Projeto e Análise de Algoritmos

A. G. Silva

Baseado nos materiais de
Souza, Silva, Lee, Rezende, Miyazawa – Unicamp
Feofiloff – USP

04 de maio de 2018

Conteúdo programático

- Introdução (4 horas/aula)
- Notação Assintótica e Crescimento de Funções (4 horas/aula)
- Recorrências (4 horas/aula)
- Divisão e Conquista (12 horas/aula)
- Buscas (4 horas/aula)
- Grafos (4 horas/aula)
- Algoritmos Gulosos (8 horas aula)
- Programação Dinâmica (8 horas/aula)
- NP-Completude e Reduções (6 horas/aula)
- Algoritmos Aproximados e Busca Heurística (6 horas/aula)

Cronograma

- **02mar** – Apresentação da disciplina. Introdução.
- **09mar** – *Prova de proficiência/dispensa.*
- **16mar** – Notação assintótica. Recorrências.
- **23mar** – *Dia não letivo.* Exercícios.
- **30mar** – *Dia não letivo.* Exercícios.
- **06abr** – Recorrências. Divisão e conquista.
- **13abr** – Divisão e conquista. Ordenação.
- **20abr** – Ordenação. Estatística de ordem.
- **27abr** – **Primeira avaliação.**
- **04mai** – Estatística de ordem. Buscas. Grafos.
- **11mai** – Algoritmos gulosos.
- **18mai** – Algoritmos gulosos.
- **25mai** – Programação dinâmica.
- **01jun** – *Dia não letivo.* Exercícios.
- **08jun** – *Semana Acadêmica.* Exercícios.
- **15jun** – Programação dinâmica. NP-Completude e reduções.
- **22jun** – Exercícios *(copa)*.
- **29jun** – **Segunda avaliação.**
- **06jul** – **Avaliação substitutiva** *(opcional)*.

Estatísticas de Ordem

Estatísticas de Ordem (Problema da Seleção)

- Estamos interessados em resolver o

  Problema da Seleção:
  Dado um conjunto A de n números reais e um inteiro i, determinar o i-ésimo menor elemento de A.

- Casos particulares importantes:

  Mínimo : $i = 1$

  Máximo : $i = n$

  Mediana : $i = \lfloor \frac{n+1}{2} \rfloor$ (mediana inferior)

  Mediana : $i = \lceil \frac{n+1}{2} \rceil$ (mediana superior)

Mínimo

Recebe um vetor $A[1 \ldots n]$ e devolve o mínimo do vetor.

```
MÍNIMO(A, n)
1   mín ← A[1]
2   para j ← 2 até n faça
3       se mín > A[j]
4           então mín ← A[j]
5   devolva mín
```

Número de comparações: $n - 1 = \Theta(n)$

Aula passada: não é possível fazer com menos comparações.

Mínimo e máximo

Recebe um vetor $A[1 \ldots n]$ e devolve o mínimo e o máximo do vetor.

```
MinMax(A, n)
1   mín ← máx ← A[1]
2   para j ← 2 até n faça
3       se A[j] < mín
4           então mín ← A[j]
5       se A[j] > máx
6           então máx ← A[j]
7   devolva (mín, máx)
```

Número de comparações: $2(n-1) = 2n-2 = \Theta(n)$

É possível fazer melhor!

Mínimo e máximo

- Processe os elementos em pares. Para cada par, compare o menor com o mínimo atual e o maior com o máximo atual. Isto resulta em 3 comparações para cada 2 elementos.
- Se n for ímpar, inicialize o mínimo e o máximo como sendo o primeiro elemento.
- Se n for par, inicialize o mínimo e o máximo comparando os dois primeiros elementos.

Número de comparações:

$3\lfloor n/2 \rfloor$	se n for ímpar
$3\lfloor n/2 \rfloor - 2$	se n for par

Pode-se mostrar que isto é o melhor possível.
(Exercício $\star$ do CLRS)

Problema da Seleção – primeira solução

Recebe $A[1 \ldots n]$ e i tal que $1 \leq i \leq n$
e devolve valor do i-ésimo menor elemento de $A[1 \ldots n]$

SELECT-ORD(A, n, i)
1 ORDENE(A, n)
2 **devolva** $A[i]$

ORDENE pode ser *MergeSort* ou *HeapSort*.

A complexidade de tempo de SELECT-ORD é $O(n \lg n)$.

Será que não dá para fazer melhor que isso?

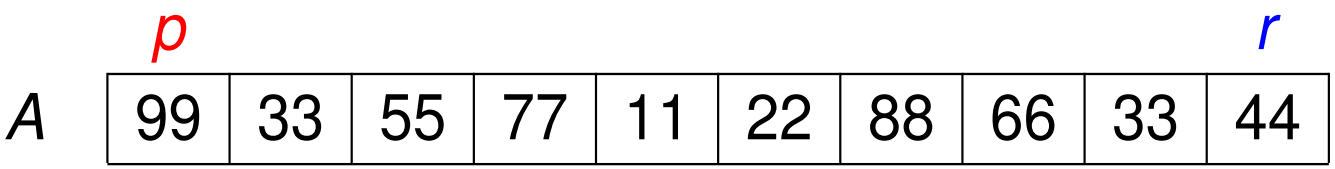
Afinal, consigo achar o mínimo e/ou máximo em tempo $O(n)$.

Relembrando – Partição

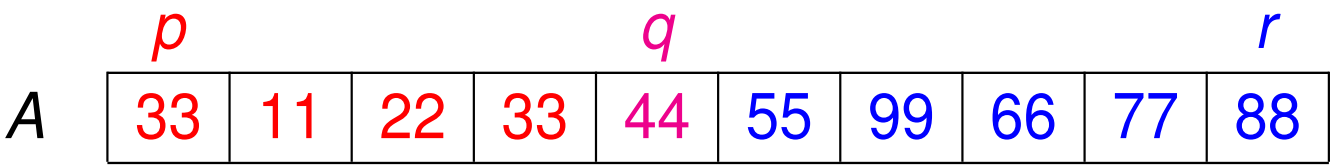
Problema: rearranjar um dado vetor $A[p \ldots r]$ e devolver um índice q, $p \leq q \leq r$, tais que

$$A[p \ldots q-1] \leq A[q] < A[q+1 \ldots r]$$

Entrada:



Saída:



Relembrando – Particione

Rearranja $A[p \ldots r]$ de modo que $p \le q \le r$ e $A[p \ldots q-1] \le A[q] < A[q+1 \ldots r]$.

PARTICIONE(A, p, r)

```
1  x ← A[r]   ▷ x é o "pivô"
2  i ← p−1
3  para j ← p até r − 1 faça
4       se A[j] ≤ x
5            então i ← i + 1
6                  A[i] ↔ A[j]
7  A[i+1] ↔ A[r]
8  devolva i + 1
```

Problema da Seleção – segunda solução

Suponha que queremos achar o i-ésimo menor de $A[1 \ldots n]$.

- Executamos PARTICIONE e este rearranja o vetor e devolve um índice k tal que

$$A[1 \ldots k-1] \leq A[k] < A[k+1 \ldots n].$$

- Eis a idéia do algoritmo:
  - Se $i = k$, então o pivô $A[k]$ é o i-ésimo menor! (Yeesss!)
  - Se $i < k$, então o i-ésimo menor está em $A[1 \ldots k-1]$;
  - Se $i > k$, então o i-ésimo menor está em $A[k+1 \ldots n]$.

Problema da Seleção – segunda solução

Recebe $A[p \ldots r]$ e i tal que $1 \leq i \leq r-p+1$
e devolve o i-ésimo menor elemento de $A[p \ldots r]$.

```
SELECT-NL(A, p, r, i)
1   se p = r
2       então devolva A[p]
3   q ← PARTICIONE(A, p, r)
4   k ← q − p + 1
5   se i = k    ▷ pivô é o i-ésimo menor!
6       então devolva A[q]
7       senão se i < k
8           então devolva SELECT-NL(A, p, q − 1, i)
9           senão devolva SELECT-NL(A, q + 1, r, i − k)
```

Segunda solução – complexidade

SELECT-NL(A, p, r, i)		Tempo
1	**se** $p = r$	?
2	**então devolva** $A[p]$	?
3	$q \leftarrow$ PARTICIONE(A, p, r)	?
4	$k \leftarrow q - p + 1$	?
5	**se** $i = k$	?
6	**então devolva** $A[q]$	?
7	**senão se** $i < k$	?
8	**então devolva** SELECT-NL$(A, p, q - 1, i)$	?
9	**senão devolva** SELECT-NL$(A, q + 1, r, i - k)$	?

$T(n) =$ complexidade de tempo no pior caso quando $n = r - p + 1$

Segunda solução – complexidade

SELECT-NL(A, p, r, i)		Tempo
1	**se** $p = r$	$\Theta(1)$
2	**então devolva** $A[p]$	$O(1)$
3	$q \leftarrow$ PARTICIONE(A, p, r)	$\Theta(n)$
4	$k \leftarrow q - p + 1$	$\Theta(1)$
5	**se** $i = k$	$\Theta(1)$
6	**então devolva** $A[q]$	$O(1)$
7	**senão se** $i < k$	$O(1)$
8	**então devolva** SELECT-NL$(A, p, q-1, i)$	$T(k-1)$
9	**senão devolva** SELECT-NL$(A, q+1, r, i-k)$	$T(n-k)$

$$T(n) = \max\{T(k-1), T(n-k)\} + \Theta(n)$$

$T(n) \in \Theta(n^2)$ (Exercício)

Segunda solução – complexidade

- A complexidade de SELECT-NL no pior caso é $\Theta(n^2)$.
- Então é melhor usar SELECT-ORD?
- Não, SELECT-NL é muito eficiente na prática.
- Vamos mostrar que no caso médio SELECT-NL tem complexidade $O(n)$.

SELECT aleatorizado

O pior caso do SELECT-NL ocorre devido a uma escolha infeliz do pivô.
Um modo de evitar isso é usar aleatoriedade (como no QUICKSORT-ALEATÓRIO).

PARTICIONE-ALEATÓRIO(A, p, r)
1 $\quad j \leftarrow$ RANDOM(p, r)
2 $\quad A[j] \leftrightarrow A[r]$
3 $\quad$ **devolva** PARTICIONE(A, p, r)

Algoritmo SELECT-ALEAT

Recebe $A[p \ldots r]$ e i tal que $1 \leq i \leq r-p+1$
e devolve o i-ésimo menor elemento de $A[p \ldots r]$

```
SELECT-ALEAT(A, p, r, i)
1   se p = r
2       então devolva A[p]
3   q ← PARTICIONE-ALEATÓRIO(A, p, r)
4   k ← q − p + 1
5   se i = k   ▷ pivô é o i-ésimo menor
6       então devolva A[q]
7       senão se i < k
8           então devolva SELECT-ALEAT(A, p, q − 1, i)
9           senão devolva SELECT-ALEAT(A, q + 1, r, i − k)
```

Análise do caso médio

Recorrência para o caso médio de SELECT-ALEAT.

$T(n)$ = complexidade de tempo médio de SELECT-ALEAT.

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) \leq \frac{1}{n} \sum_{k=1}^{n} T(\max\{k-1, n-k\}) + \Theta(n).$$

$T(n)$ é $\Theta(???)$.

Análise do caso médio

$$\begin{aligned} T(n) &\leq \frac{1}{n}\sum_{k=1}^{n} T(\max\{k-1, n-k\}) + an \\ &\leq \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) + an \end{aligned}$$

pois

$$\max\{k-1, n-k\} = \begin{cases} k-1 & \text{se } k > \lceil n/2 \rceil, \\ n-k & \text{se } k \leq \lceil n/2 \rceil. \end{cases}$$

Se n é par, cada termo de $T(\lceil n/2 \rceil)$ a $T(n-1)$ aparece exatamente duas vezes na somatória.
Se n é ímpar, esses termos aparecem duas vezes e $T(\lfloor n/2 \rfloor)$ aparece uma vez.

Demonstração: $T(n) \leq cn$

$$
\begin{aligned}
T(n) &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) + an \\
&\overset{\text{hi}}{\leq} \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an
\end{aligned}
$$

Demonstração: $T(n) \leq cn$

$$
\begin{aligned}
T(n) &= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2}\right) + an \\
&\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-1-1)(n/2-1)}{2}\right) + an \qquad (**) \\
&= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an \\
&\leq \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) \leq cn.
\end{aligned}
$$

$$\lfloor n/2 \rfloor \geq n/2 - 1 \qquad (*)$$

Considerando (*) válido, o numerador (**) é menor, Então um valor **menor** será subtraído. Logo, o resultado da subtração será **maior**. Portanto, a desigualdade está correta.

Isto funciona se $c > 4a$ e $n \geq 2c/(c-4a)$.
Logo, $T(n) = O(n)$.

Conclusão

A complexidade de tempo de SELECT-ALEAT no caso médio é $O(n)$.

Na verdade,

A complexidade de tempo de SELECT-ALEAT no caso médio é $\Theta(n)$.

Veremos depois:
Algoritmo que resolve o Problema da Seleção em tempo linear (no pior caso).

Problema da Seleção – terceira solução

Problema da Seleção:
Dado um conjunto A de n números reais e um inteiro i, determinar o i-ésimo menor elemento de A.

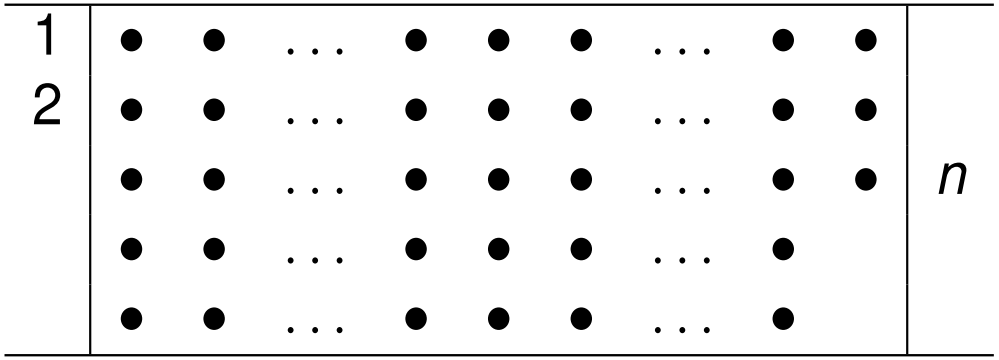
Veremos um algoritmo linear para o Problema da Seleção.

BFPRT = Blum, Floyd, Pratt, Rivest e Tarjan

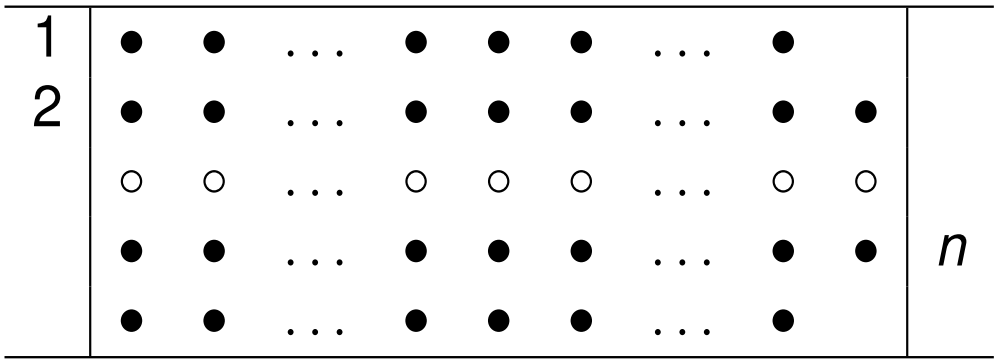
Para simplificar a exposição, vamos supor que os elementos em A são distintos.

Problema da Seleção – terceira solução

1. Divida os n elementos em $\lfloor \frac{n}{5} \rfloor$ subconjuntos de 5 elementos e um subconjunto de $n \bmod 5$ elementos.



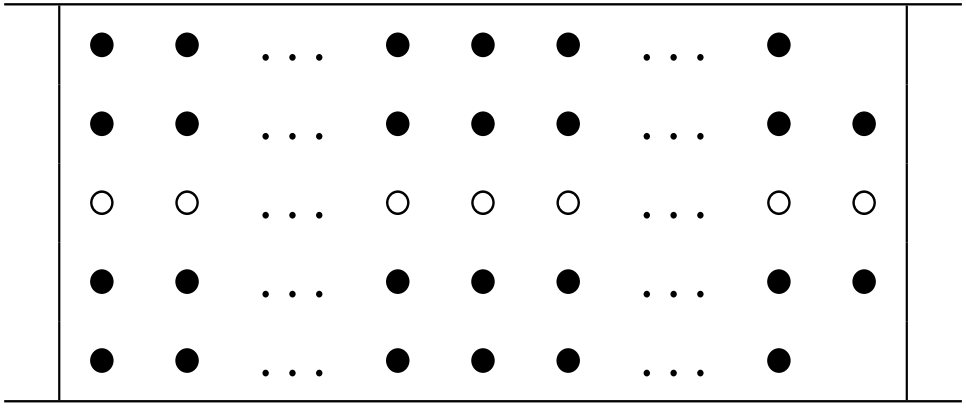
2. Encontre a mediana de cada um dos $\lceil \frac{n}{5} \rceil$ subconjuntos.



