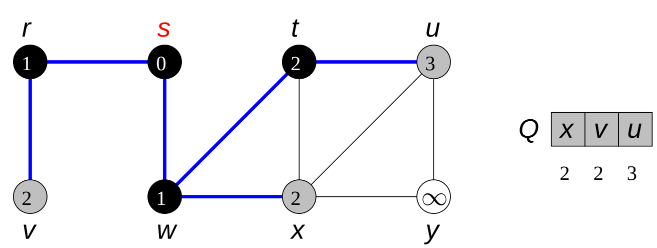
Exemplo (CLRS)



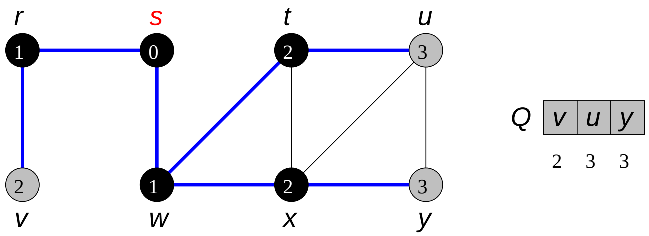
Exemplo (CLRS)



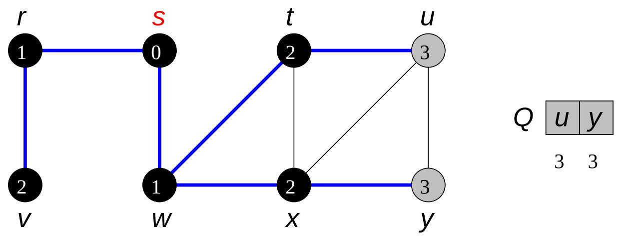
Exemplo (CLRS)



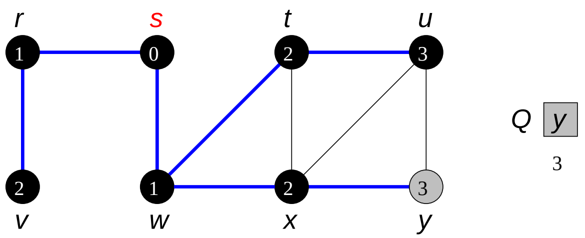
Exemplo (CLRS)



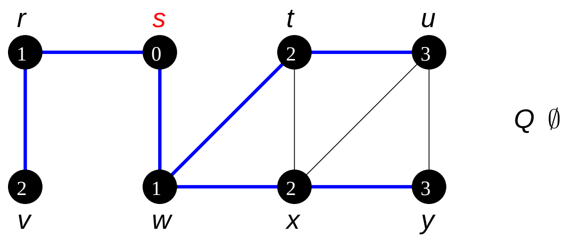
Exemplo (CLRS)



Exemplo (CLRS)



Exemplo (CLRS)



Cores

- Para cada vértice v guarda-se sua cor atual $\text{cor}[v]$ 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.

$\text{BFS}(G, s)$

$\triangleright$ Inicialização
para cada $u \in V[G] - \{s\}$ **faça**
  $\text{cor}[u] \leftarrow \text{branco}$
  $d[u] \leftarrow \infty$
  $\pi[u] \leftarrow \text{NIL}$
$\text{cor}[s] \leftarrow \text{cinza}$
$d[s] \leftarrow 0$
$\pi[s] \leftarrow \text{NIL}$

Busca em largura

$Q \leftarrow \emptyset$
$\text{Enqueue}(Q, s)$
enquanto $Q \neq \emptyset$ **faça**
  $u \leftarrow \text{Dequeue}(Q)$
  para cada $v \in \text{Adj}[u]$ **faça**
    se $cor[v] = \text{branco}$ **então**
      $\text{cor}[v] \leftarrow \text{cinza}$
      $d[v] \leftarrow d[u] + 1$
      $\pi[v] \leftarrow u$
      $\text{Enqueue}(Q, v)$
  $\text{cor}[u] \leftarrow \text{preto}$

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 BFS é $O(V+E)$.

Agora falta mostrar que BFS 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.

Outro lema

Lema 4. Se v pertence à árvore T induzida por $\pi[\,]$, então o caminho de s a v em T tem comprimento $d[v]$.

Prova:

Segue por indução no número de operações ENQUEUE.