Na figura acima, cada subconjunto está em ordem crescente, de cima para baixo.

Problema da Seleção – terceira solução

3. Determine, recursivamente, a mediana x das medianas dos subconjuntos de no máximo 5 elementos.



A figura acima é a mesma que a anterior, com as colunas ordenadas pela mediana de cada grupo. A ordem dos elementos em cada coluna permanece inalterada.
Por simplicidade de exposição, supomos que a última coluna permanece no mesmo lugar.
Note que o algoritmo não ordena as medianas!

Problema da Seleção - terceira solução

4. Usando x como pivô, particione o conjunto original A criando dois subconjuntos $A_<$ e $A_>$, onde
   - $A_<$ contém os elementos $< x$ e
   - $A_>$ contém os elementos $> x$.

   Se a posição final de x após o particionamento é k, então $|A_<| = k - 1$ e $|A_>| = n - k$.

Problema da Seleção - terceira solução

5. Finalmente, para encontrar o i-ésimo menor elemento do conjunto, compare i com a posição k de x após o particionamento:
   - Se $i = k$, x é o elemento procurado;
   - Se $i < k$, então determine recursivamente o i-ésimo menor elemento do subconjunto $A_<$;
   - Senão, determine recursivamente o $(i - k)$-ésimo menor elemento do subconjunto $A_>$.

Note que esta parte é idêntica ao feito em SELECT-NL e em SELECT-ALEAT. O que diferencia este algoritmo dos outros é a escolha do pivô. Escolhendo-se a mediana das medianas, vamos poder garantir que nenhum dos lados é muito “grande”.

Terceira solução – complexidade

$T(n)$: complexidade de tempo no pior caso

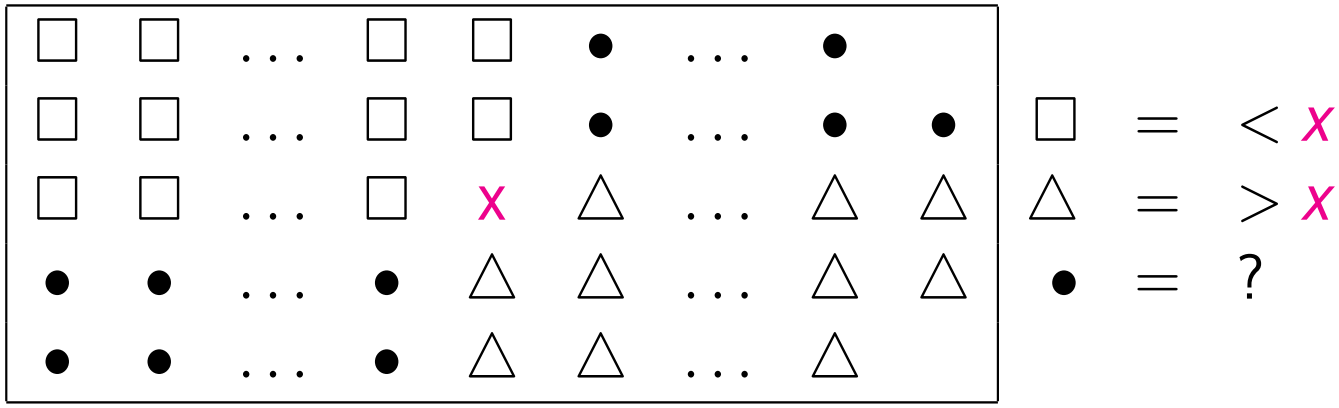
1. Divisão em subconjuntos de 5 elementos. $\Theta(n)$
2. Encontrar a mediana de cada subconjunto. $\Theta(n)$
3. Encontrar x, a mediana das medianas. $T(\lceil n/5 \rceil)$
4. Particionamento com pivô x. $O(n)$
5. Encontrar o i-ésimo menor de $A_<$ $T(k-1)$
   OU encontrar o $i-k$-ésimo menor de $A_>$. $T(n-k)$

Temos então a recorrência

$$T(n) = T(\lceil n/5 \rceil) + T(\max\{k-1, n-k\}) + \Theta(n)$$

Terceira Solução - Complexidade

O diagrama abaixo classifica os elementos da última figura.



Veja que o número de elementos $> x$, isto é $\triangle$s, é no mínimo $\frac{3n}{10} - 6$.

Isto porque no mínimo $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$ grupos contribuem com 3 elementos $> x$, exceto possivelmente o último e aquele que contém x. Portanto, $3\left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2\right) \geq \frac{3n}{10} - 6$.

Terceira Solução - Complexidade

Da mesma forma, o número de elementos $< x$, isto é □s, é no mínimo $\frac{3n}{10} - 6$.
Assim, no passo 5 do algoritmo,

$$\max\{k-1, n-k\} \leq n - \left(\frac{3n}{10} - 6\right) \leq \frac{7n}{10} + 6.$$

A recorrência $T(n)$ está agora completa:

$$T(n) \leq \begin{cases} \Theta(1), & n \leq 140 \\ T(\lceil n/5 \rceil) + T(\lfloor 7n/10 \rfloor + 6) + \Theta(n), & n > 140, \end{cases}$$

140 é um "número mágico" que faz as contas funcionarem...

A solução é $T(n) \in \Theta(n)$

Solução da recorrência: $T(n) \leq cn$

$$
\begin{aligned}
T(n) &\leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 \rfloor + 6) + an \\
&\text{hi} \\
&\leq c\lceil n/5 \rceil + c(\lfloor 7n/10 \rfloor + 6) + an \\
&\leq c(n/5 + 1) + c(7n/10 + 6) + an \\
&= 9cn/10 + 7c + an \\
&= cn + (-cn/10 + 7c + an) \\
&\leq cn,
\end{aligned}
$$

Quero que $(-cn/10 + 7c + an) \leq 0$.

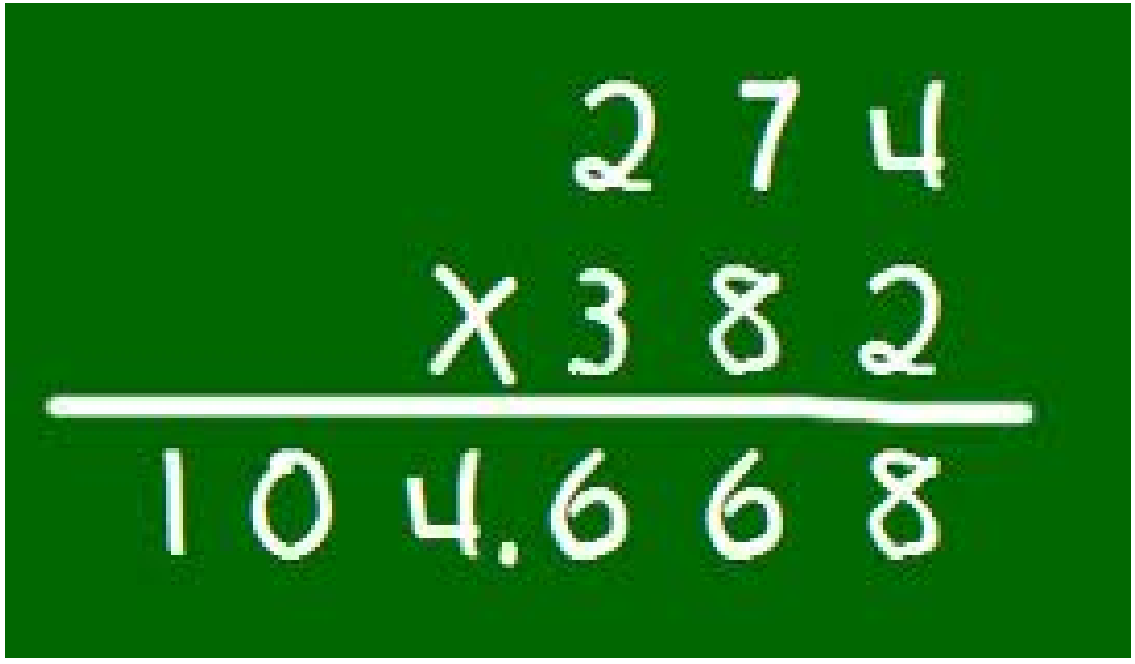
Isto equivale a $c \geq 10a(n/(n-70))$ quando $n > 70$. Como $n > 140$, temos $n/(n-70) \leq 2$ e assim basta escolher $c \geq 20a$.

Algoritmo SELECT

SELECT(A, p, r, i) ▷ Comentários detalhados neste LINK

1: **se** $p = r$ **então**
2: **devolve** $A[p]$
3: **senão**
4: $n \leftarrow \left\lceil \frac{r-p+1}{5} \right\rceil$ ▷ Consideramos um vetor B de tamanho n
5: **para** $j \leftarrow 1$ **até** n **faça**
6: $B[j] \leftarrow$ MEDIANA$(A,\ p+5(j-1),\ \min(p+5j-1; r))$
7: $x \leftarrow$ SELECT$(B, 1, n, \lceil \frac{n}{2} \rceil)$
8: $j \leftarrow$ busca(A, p, r, x)
9: $A[j] \leftrightarrow A[r]$
10: $q \leftarrow$ PARTITION(A, p, r)
11: $k \leftarrow q - p + 1$
12: **se** $i = k$ **então**
13: **devolve** x
14: **senão se** $i < k$ **então**
15: **devolve** SELECT$(A, p, q-1, i)$
16: **senão**
17: **devolve** SELECT$(A, q+1, r, i-k)$

Divisão e Conquista – Multiplicação de Inteiros



$$35871227428009 \times 11234908764388 \ = \ ?$$

Problema

Dados números inteiros positivos u e v com n dígitos cada
calcular o produto $u \times v$.

- exemplo: 99998888×77776666
- exemplo: 99998888×00076666
- imagine-se fazendo as contas com lápis e papel, dígito a dígito
- quanto tempo vai levar?
- tempo $\cong$ número de operações elementares
- *operação elementar* $=$ adição e multiplicação de dígitos
- n grande: aplicações à criptografia, fast Fourier transform, etc.

Se u e v têm n dígitos cada então

- $u \times v$ tem no máximo $2n$ dígitos
- $u + v$ tem no máximo $n + 1$ dígitos

Exemplos:

- $9999 \times 9999 = 9998\,0001$
- $9999 + 9999 = 1\,9998$

Algoritmo de multiplicação ordinário

Adição ordinária

$$\begin{array}{rl} 9\,9\,9\,9 & u \\ 7\,7\,7\,7 & v \\ \hline 1\,7\,7\,7\,6 & u+v \end{array}$$

- base do algoritmo: adição de dígitos
- n operações elementares

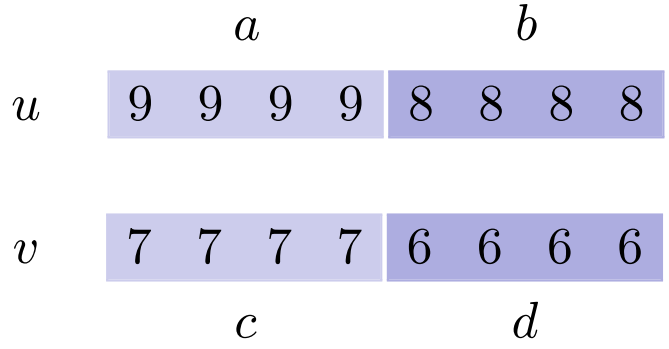
Multiplicação ordinária

$$
\begin{array}{rrrrrrrrl}
 & & & & 9 & 9 & 9 & 9 & \quad u \\
 & & & & 7 & 7 & 7 & 7 & \quad v \\
\hline
 & & & 6 & 9 & 9 & 9 & 3 & \\
 & & 6 & 9 & 9 & 9 & 3 & & \\
 & 6 & 9 & 9 & 9 & 3 & & & \\
6 & 9 & 9 & 9 & 3 & & & & \\
\hline
7 & 7 & 7 & 6 & 2 & 2 & 2 & 3 & \quad u \times v
\end{array}
$$

- base do algoritmo: multiplicação e adição de dígitos
- n^2 operações elementares (na verdade, $2n^2 + n$)
- n vezes mais lenta que adição
- n^2 é lento: $(2n)^2 = 4\,n^2$ e $(10n)^2 = 100\,n^2$

Desafio:

- inventar um algoritmo de multiplicação mais rápido
- há 50 anos acreditava-se que não existe algoritmo mais rápido
- acreditava-se que n^2 operações elementares são inevitáveis
- vou mostrar que $n^{1.6}$ operações elementares são suficientes



- $u = a \cdot 10^{n/2} + b$
- $v = c \cdot 10^{n/2} + d$
- $u \times v = (a \times c) \cdot 10^n + (a \times d + b \times c) \cdot 10^{n/2} + b \times d$
- estamos supondo n par

$$
\begin{array}{rl}
99998888 & u \\
77776666 & v \\
 & \\
9999 & a \\
\quad 8888 & b \\
7777 & c \\
\quad 6666 & d \\
 & \\
77762223 & ac \\
\quad 135775310 & ad + bc \\
\quad\quad 59247408 & bd \\
 & \\
7777580112347408 & x
\end{array}
$$

bom algoritmo para calculadora de bolso!

Número de operações elementares:

- $u \times v \;=\; (a \times c) \cdot 10^n \;+\; (a \times d + b \times c) \cdot 10^{n/2} \;+\; b \times d$
- 4 multiplicações de tamanho $n/2$ (mais 3 adições)
- operações elementares: $4\,(n/2)^2 = n^2$
- não ganhamos nada...

Tente, então, repetir o truque recursivamente

Divisão e conquista (n é potência de 2)

```
DIVIDA-E-CONQUISTE (u, v, n)
1  se n = 1
2      então devolva u × v
3      senão m ← n/2
4            a ← ⌊u/10^m⌋
5            b ← u mod 10^m
6            c ← ⌊v/10^m⌋
7            d ← v mod 10^m
8            ac ← DIVIDA-E-CONQUISTE (a, c, m)
9            bd ← DIVIDA-E-CONQUISTE (b, d, m)
10           ad ← DIVIDA-E-CONQUISTE (a, d, m)
11           bc ← DIVIDA-E-CONQUISTE (b, c, m)
12           x ← ac · 10^(2m) + (ad + bc) · 10^m + bd
13           devolva x
```

Número $T(n)$ de operações elementares:

- $T(1) = 1$ e $T(n) = 4\,T(n/2) + n$ para $n > 1$
- queremos uma "fórmula"
- solução da recorrência:

$$\begin{aligned}
T(n) &= 4\,T(n/2) + n \\
&= 4\left(4\,T(n/4) + n/2\right) + n \\
&= 16\,T(n/4) + 3n \\
&= 16\left(4\,T(n/8) + n/4\right) + 3n \\
&= 64\,T(n/8) + 7n \\
&= (2^j)^2\,T(n/2^j) + (2^j - 1)n \\
&= n^2\,T(n/n) + (n-1)n \\
&= 2n^2 - n
\end{aligned}$$

- $T(n) = 2n^2 - n$
- $T(n)$ é proporcional a n^2
- tão lento quanto o algoritmo ordinário...

Truque de Karatsuba

Divisão e conquista:

- $u = a \cdot 10^{n/2} + b \qquad v = c \cdot 10^{n/2} + d$
- $u \times v = (a \times c) \cdot 10^{n} + (a \times d + b \times c) \cdot 10^{n/2} + b \times d$
- 4 multiplicações de tamanho $n/2$ (e 3 adições)

Truque

- $y = (a + b) \times (c + d) = a \times c + a \times d + b \times c + b \times d$
- $u \times v = (a \times c) \cdot 10^{n} + (y - a \times c - b \times d) \cdot 10^{n/2} + b \times d$
- 3 multiplicações de tamanho $n/2$ (e 6 adições)

Estamos supondo n par

99998888	u
77776666	v
77762223	ac
59247408	bd
18887	$a+b$
14443	$c+d$
272784941	y
135775310	$y-ac-bd$
7777580112347408	x

Número de operações elementares:

- 3 multiplicações de tamanho $n/2$
- $3\,(n/2)^2 = 0.75\,n^2$ operações elementares
- ganhamos 25% com *uma* aplicação do truque
- para ganhar mais, repita o truque recursivamente

Algoritmo de Karatsuba (rascunho)

RASCUNHO-DE-KARATSUBA $(u,\ v,\ n)$

1 se $n = 1$
2 então devolva $u \times v$
3 senão $m \leftarrow n/2$
5 $a \leftarrow \lfloor u/10^m \rfloor$
6 $b \leftarrow u \bmod 10^m$
7 $c \leftarrow \lfloor v/10^m \rfloor$
8 $d \leftarrow v \bmod 10^m$
9 $ac \leftarrow$ RASCUNHO-DE-KARATSUBA $(a,\ c,\ m)$
10 $bd \leftarrow$ RASCUNHO-DE-KARATSUBA $(b,\ d,\ m)$
11 $y \leftarrow$ RASCUNHO-DE-KARATSUBA $(a+b,\ c+d,\ m+1)$
12 $x \leftarrow ac \cdot 10^{2m} + (y - ac - bd) \cdot 10^m + bd$
13 devolva x

Idéia básica correta, mas tem erros técnicos

Número de operações elementares:

- $T(n) = 3\,T(n/2) \;+\; n$ para $n = 2, 4, 8, 16, \ldots$ e $T(1) = 1$
- solução da recorrência: $T(n) = 3\,n^{\lg 3} - 2n$
- prova:

$$\begin{aligned} T(n) &= 3\,T(n/2) \;+\; n \\ &= 3\,(3\,(n/2)^{\lg 3} - 2(n/2)) \;+\; n \\ &= 3\,(n^{\lg 3} - n) \;+\; n \\ &= 3\,n^{\lg 3} \;-\; 2n \end{aligned}$$

- $T(2^j) = 3 \cdot 3^j - 2 \cdot 2^j$
- conclusão: $T(n)$ é proporcional a $n^{\lg 3}$

n	n^2	$n^{\lg 3}$	
8	64	27	42%
16	256	81	32%
32	1024	243	24%
64	4096	729	18%
128	16384	2187	13%
256	65536	6561	10%
512	262144	19683	8%
1024	1048576	59049	6%
2048	4194304	177147	4%
2^j	4^j	3^j	

- $\lg 3 \approx 1.584$
- $n^{\lg 3} \approx n\sqrt{n}$
- $n^{\lg 3}$ cresce mais devagar que n^2
- Karatsuba é mais rapido que o algoritmo ordinário (quando n é grande)

Algoritmo de Karatsuba: versão final, n arbitrário

KARATSUBA $(u,\ v,\ n)$

1 se $n \leq 3$
2 então devolva $u \times v$
3 senão $m \leftarrow \lceil n/2 \rceil$
4 $a \leftarrow \lfloor u/10^m \rfloor$
5 $b \leftarrow u \bmod 10^m$
6 $c \leftarrow \lfloor v/10^m \rfloor$
7 $d \leftarrow v \bmod 10^m$
8 $ac \leftarrow$ KARATSUBA $(a,\ c,\ m)$
9 $bd \leftarrow$ KARATSUBA $(b,\ d,\ m)$
10 $y \leftarrow$ KARATSUBA $(a+b,\ c+d,\ m+1)$
11 $x \leftarrow ac \cdot 10^{2m} + (y - ac - bd) \cdot 10^m + bd$
12 devolva x

Linha 10: $m + 1 < n$ pois $n > 3$

Testes de Liu, Huang, Lei:

	milissegundos		
n	ordinário	Karatsuba	
8	0.000	0.008	–
16	0.002	0.010	500%
32	0.007	0.019	271%
64	0.027	0.045	167%
128	0.101	0.128	127%
256	0.388	0.368	95%
512	1.541	1.111	72%
1024	6.079	3.400	56%
2048	24.460	12.000	49%

Quem é/foi Karatsuba?

- A.A. Karatsuba (1937–2008), matemático russo ▸ homepage
- Karatsuba descobriu o algoritmo quando tinha 23 anos
- o artigo foi publicado em 1962 sob os nomes de Karatsuba e Ofman

homepage:

- http://www.mi.ras.ru/~karatsuba/index.html

O algoritmo de Karatsuba é o mais rápido?

- existem algoritmos de multiplicação ainda mais rápidos
- algoritmo Toom-Cook: $n^{1.465}$
- algoritmo Schönhage-Strassen: $n \lg n \lg \lg n$

Conclusão

- nem sempre um algoritmo óbvio é o melhor
- algoritmos sofisticados podem ser mais rápidos (para n grande)
- matemática ajuda a projetar algoritmos mais sofisticados
- o poder do método de divisão-e-conquista
- o poder da recursão
- é útil saber resolver recorrências

Próximo passo: multiplicação rápida de matrizes

Referências

- Knuth, *The Art of Computer Programming*, 1997
- Brassard, and Bratley, *Algorithmics: Theory and Practice*, 1988
- Dasgupta, Papadimitriou, and Vazirani, *Algorithms*, 2006
- Kleinberg, and Tardos, *Algorithm Design*, 2005
- *Multiplication algorithm* ▸ Wikipedia
- *Karatsuba algorithm* ▸ Wikipedia

Wikipedia:

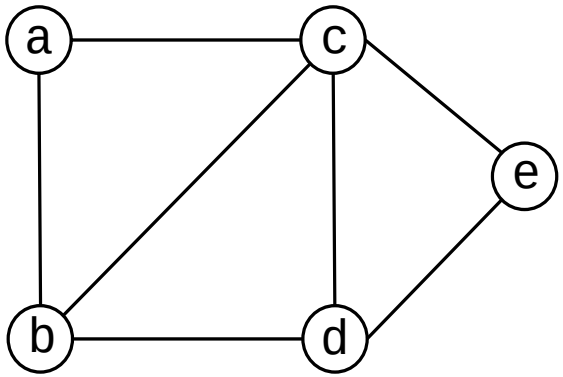
- https://en.wikipedia.org/wiki/Karatsuba_algorithm

Grafos: noções básicas e representação

Definição de Grafo

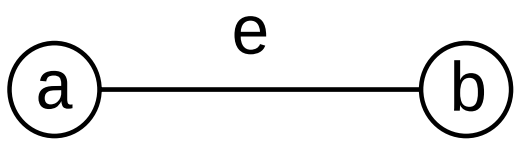
Um *grafo* é um par $G = (V, E)$ onde

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



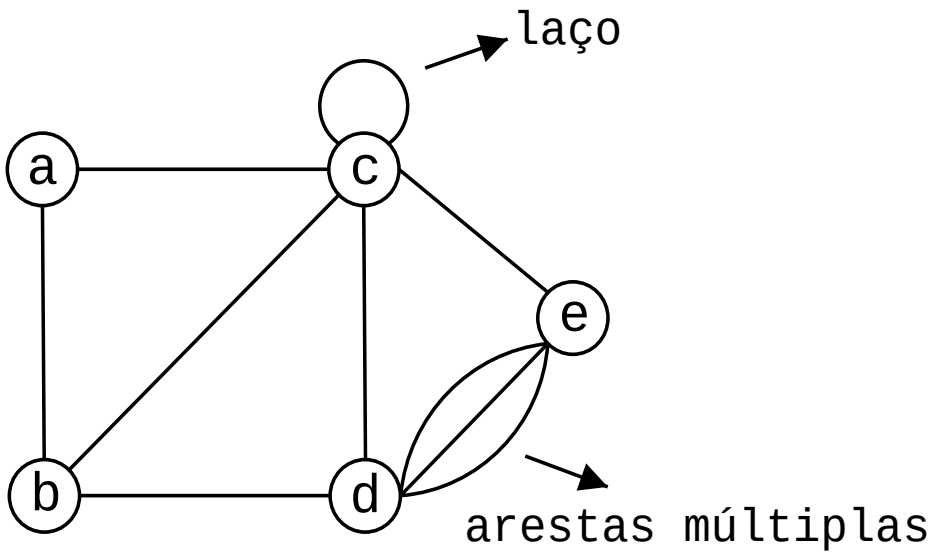
Definição de Grafo

- 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.



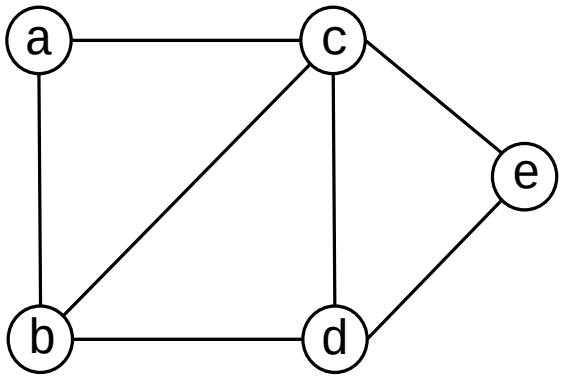
Grafo Simples

- 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

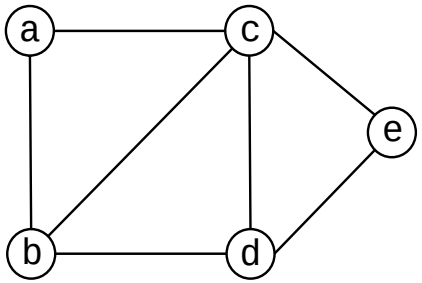
- Denotamos por $|V|$ e $|E|$ a cardinalidade dos conjuntos de vértices e arestas de um grafo G, respectivamente.
- No exemplo abaixo temos $|V| = 5$ e $|E| = 7$.



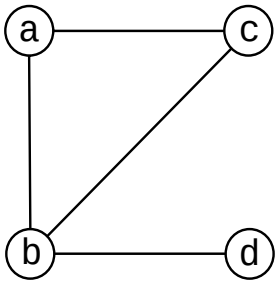
O *tamanho* do grafo G é dado por $|V| + |E|$.

Subgrafo e Subgrafo Gerador

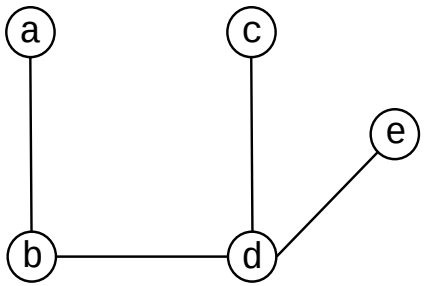
- Um *subgrafo* $H = (V', E')$ de um grafo $G = (V, E)$ é um grafo tal que $V' \subseteq V$, $E' \subseteq E$.
- Um *subgrafo gerador* de G é um subgrafo H com $V' = V$.
- **Exemplo:**



Grafo G



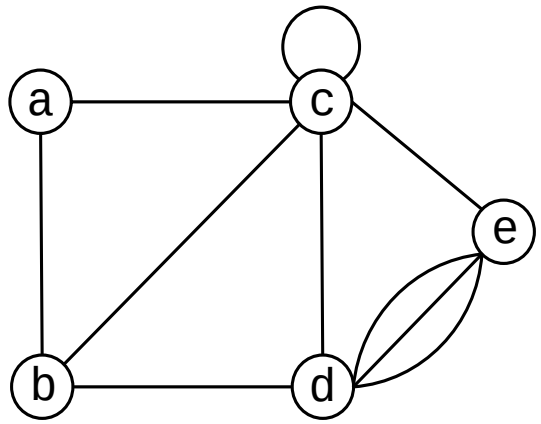
Subgrafo não gerador



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

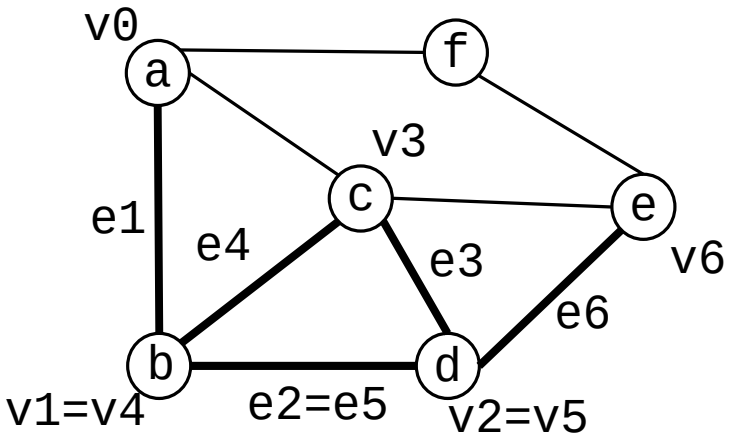
Teorema (*Handshaking lemma*)

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

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

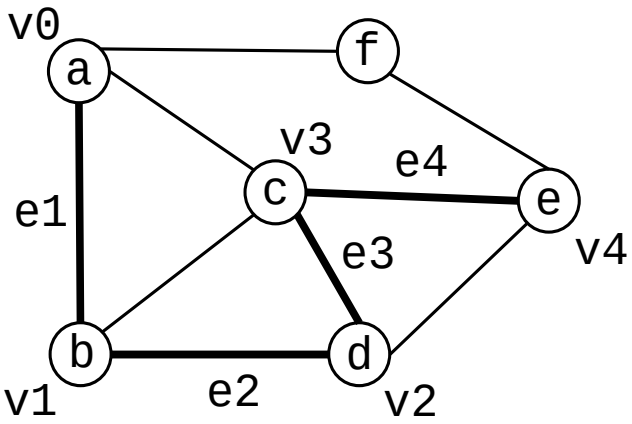
Caminhos em Grafos

- 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:**

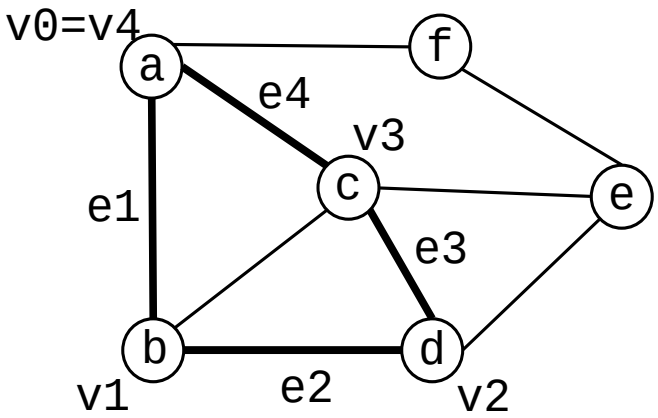


Caminhos Simples e Ciclos

- 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 uma caminho em que $v_0 = v_n$.
- Um ciclo é dito ser *simples* se $v_0, \ldots, v_{n-1}$ são distintos.
- Um grafo que não possui ciclos é dito ser *acíclico*.
- **Exemplo:**



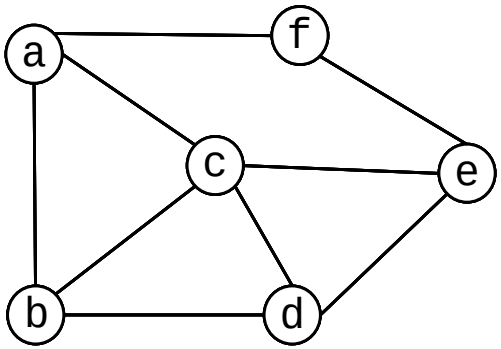
Caminho Simples



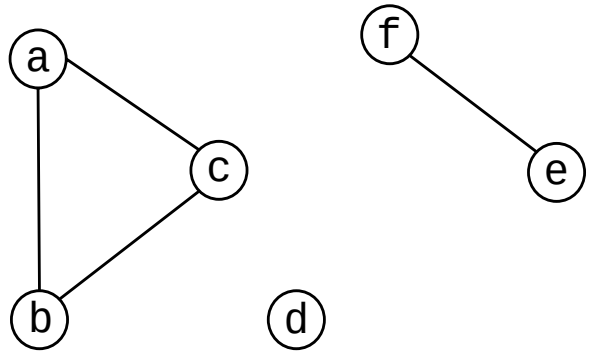
Ciclo

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.
- Um *componente conexo* de um grafo é um subgrafo maximal conexo. Algumas vezes chamaremos de componente conexo apenas o conjunto de seus vértices.
- **Exemplo:**



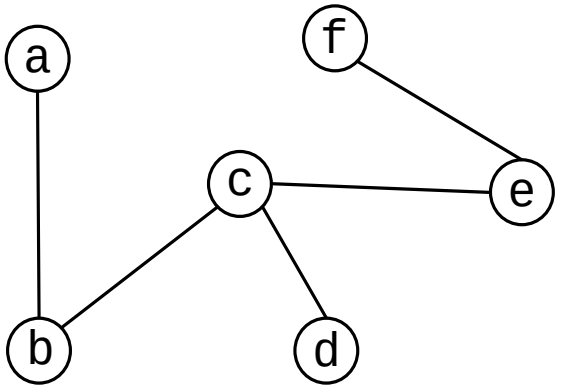
Conexo



Não-conexo com
3 componentes conexos

Árvore

- Um grafo G é uma *árvore* se é conexo e 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:**

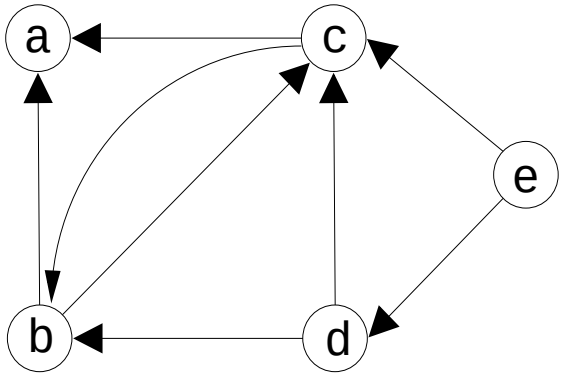


Alguns exemplos de grafos

- 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

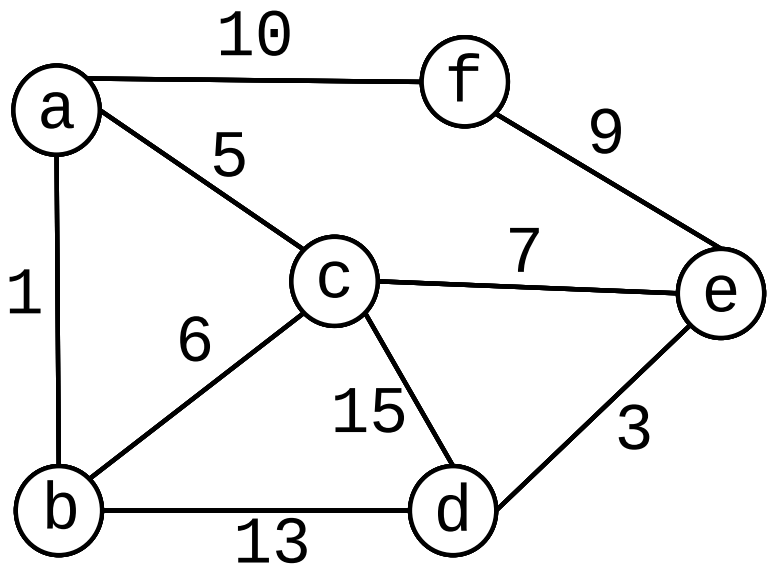
- Se $e = (u, v)$ é uma aresta de um grafo orientado G, então dizemos que e sai de u e entra em v.
- O *grau de saída* $d^+(v)$ de um vértice v é o número de arestas que saem de v. O *grau de entrada* $d^-(v)$ de v é o número de arestas que entram em v.
  Teorema. *Para todo grafo orientado* $G = (V,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

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

- Em geral, considera-se que $P = (v_0, e_1, v_1, \ldots, e_n, v_n)$ é um caminho (ciclo) em um grafo orientado se $e_i = (v_{i-1}, v_i)$.
- Há um conceito de conexidade para grafos orientados que veremos mais tarde.