Árvore

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

Então após a execução de BFS,

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

e

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

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 BFS). 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, $v \in \mathrm{Adj}[w]$) 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)
  se v = s então
      imprime s
  senão
      se π[v] = NIL então
          imprime não existe caminho de s a v.
      senão
          Print-Path(G, s, π[v])
          imprime v.
```

Complexidade: $O(\text{comprimento do caminho}) = O(V)$.

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 u 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".

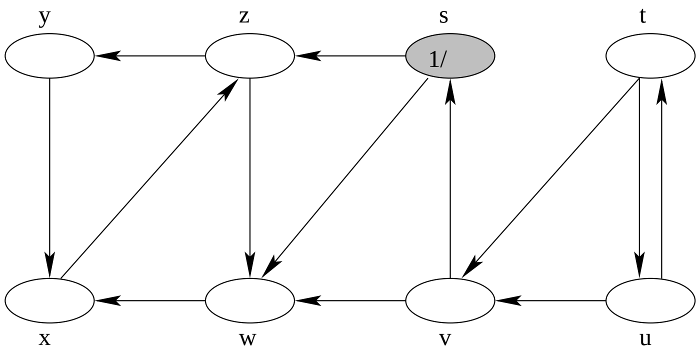
Estampas/rótulos

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

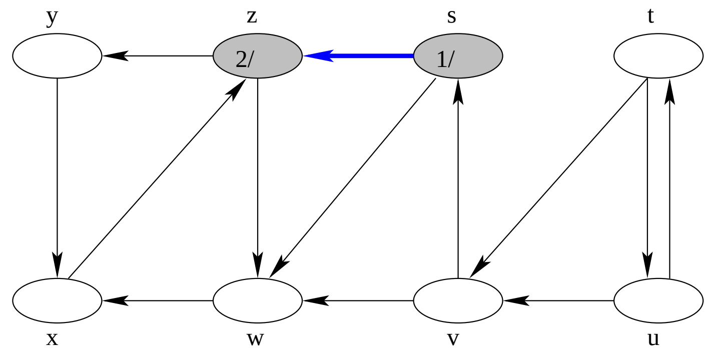
- $d[x]$: instante de descoberta de x.
  Neste instante x torna-se cinza.
- $f[x]$: instante de finalização de x.
  Neste instante x torna-se preto.

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

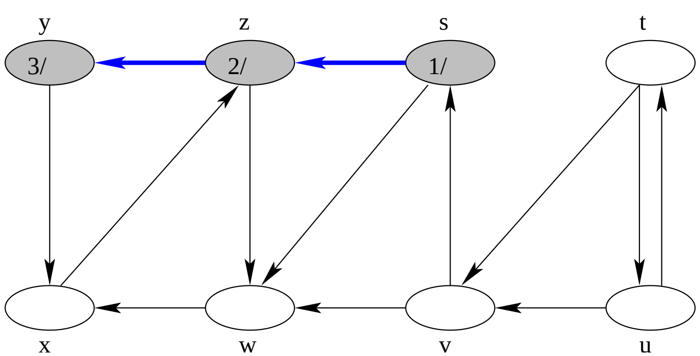
Exemplo



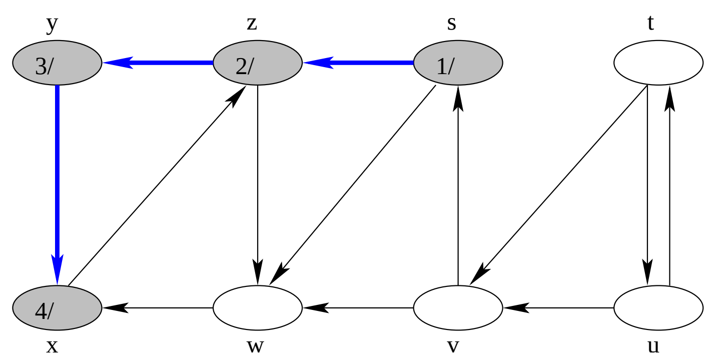
Exemplo



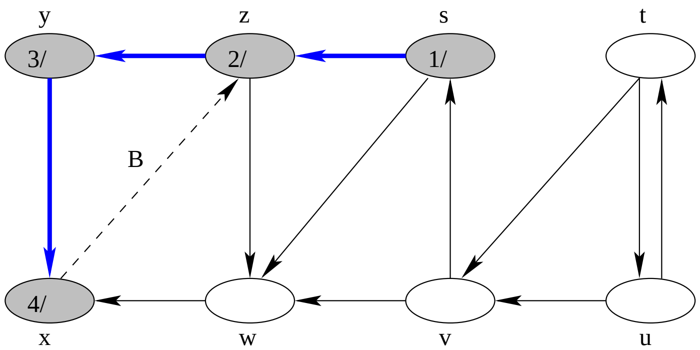
Exemplo



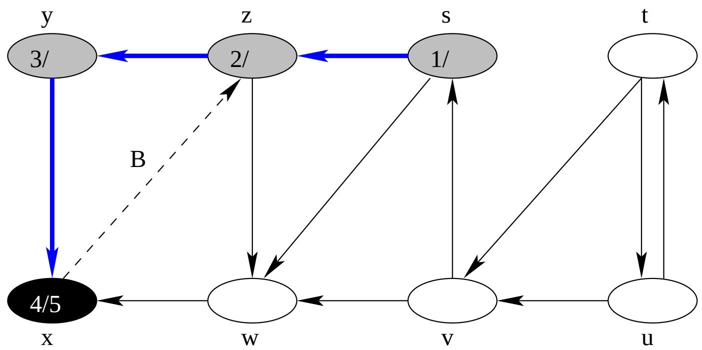
Exemplo



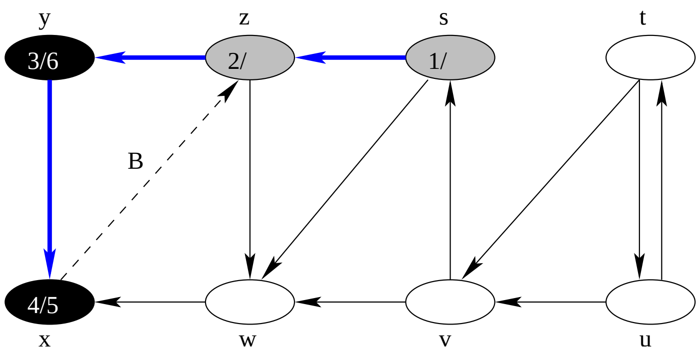
Exemplo



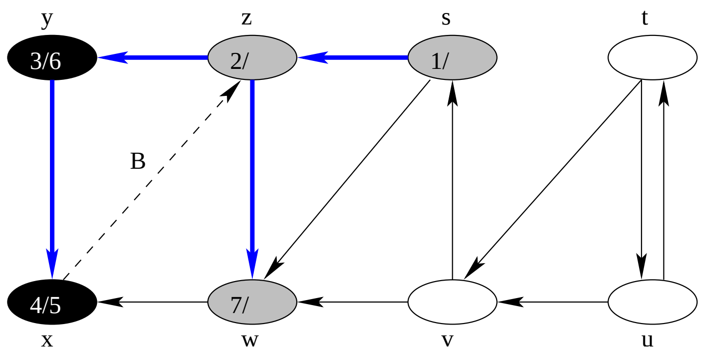
Exemplo



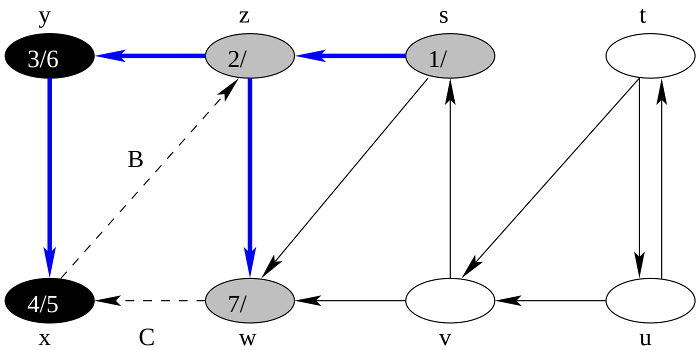
Exemplo



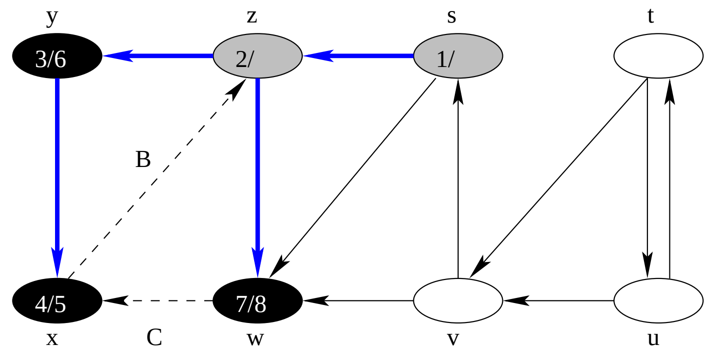
Exemplo



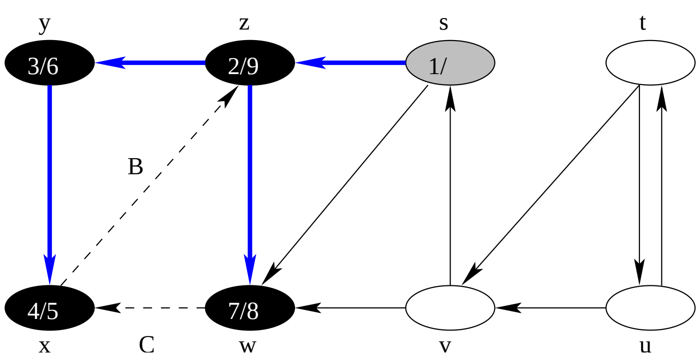
Exemplo



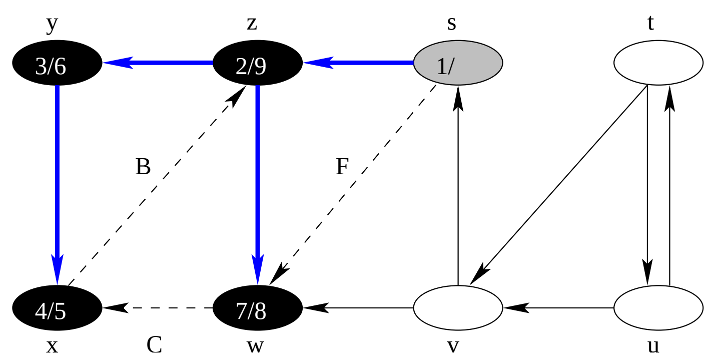
Exemplo



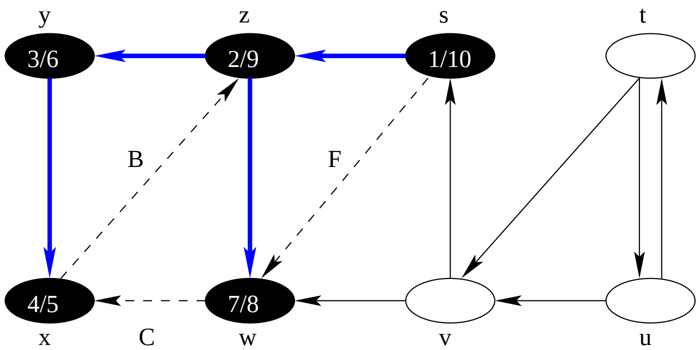
Exemplo