Grafo Ponderado

- Um grafo (orientado ou não) é *ponderado* se 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 pode ser ocupada por 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 passeio 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 passeio 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

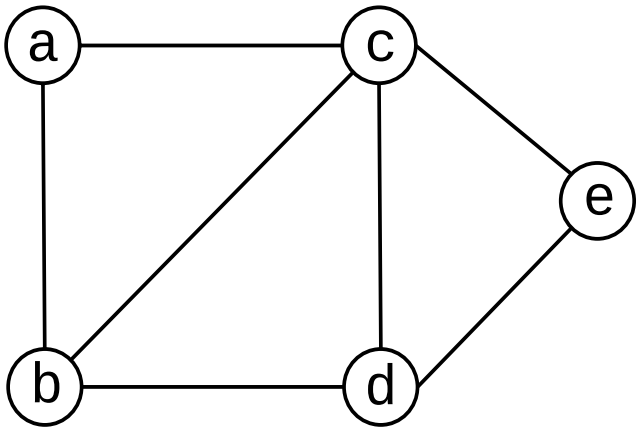
- Seja $G = (V, E)$ um grafo simples (orientado ou não).
- A *matriz de adjacência* de G é uma matriz quadrada A de ordem $|V|$, cujas linhas e colunas são indexadas pelos vértices em V, e tal que:

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

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

Matriz de adjacência

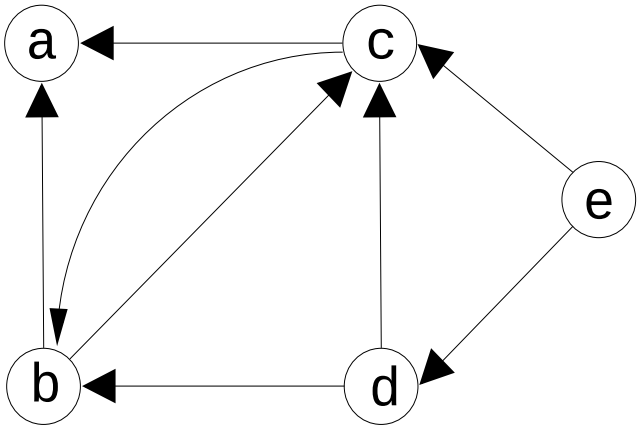
- Exemplo de um grafo e a matriz de adjacência correspondente.



	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

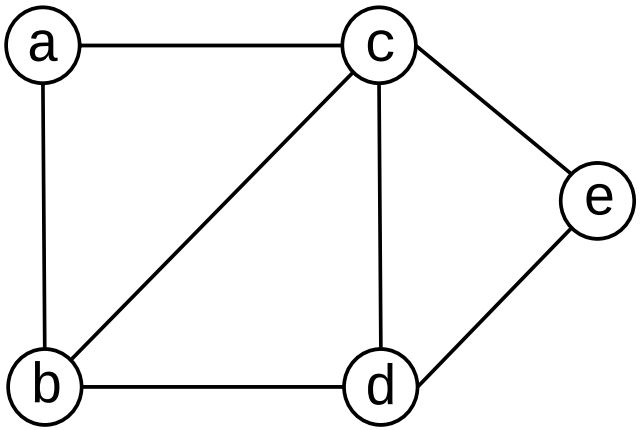
Listas de adjacência

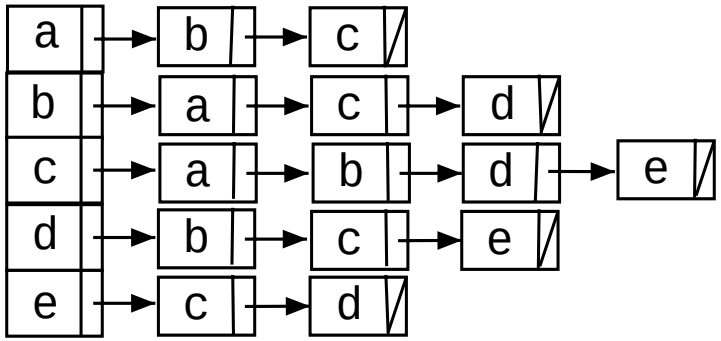
- Seja $G = (V, E)$ um grafo simples (orientado ou não).
- A representação de G por uma *lista de adjacências* consiste no seguinte.

  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

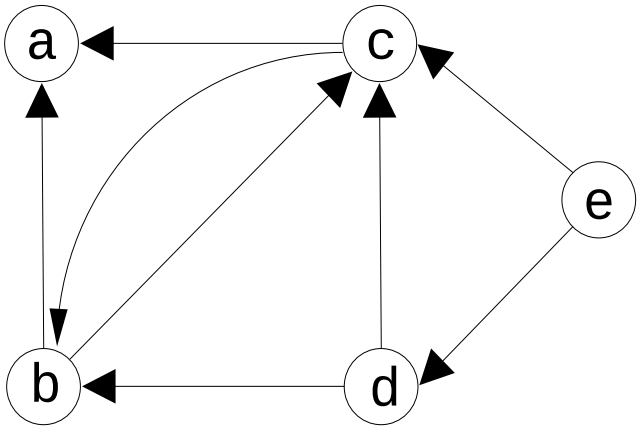
- Exemplo de um grafo não-orientado e a listas de adjacência correspondente.

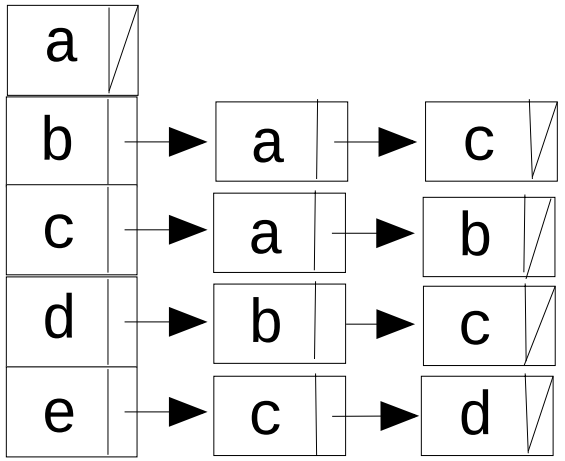




Lista de adjacências

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)
  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 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.

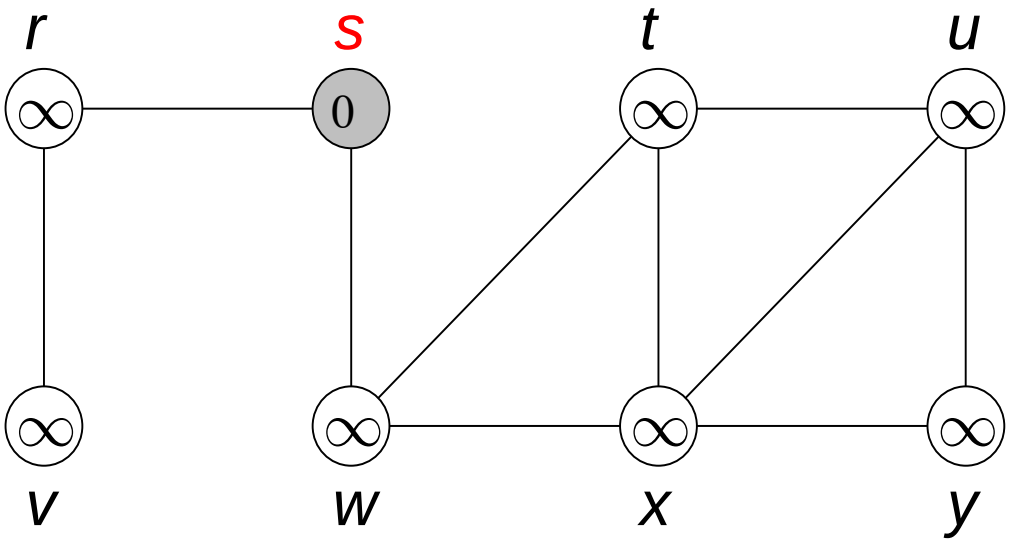
Busca em largura

- 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”.

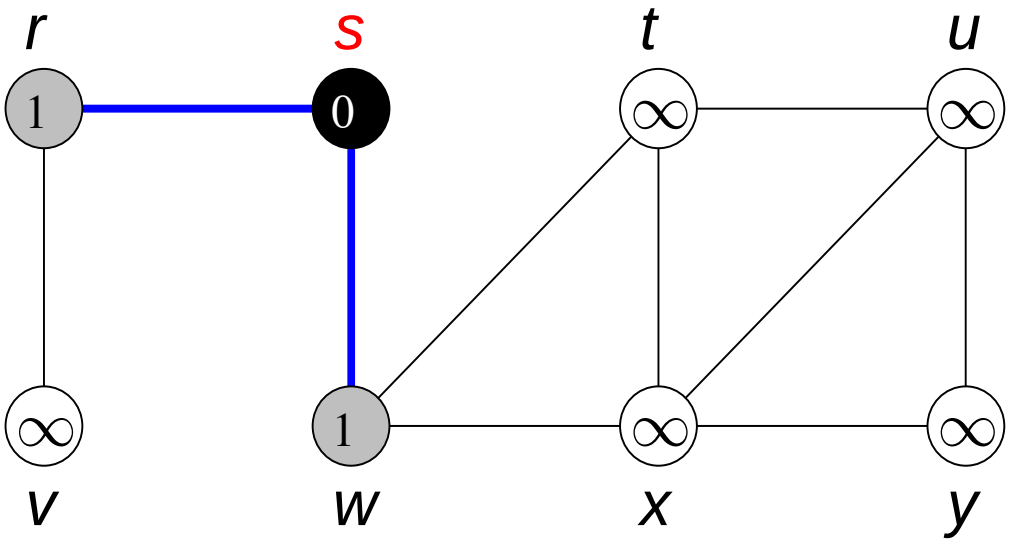
Exemplo (CLRS)



Q s

0

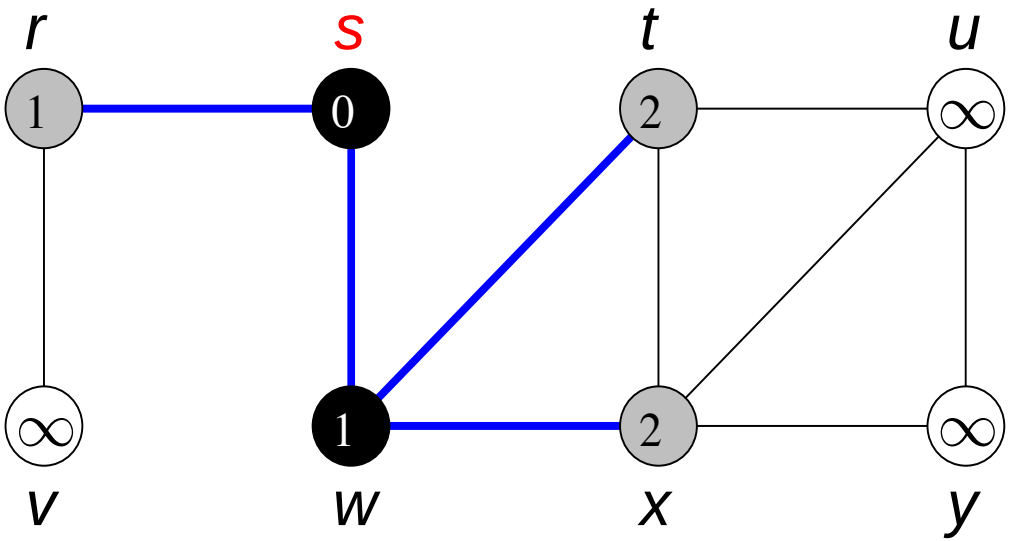
Exemplo (CLRS)

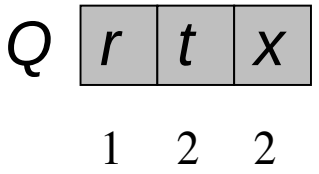


Q | w | r

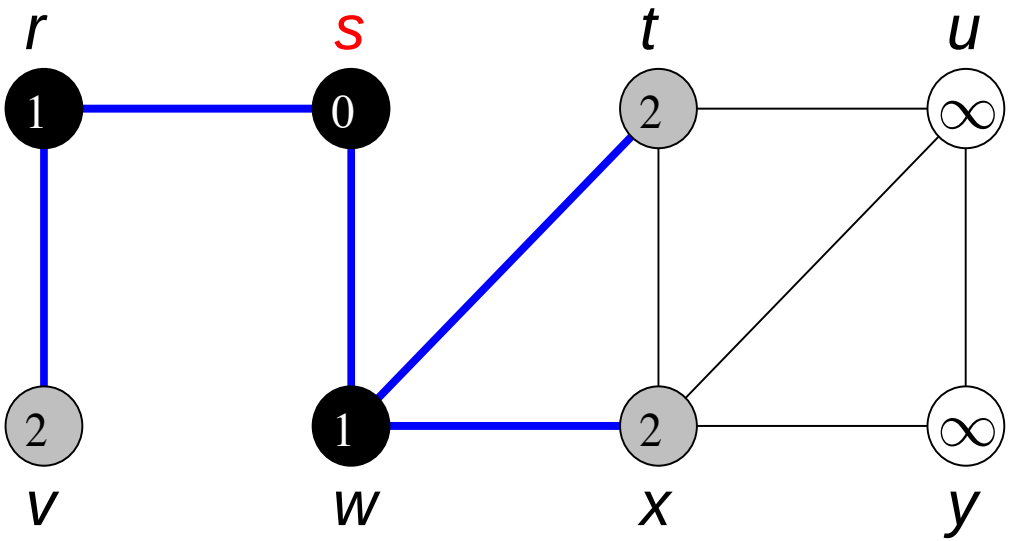
1 1

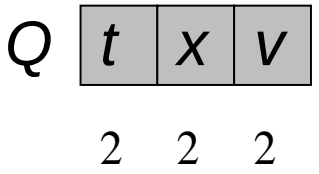
Exemplo (CLRS)



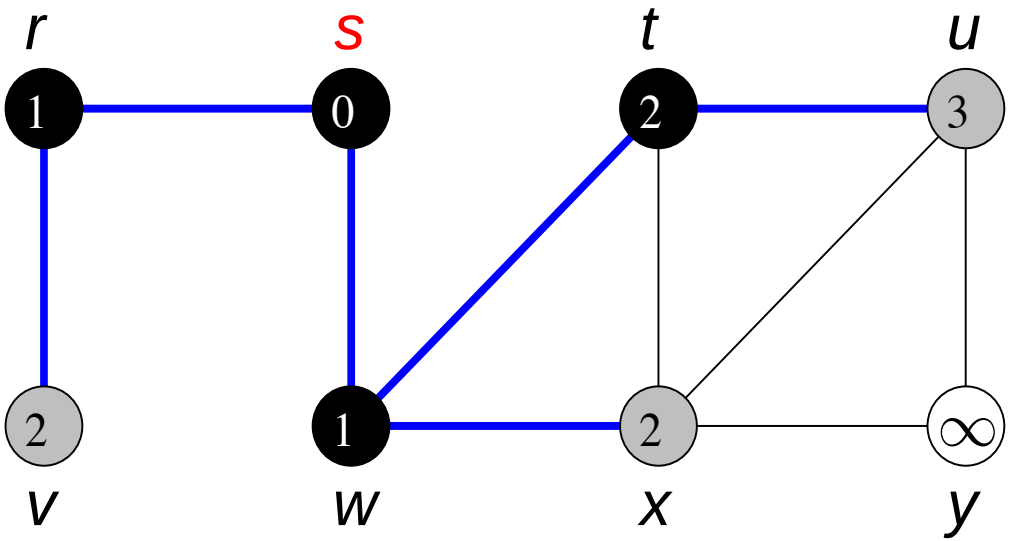


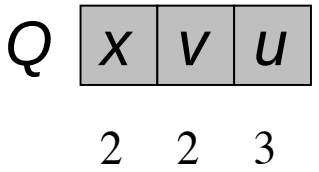
Exemplo (CLRS)