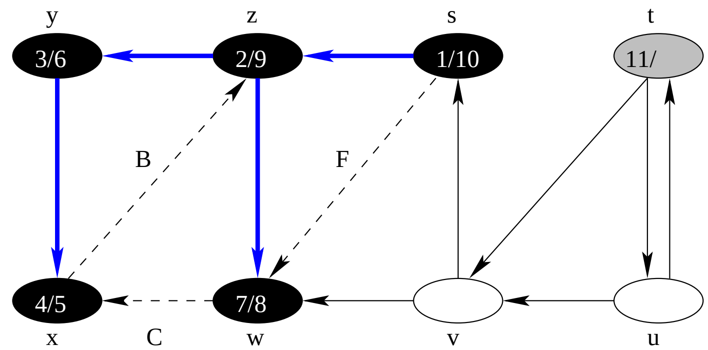
Exemplo



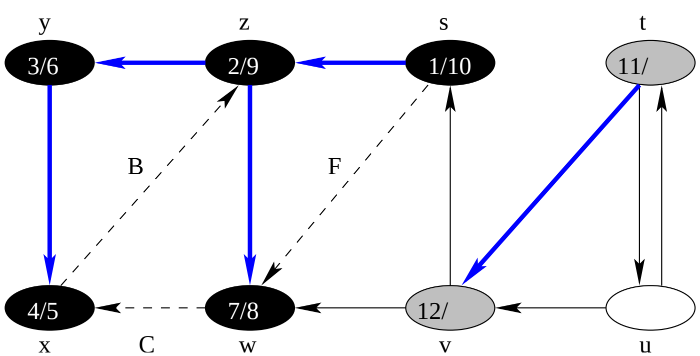
Exemplo



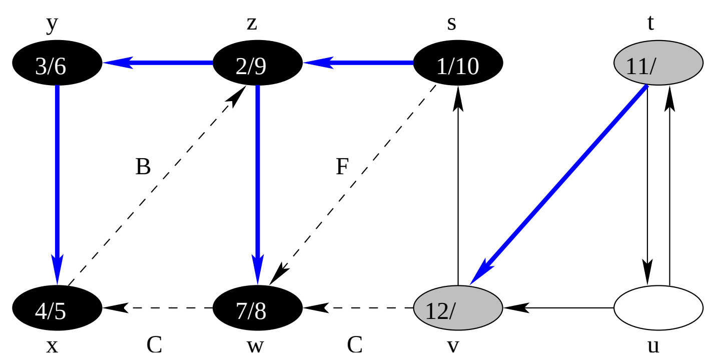
Exemplo



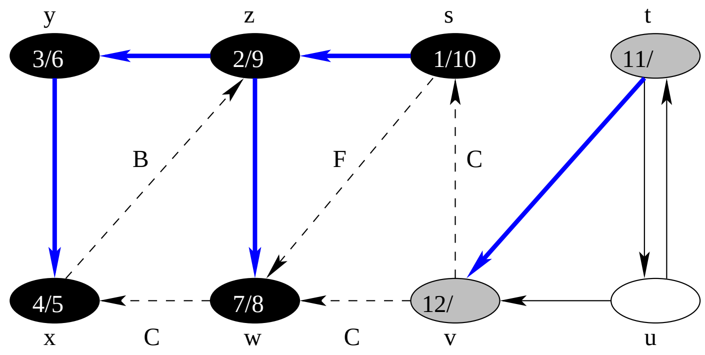
Exemplo



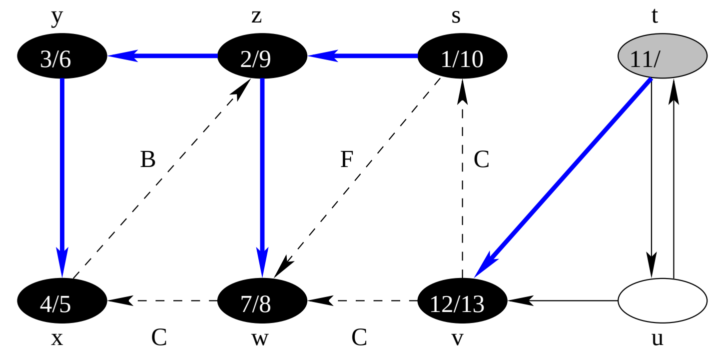
Exemplo



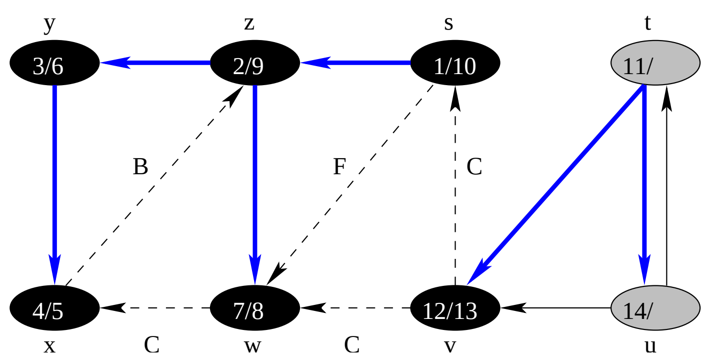
Exemplo



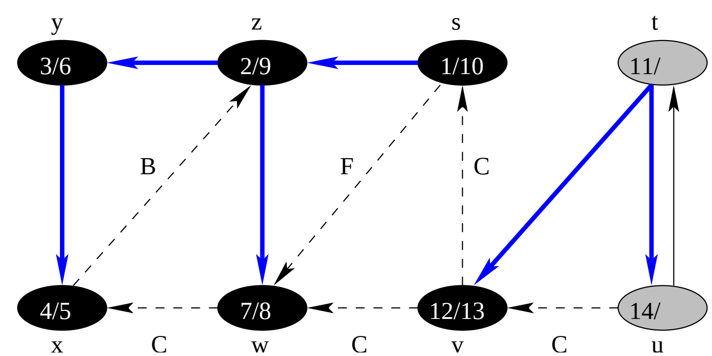
Exemplo



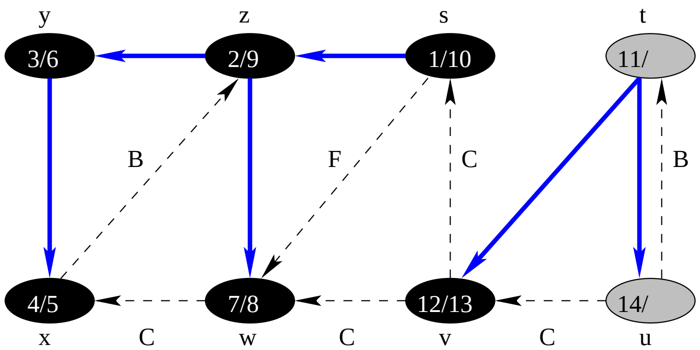
Exemplo



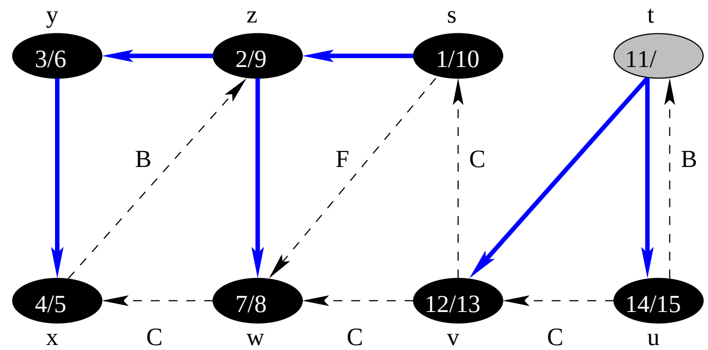
Exemplo



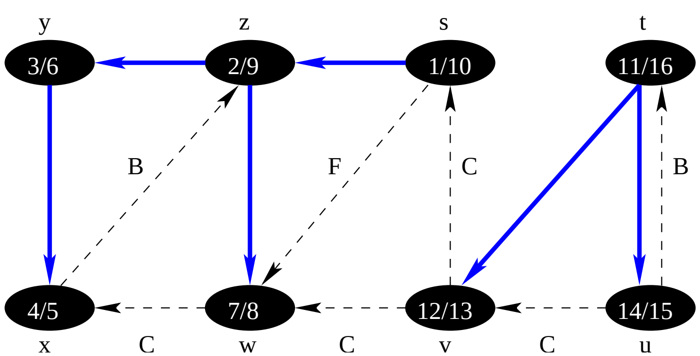
Exemplo



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 $\quad$ $\mathtt{cor}[u] \leftarrow \text{branco}$
3 $\quad$ $\pi[u] \leftarrow$ NIL
4 $\text{tempo} \leftarrow 0$
5 **para cada** $u \in V[G]$ **faça**
6 $\quad$ **se** $\mathtt{cor}[u] = \text{branco}$
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 $\mathtt{cor}[u] \leftarrow \text{cinza}$