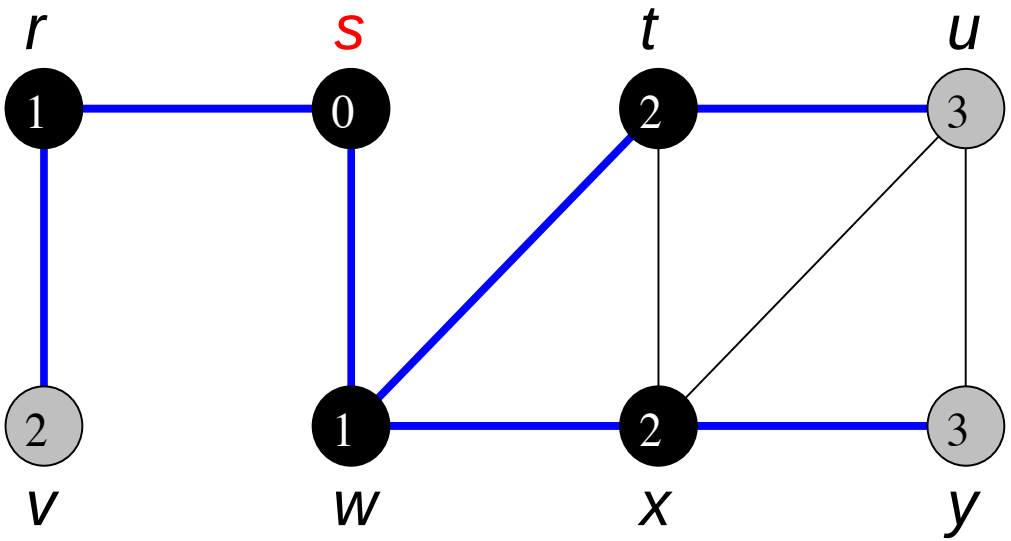


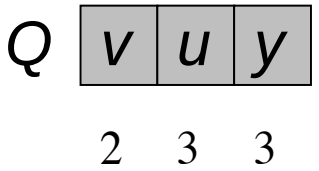
Exemplo (CLRS)



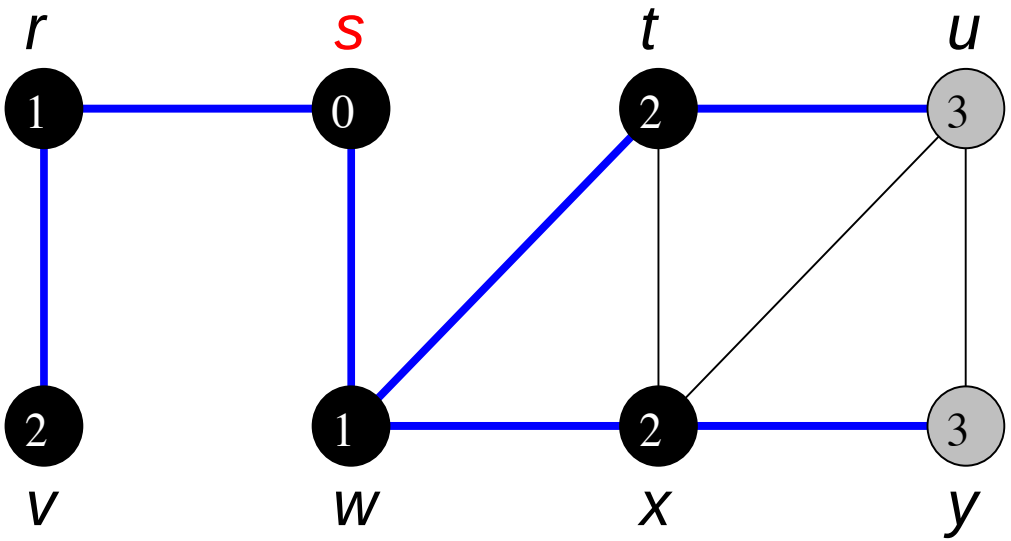


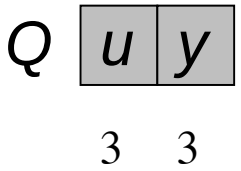
Exemplo (CLRS)



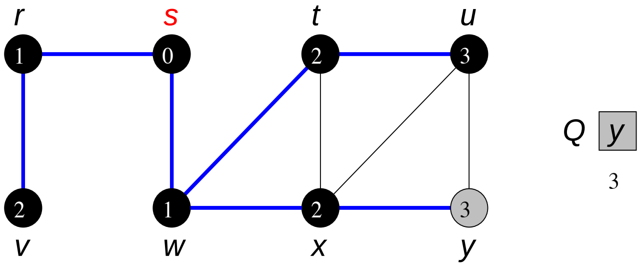


Exemplo (CLRS)

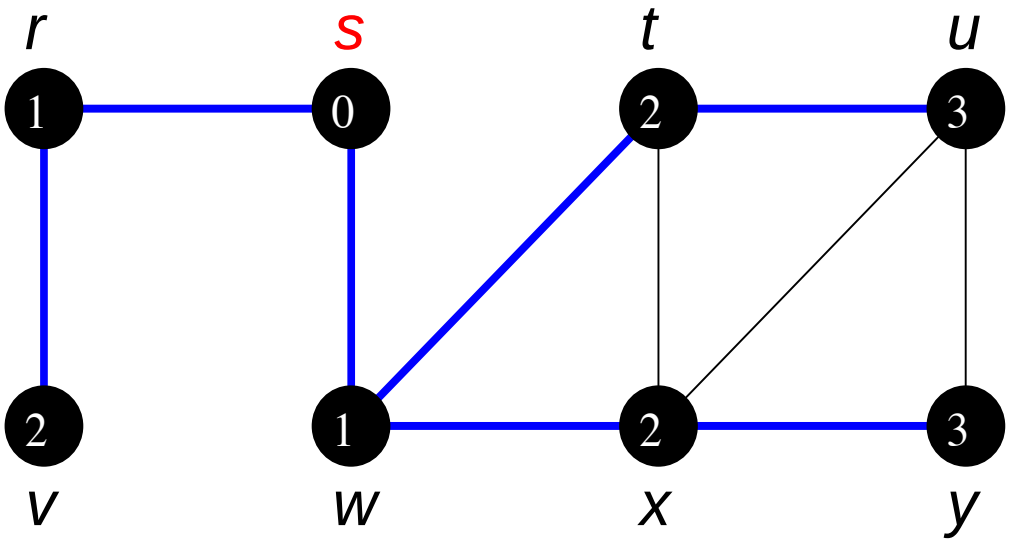




Exemplo (CLRS)



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 ∈ V[G] − {s} faça
 2        cor[u] ← branco
 3        d[u] ← ∞
 4        π[u] ← NIL
 5  cor[s] ← cinza
 6  d[s] ← 0
 7  π[s] ← 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 $O(V + E)$

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.
```

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 $2|V|$.

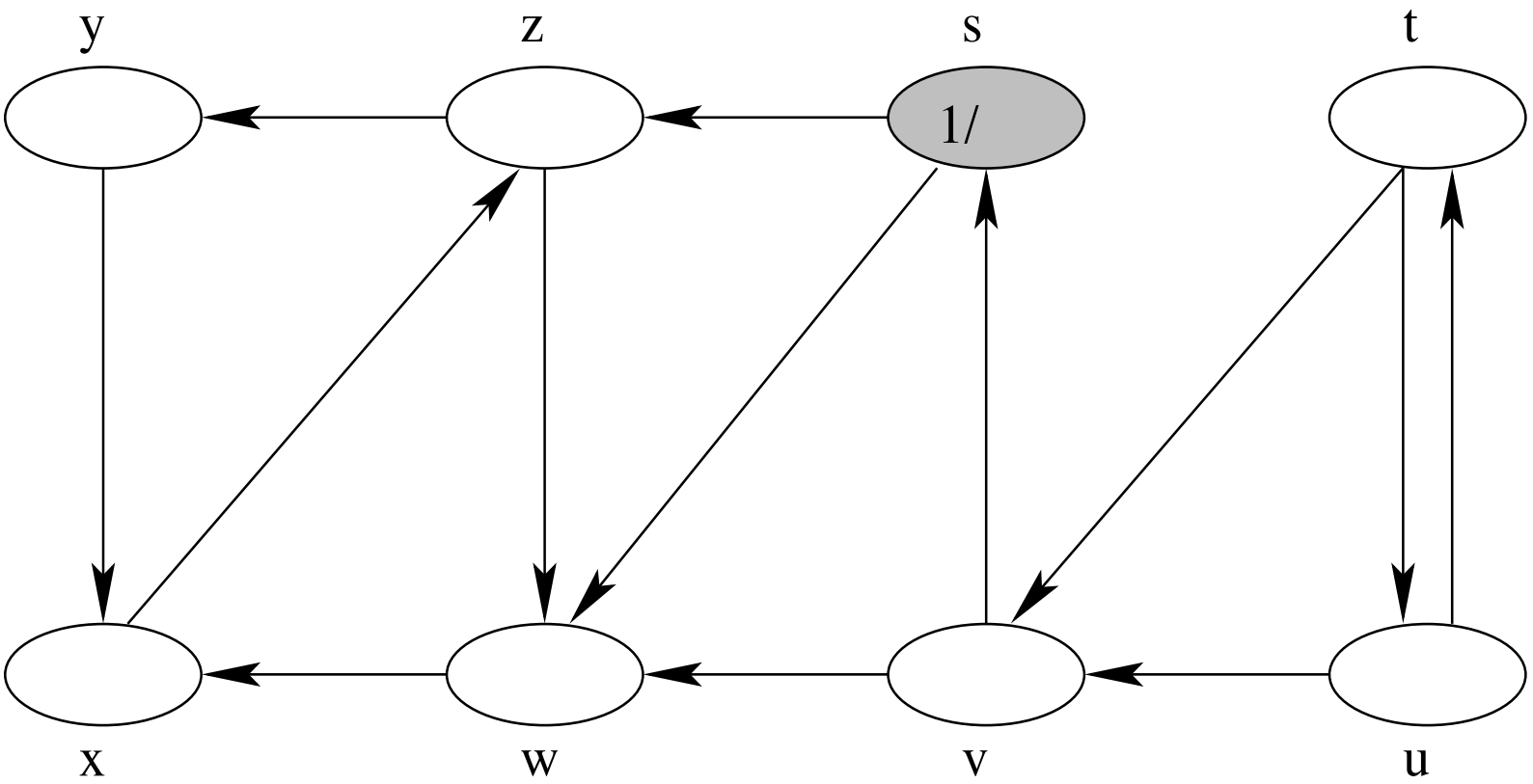
Classificação de arestas

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

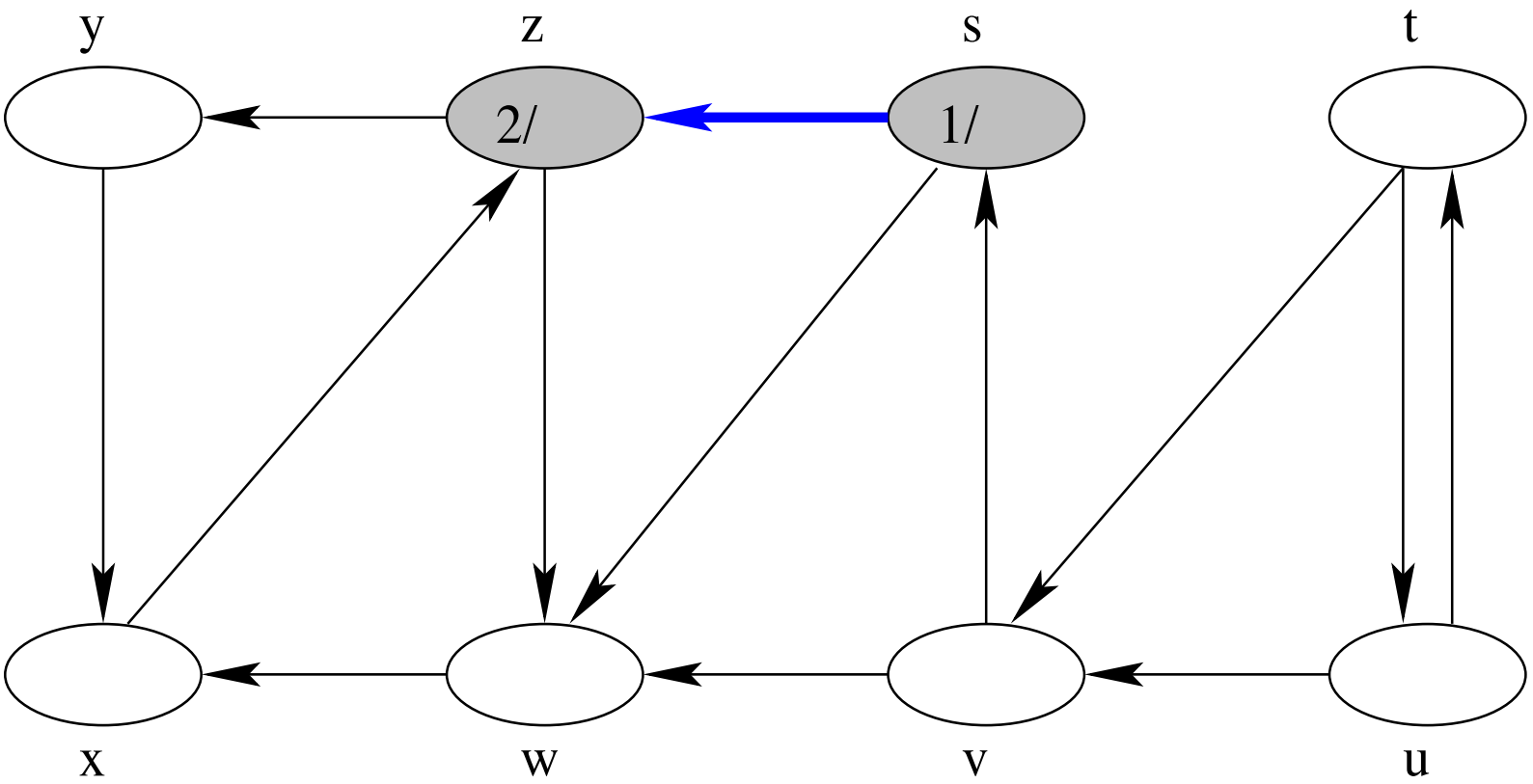
Ela classifica as arestas em quatro tipos:

- 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.