2 $\text{tempo} \leftarrow \text{tempo} + 1$
3 $d[u] \leftarrow \text{tempo}$
4 **para cada** $v \in \text{Adj}[u]$ **faça**
5 $\quad$ **se** $\mathtt{cor}[v] = \text{branco}$
6 $\quad\quad$ **então** $\pi[v] \leftarrow u$
7 $\quad\quad\quad\quad$ DFS-VISIT(v)
8 $\mathtt{cor}[u] \leftarrow \text{preto}$
9 $f[u] \leftarrow \text{tempo} \leftarrow \text{tempo} + 1$

Algoritmo DFS

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

Consumo de tempo

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

Algoritmo DFS-VISIT

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

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

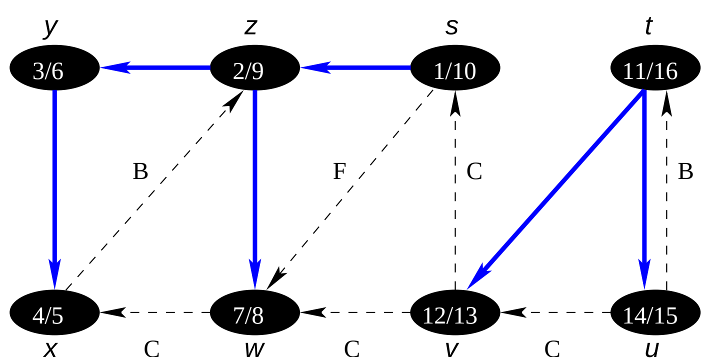
  $\sum_{v \in V} |\mathrm{Adj}(v)| = \Theta(E)$.

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

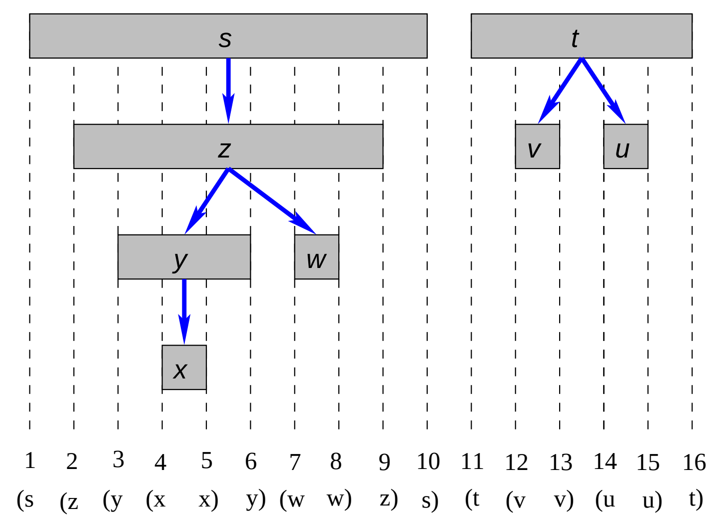