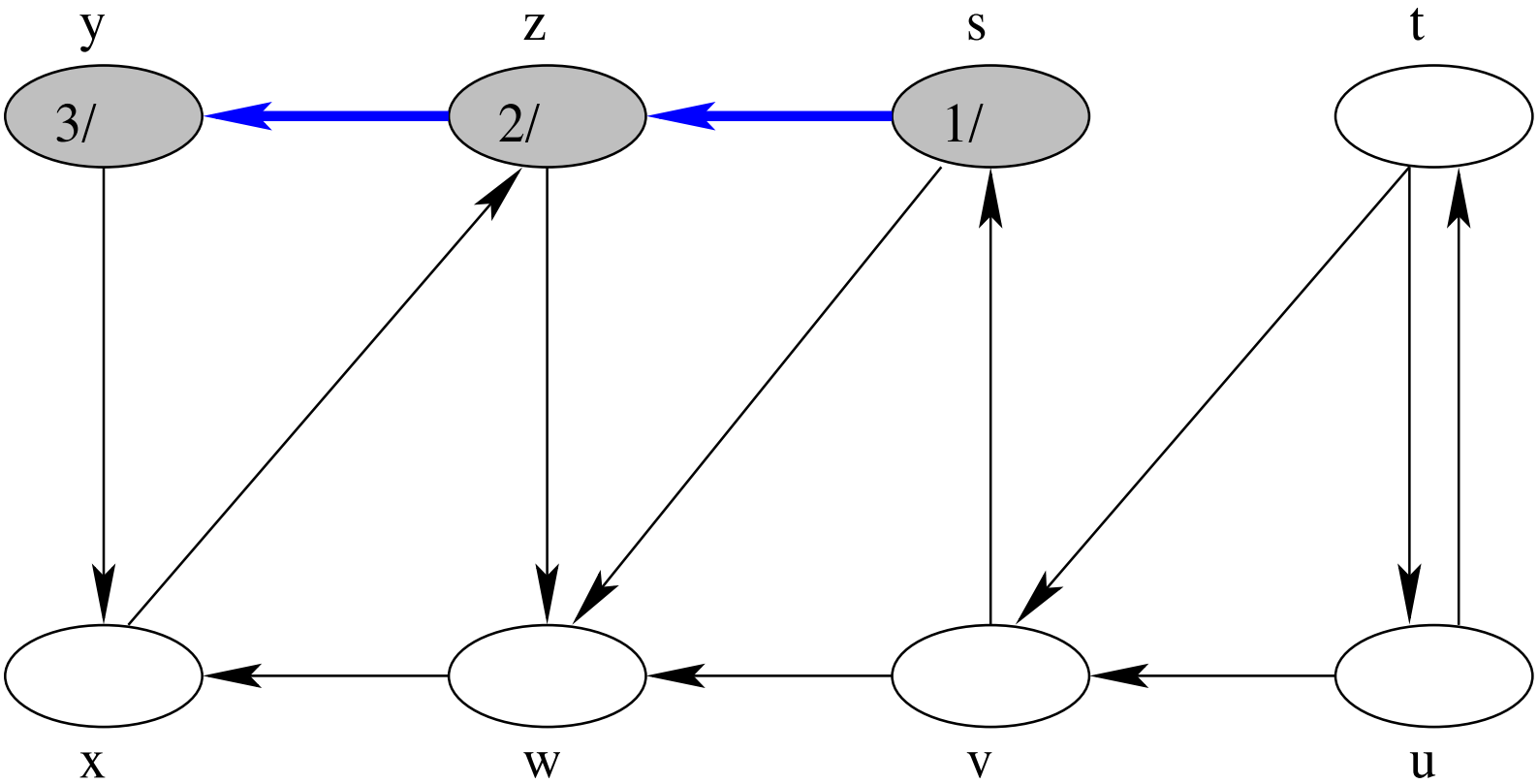
Exemplo



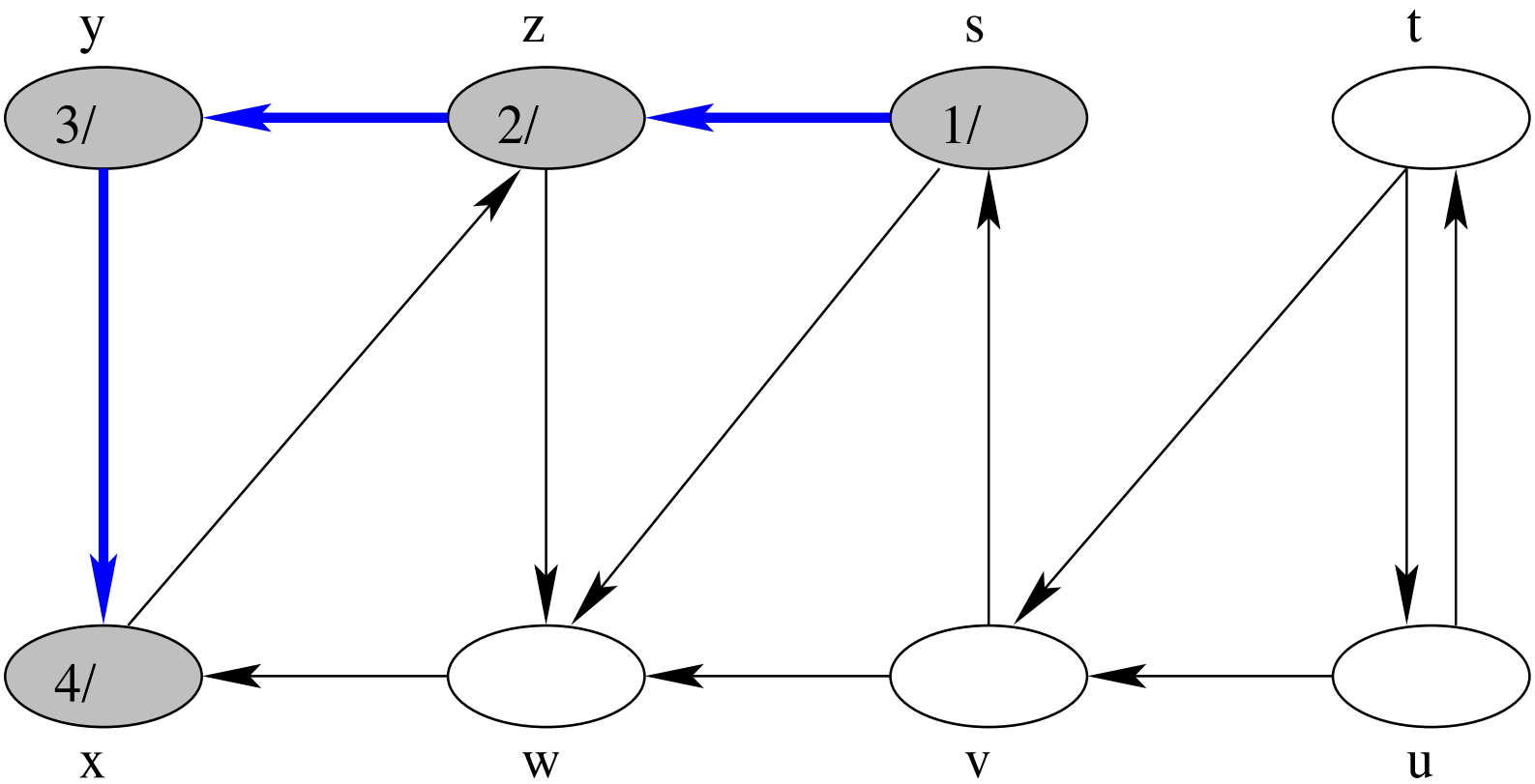
Exemplo



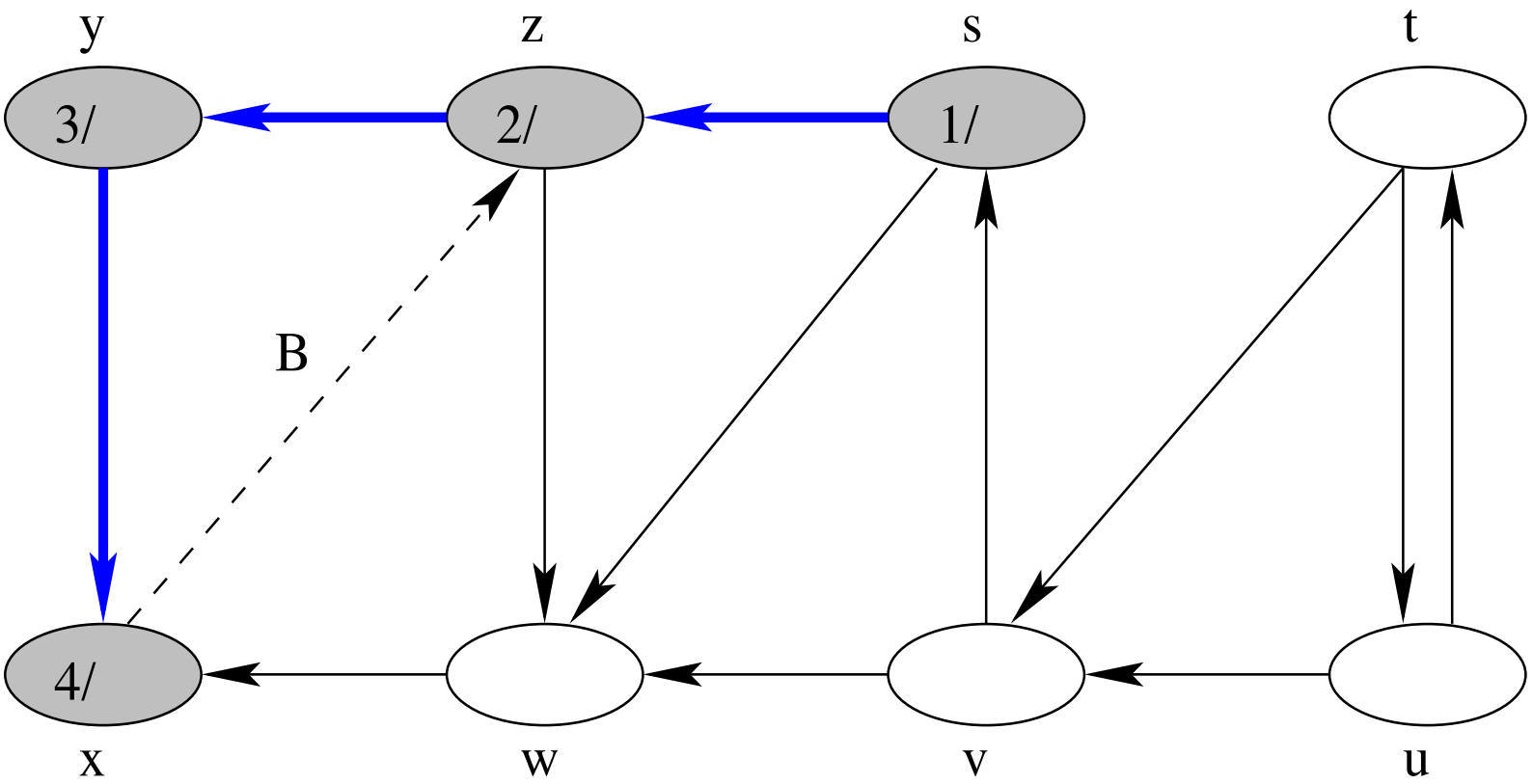
Exemplo



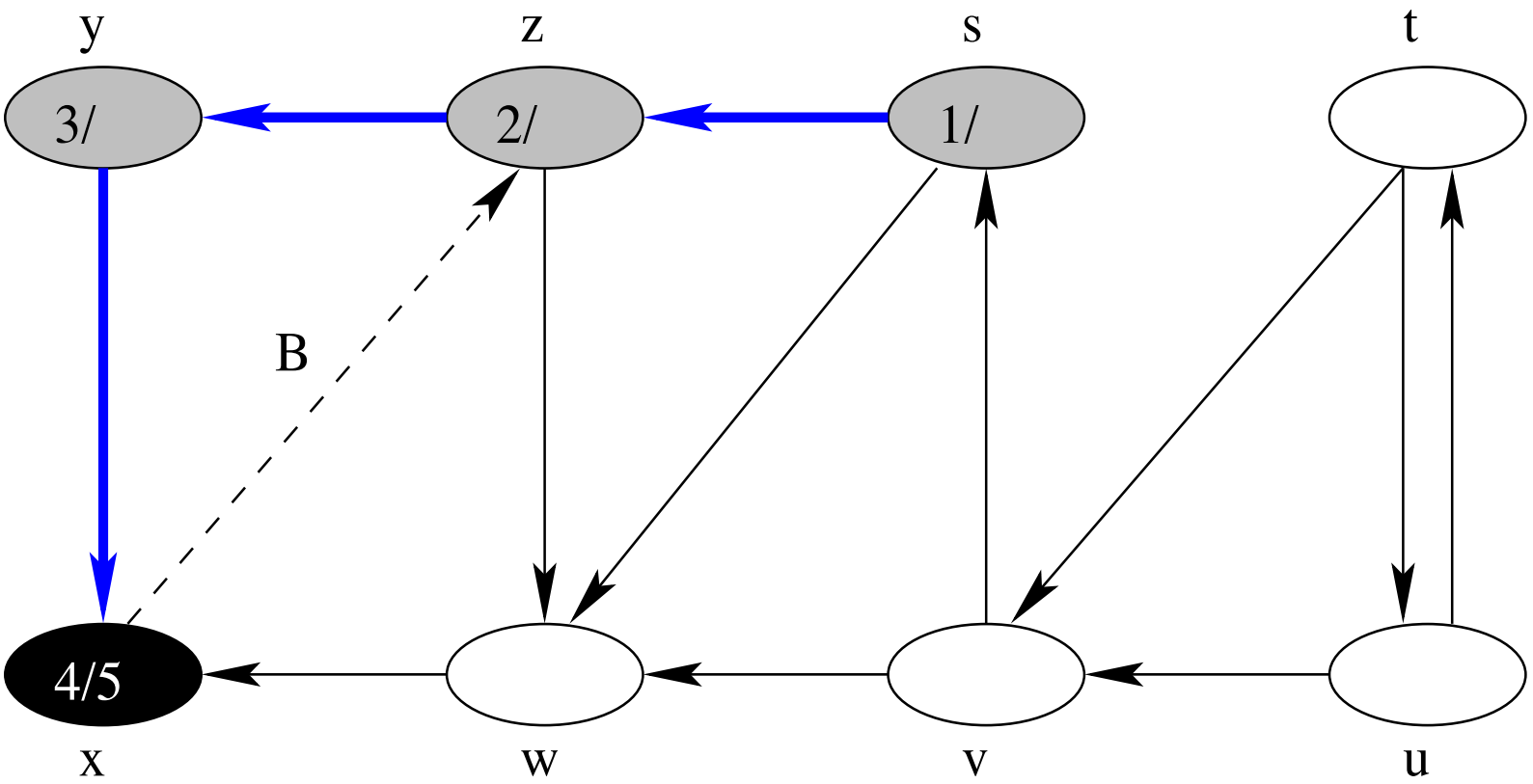
Exemplo



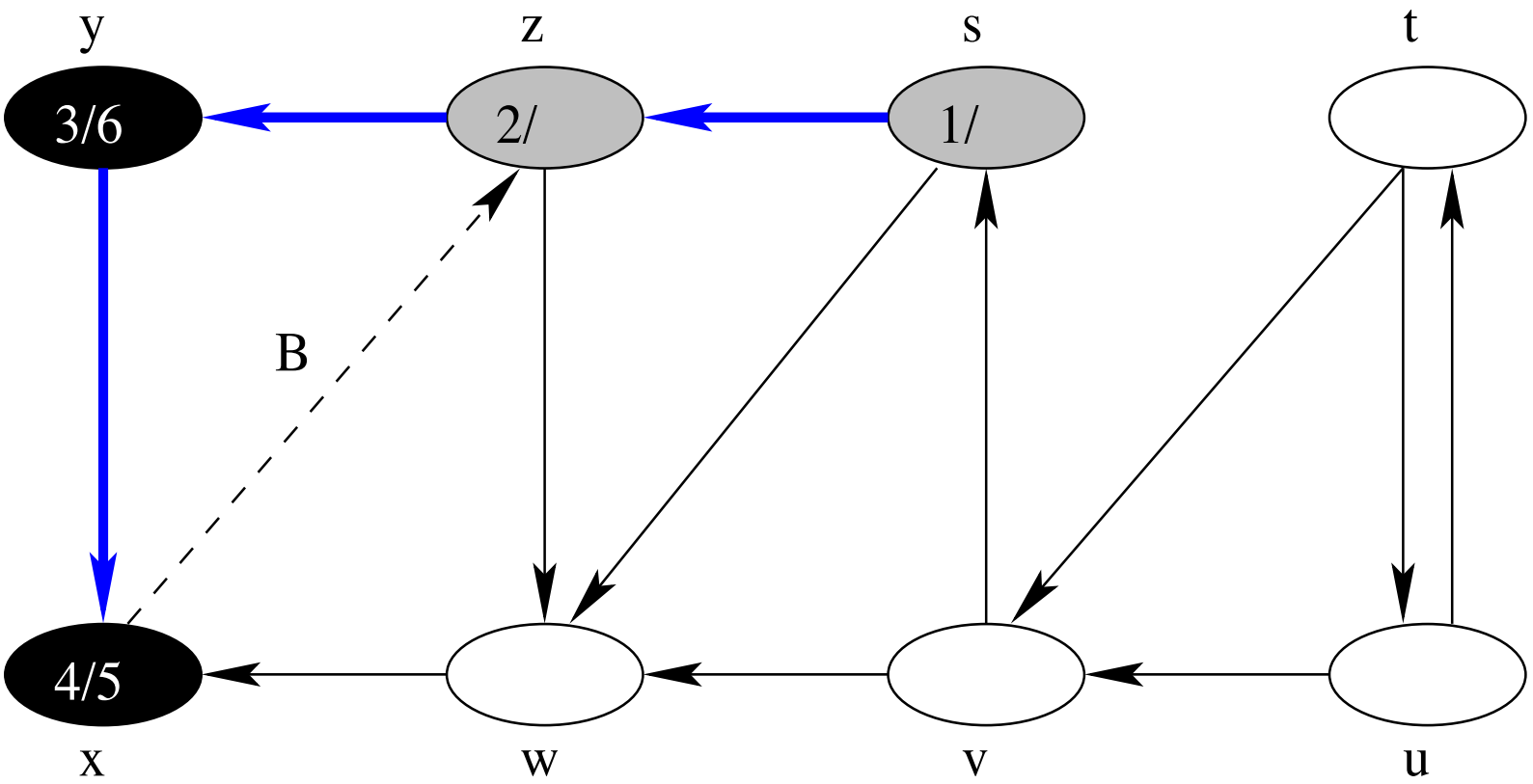
Exemplo



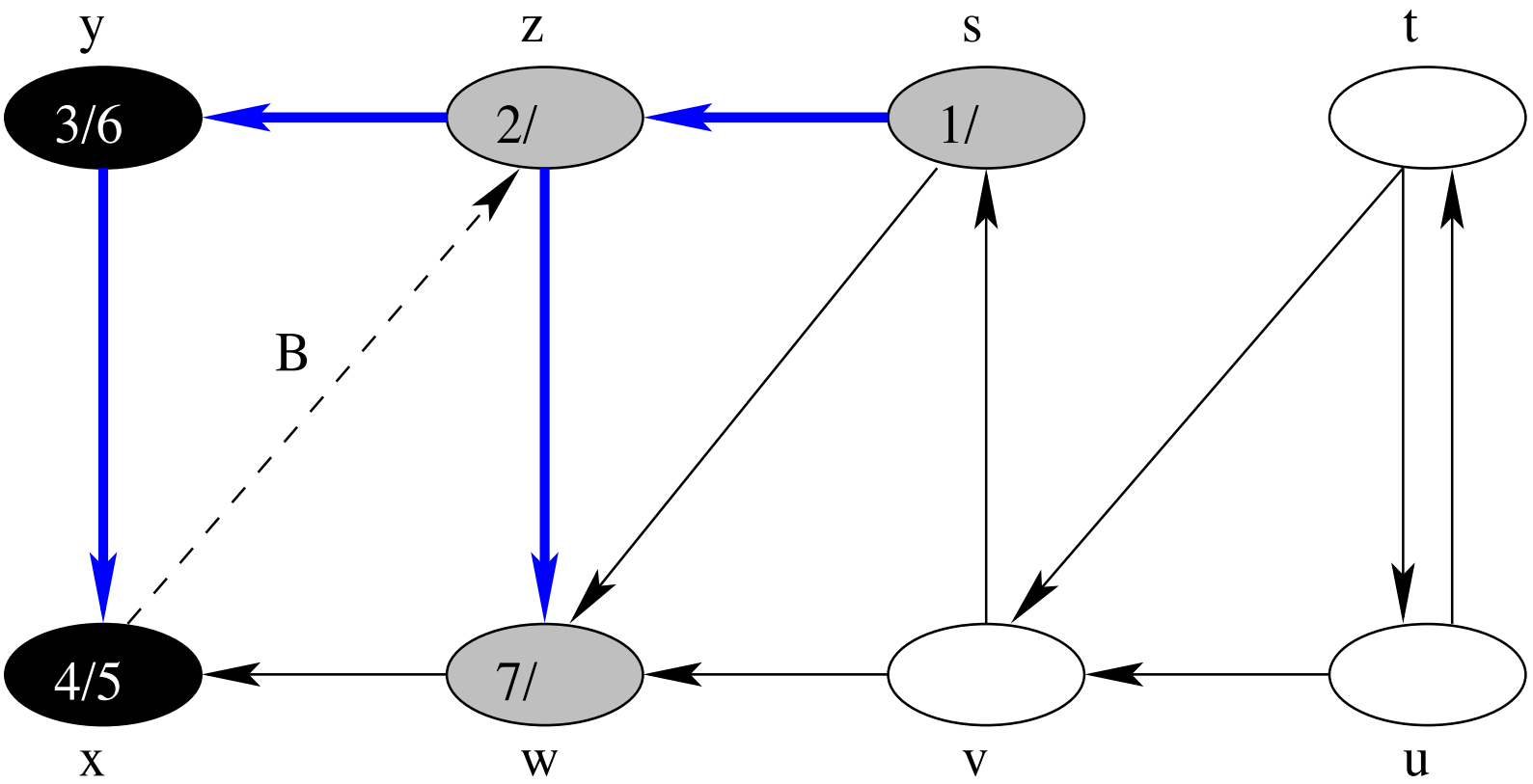
Exemplo



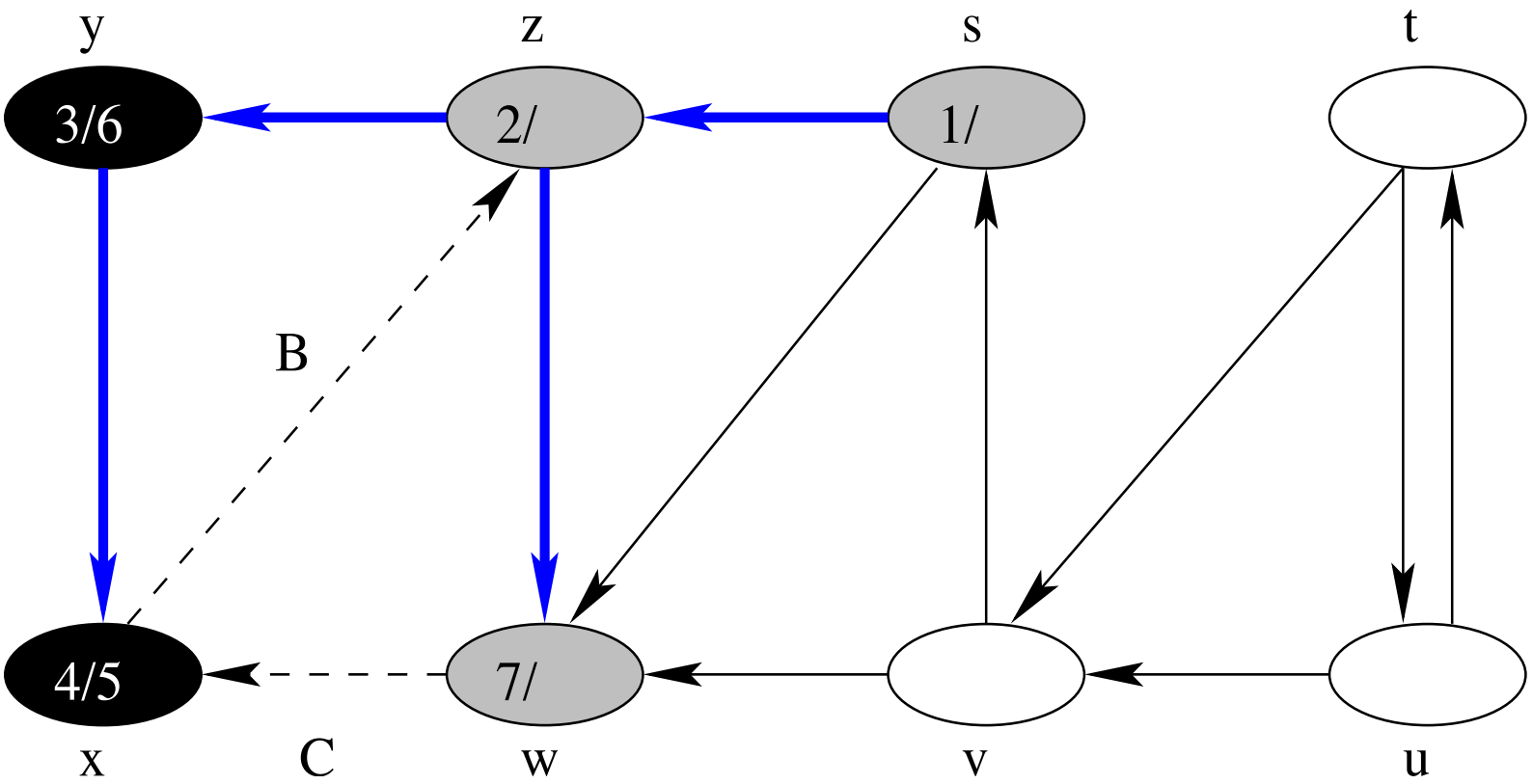