Estrutura de parênteses

- 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)

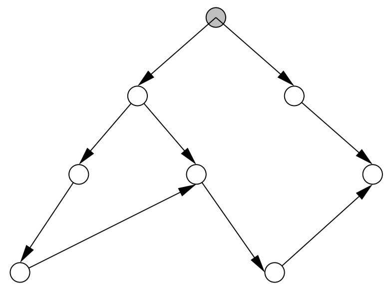
Um vértice v é um descendente próprio de u na Floresta de BP se e somente se $d[u] < d[v] < f[v] < f[u]$.

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.



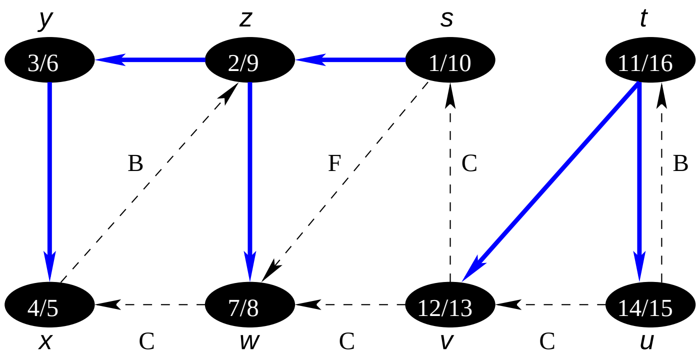
Classificação de arestas

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

Ela classifica as arestas em quatro tipos:

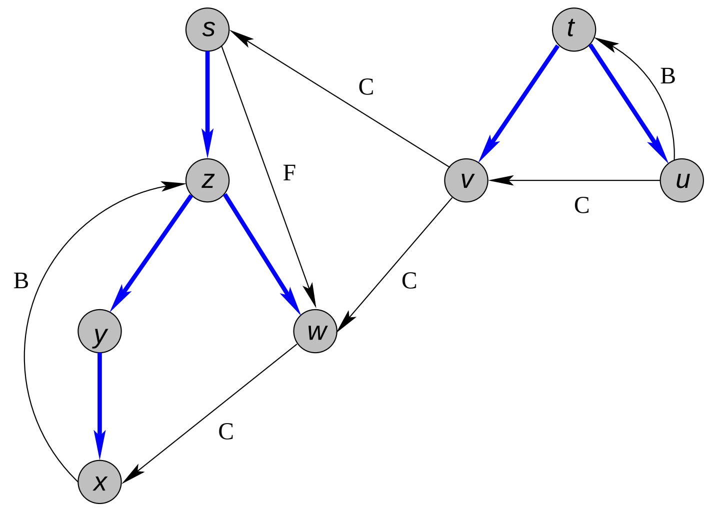
- Arestas da árvore (tree edges): arestas que pertencem à Floresta de BP.
- Arestas de retorno (backward edges): arestas (u, v) ligando um vértice u a um ancestral v na Árvore de BP.
- Arestas de avanço (forward edges): arestas (u, v) ligando um vértice u a um descendente próprio v na Árvore de BP.
- Arestas de cruzamento (cross edges): 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



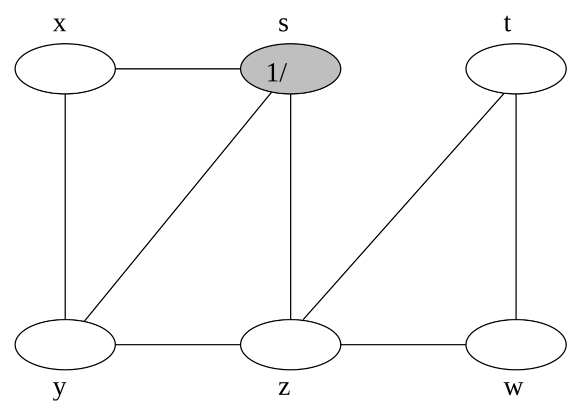
Grafos não-orientados

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.

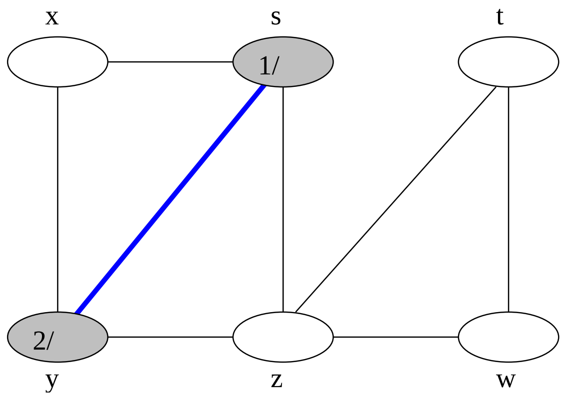
Para grafos não-orientados, existem apenas dois tipos de arestas.

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.

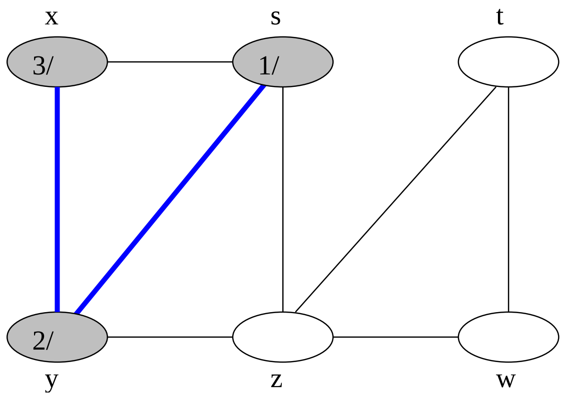
Exemplo



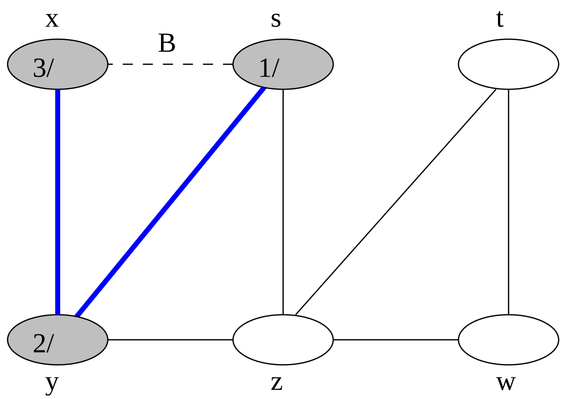
Exemplo



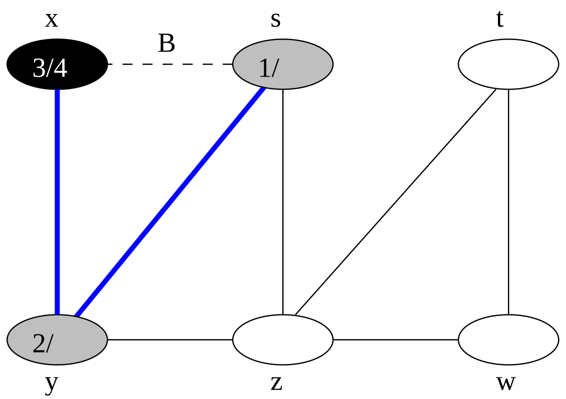
Exemplo



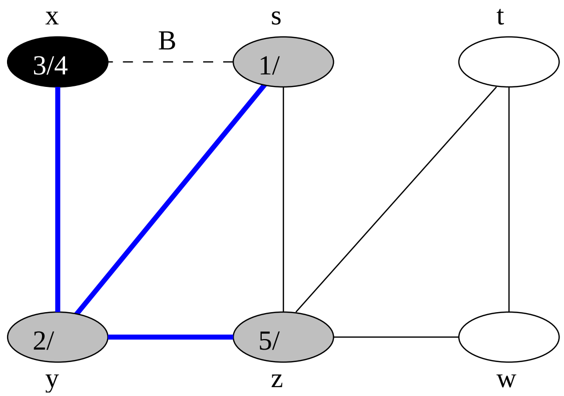
Exemplo



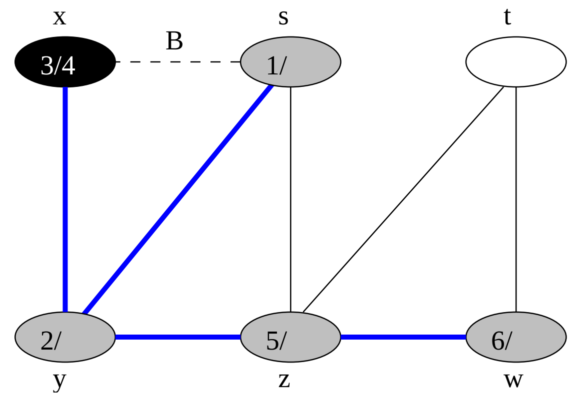
Exemplo



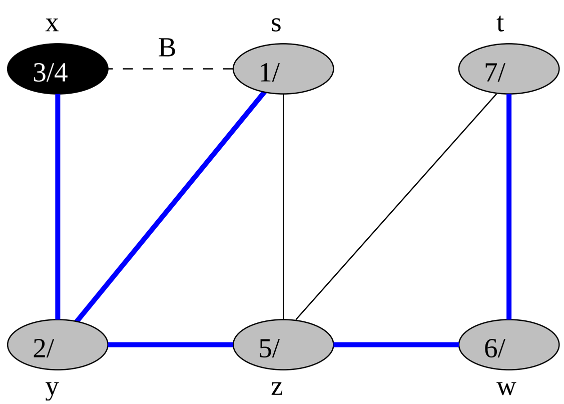
Exemplo



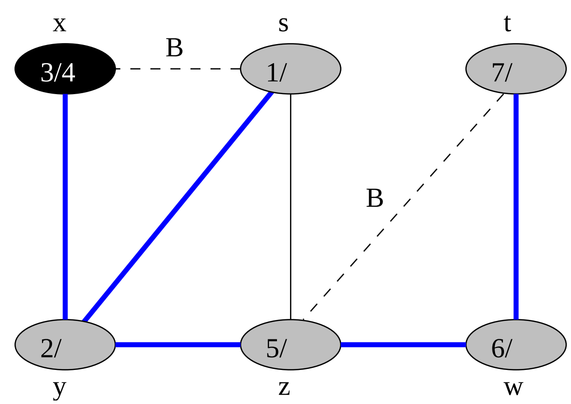
Exemplo



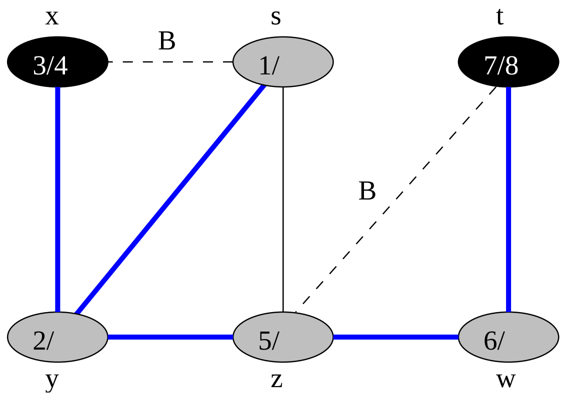
Exemplo



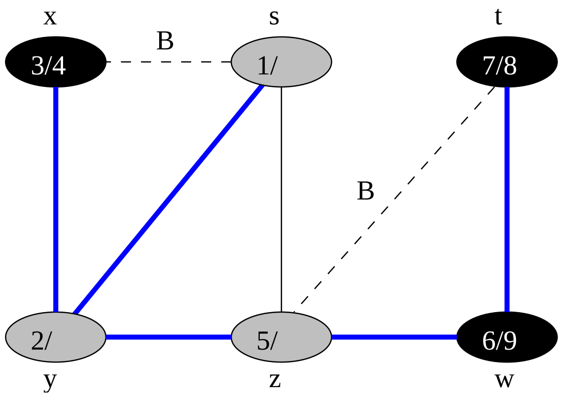
Exemplo



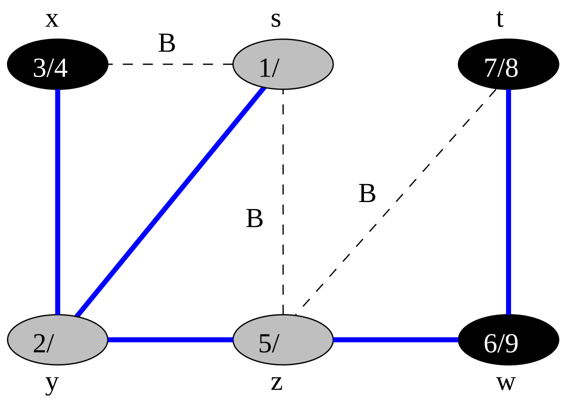
Exemplo



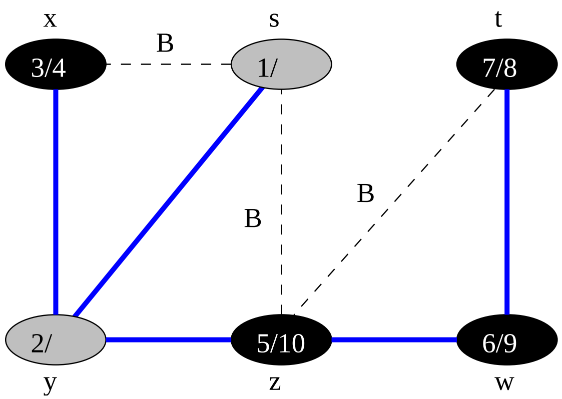
Exemplo



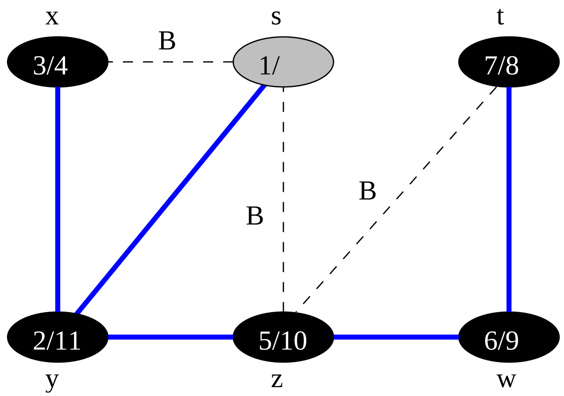
Exemplo



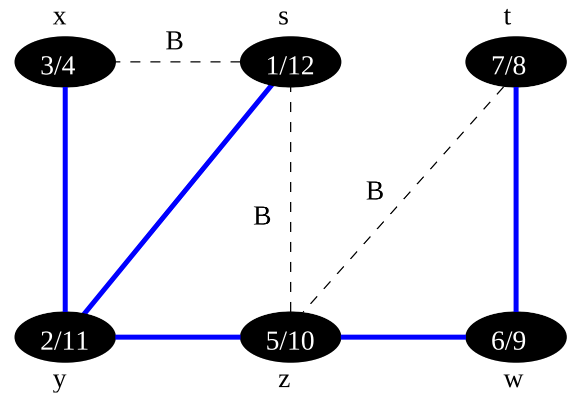
Exemplo



Exemplo



Exemplo

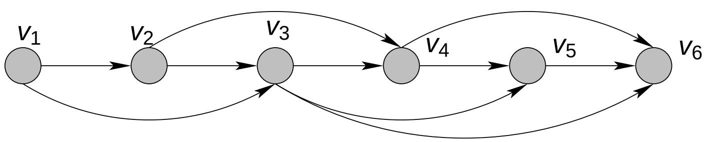


Ordenação Topológica

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

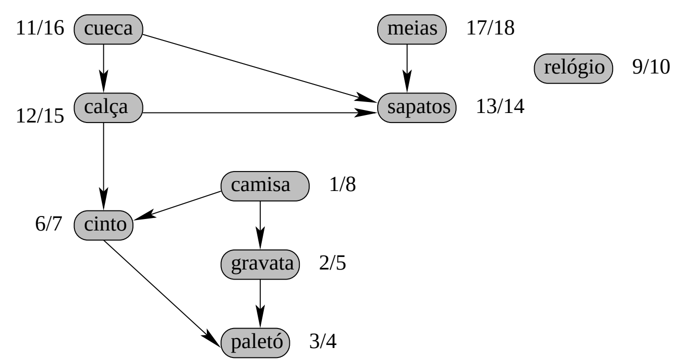
$v_1 \quad v_2 \quad v_3 \quad \ldots \quad v_{n-2} \quad v_{n-1} \quad v_n$

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

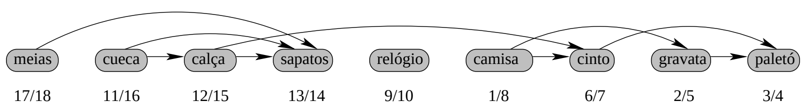


Ordenação Topológica

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

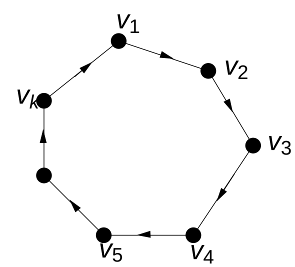


Ordenação Topológica



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:

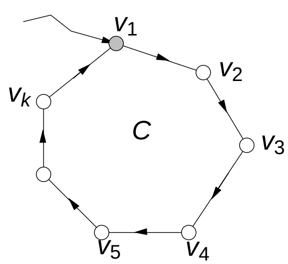
Suponha que (u, v) é uma aresta de retorno.

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.