Exemplo



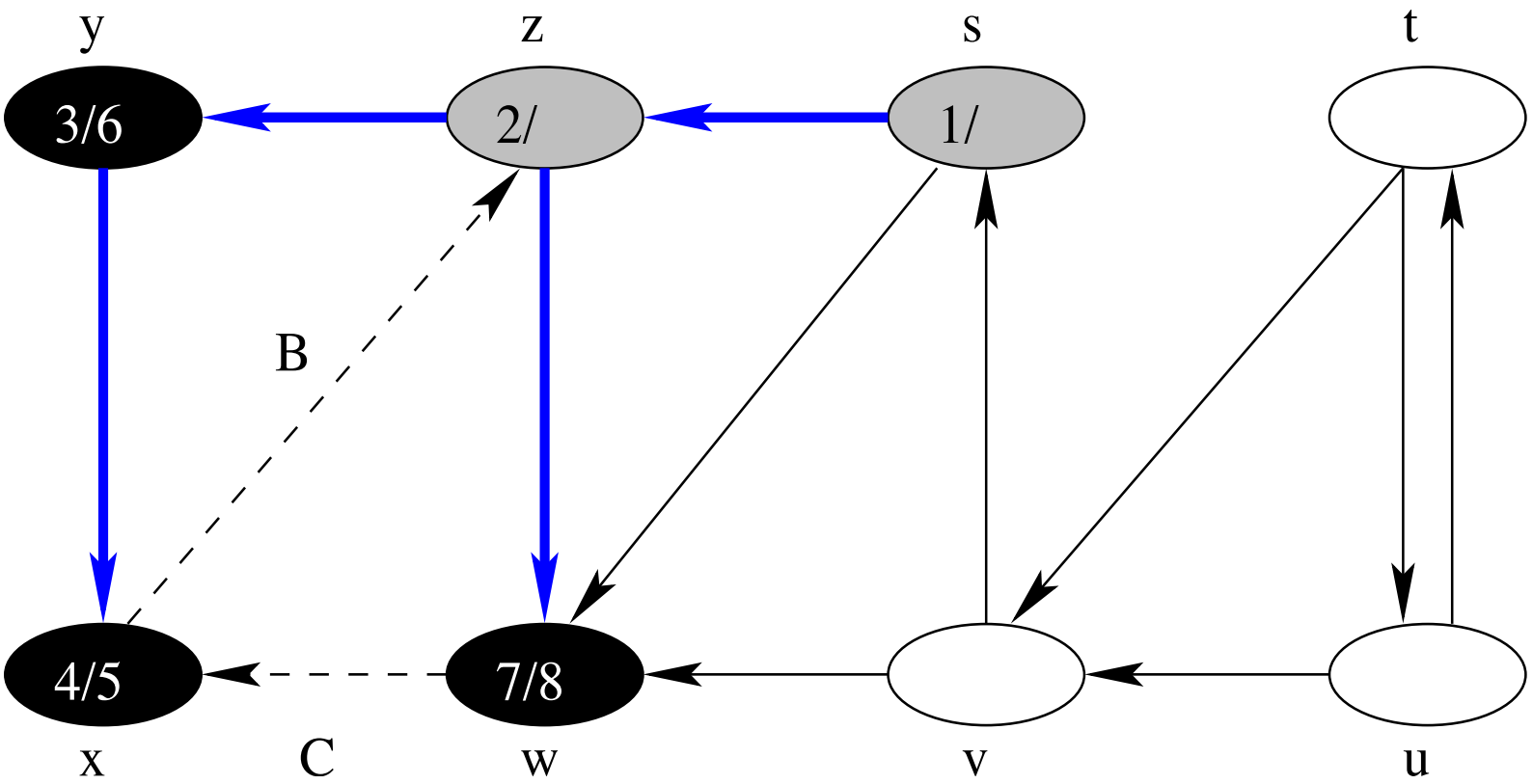
Exemplo



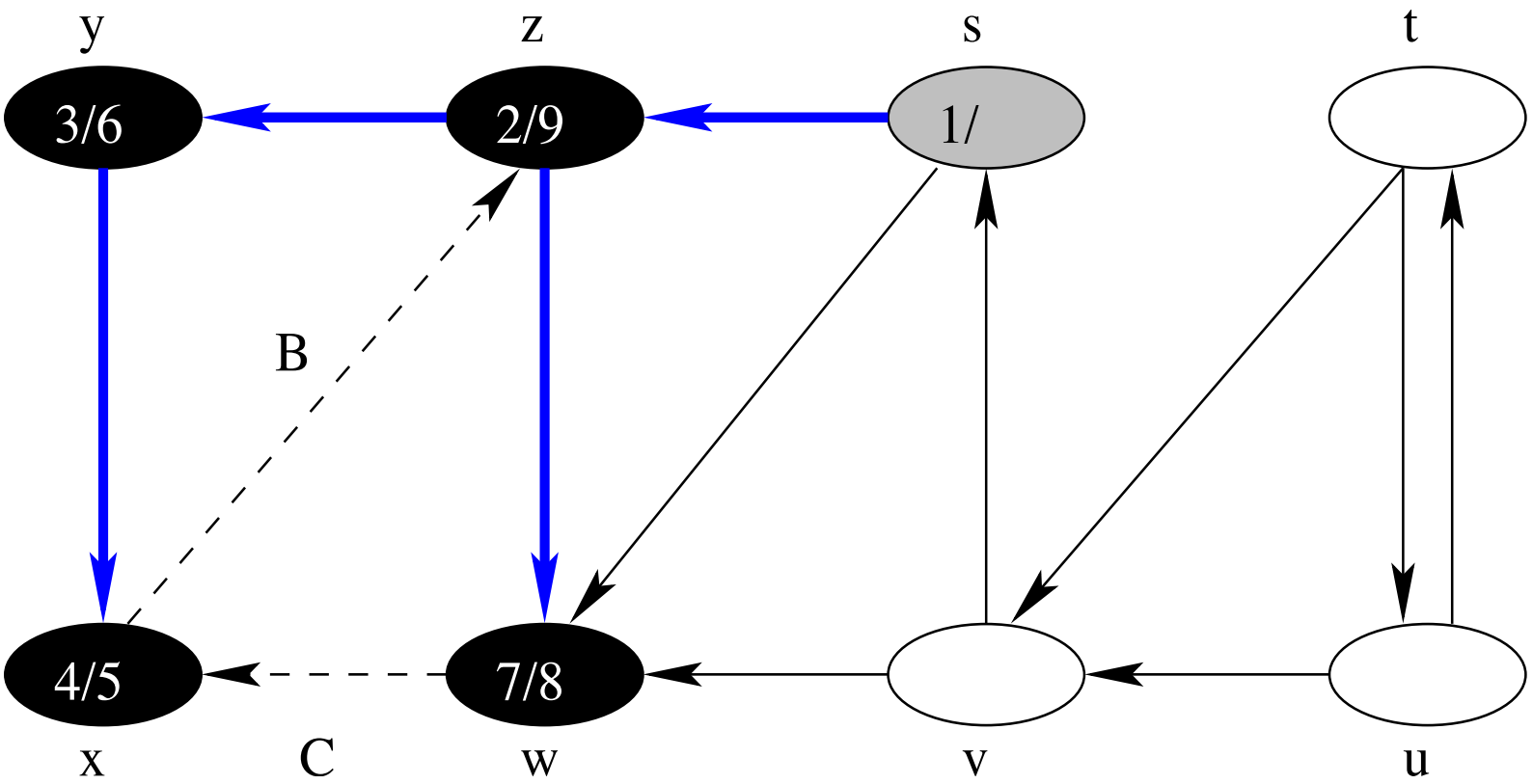
Exemplo



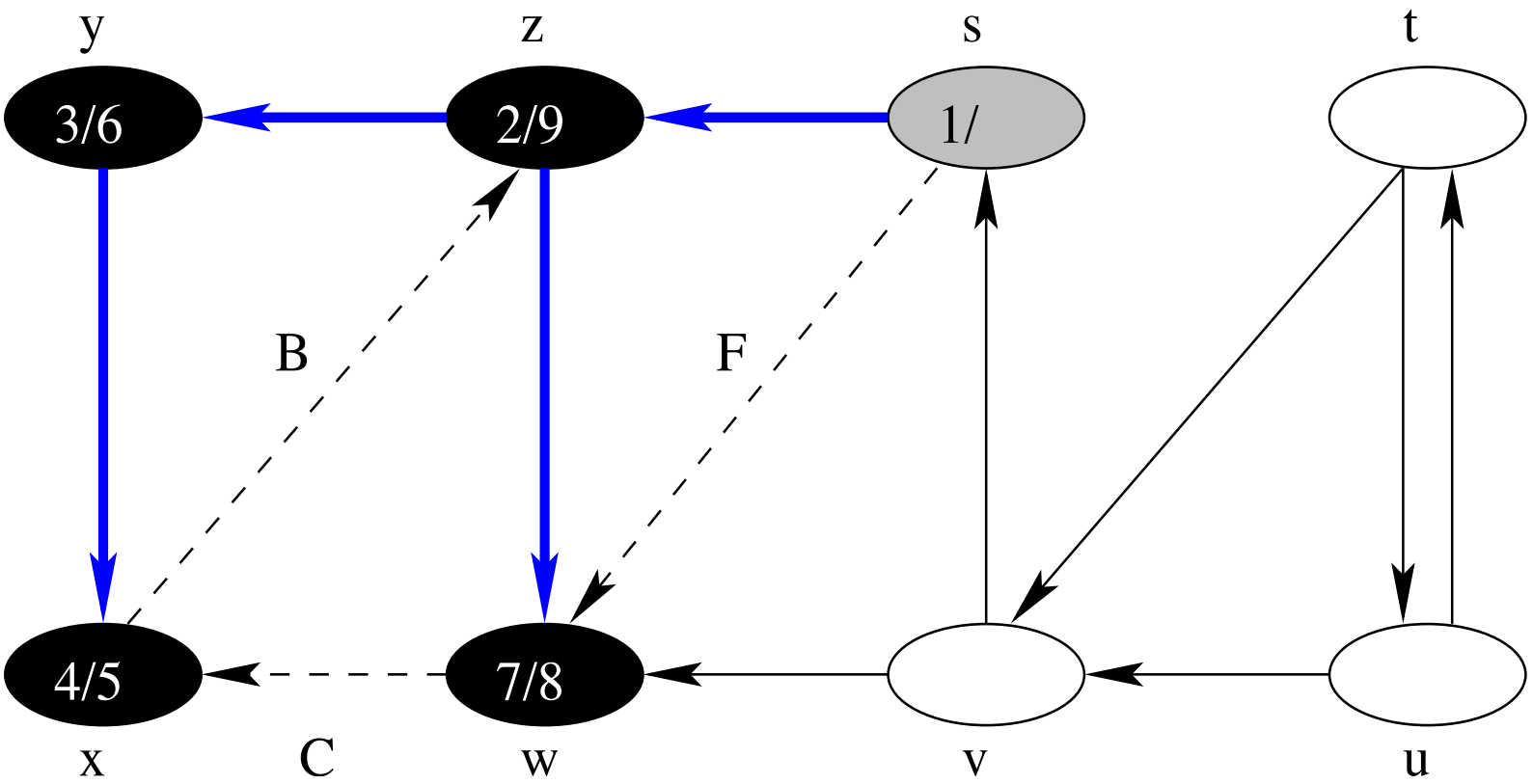
Exemplo



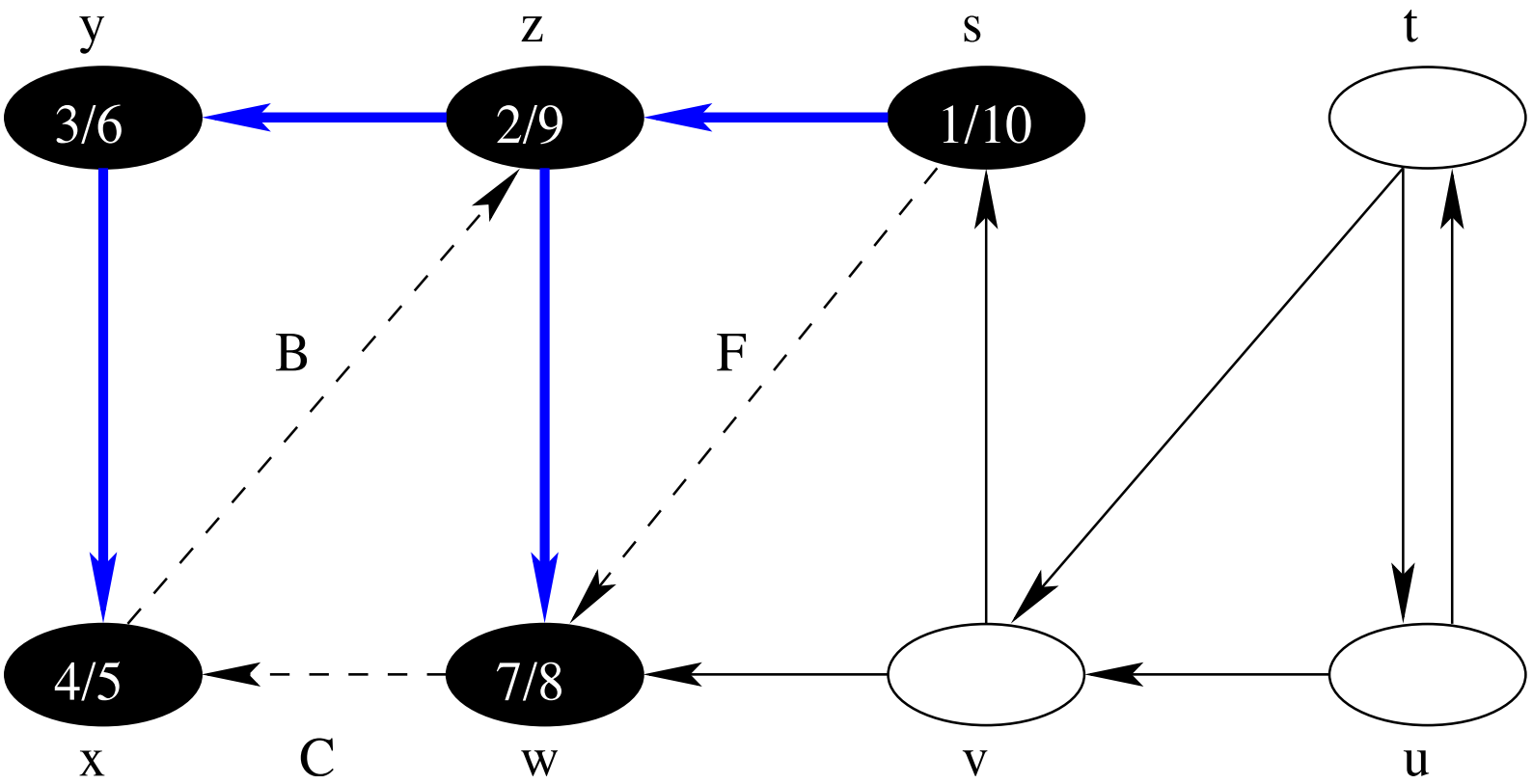
Exemplo



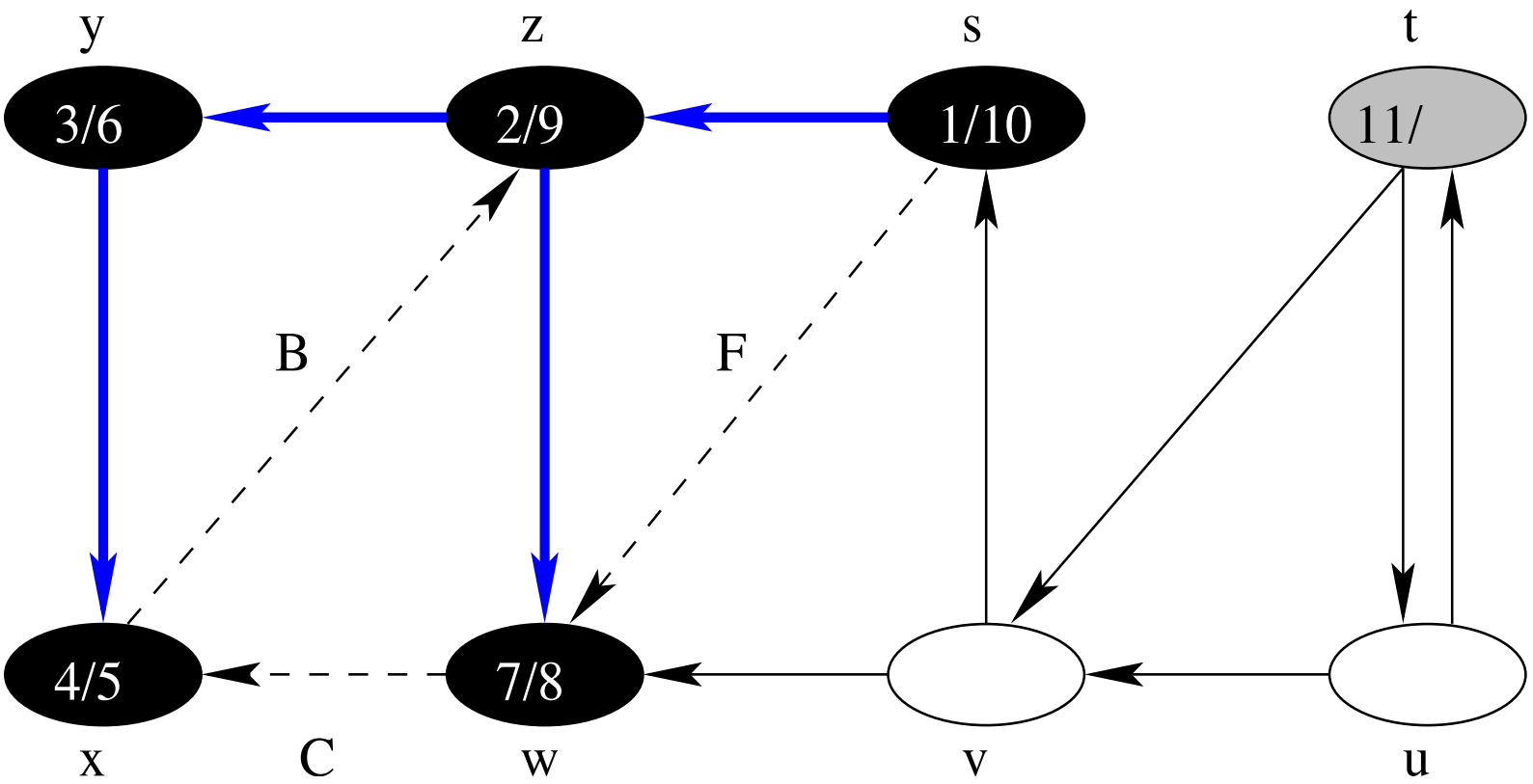
Exemplo



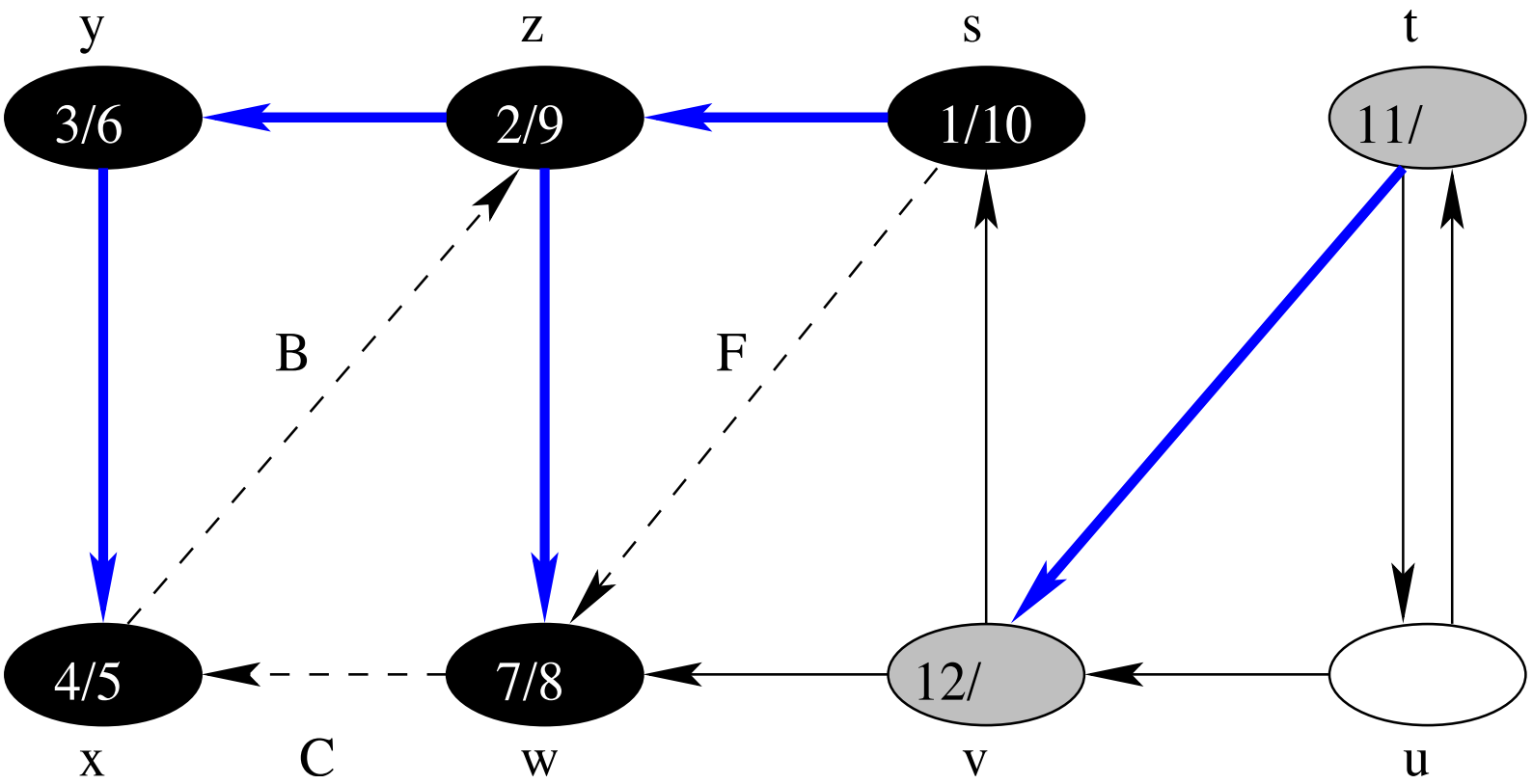
Exemplo



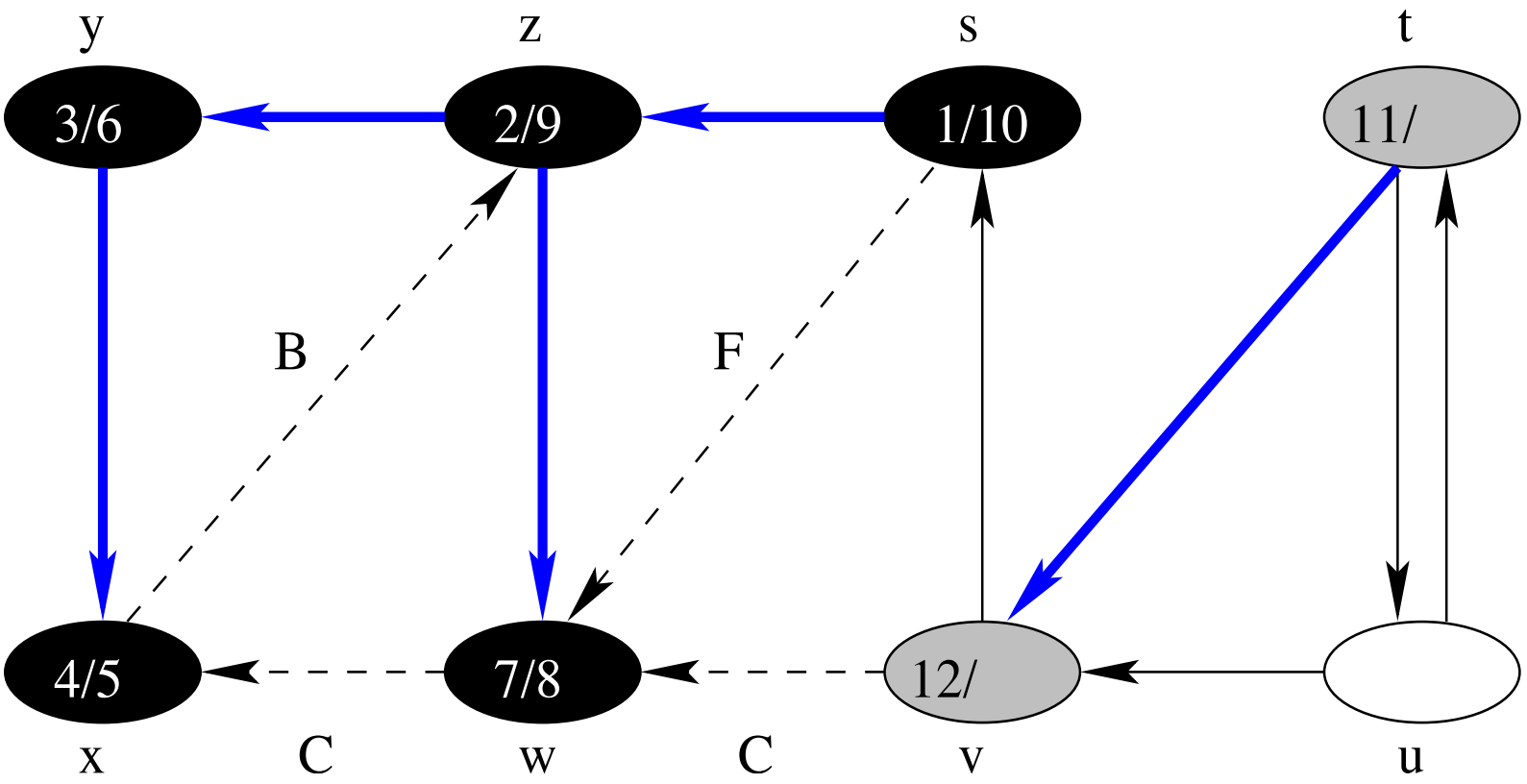
Exemplo



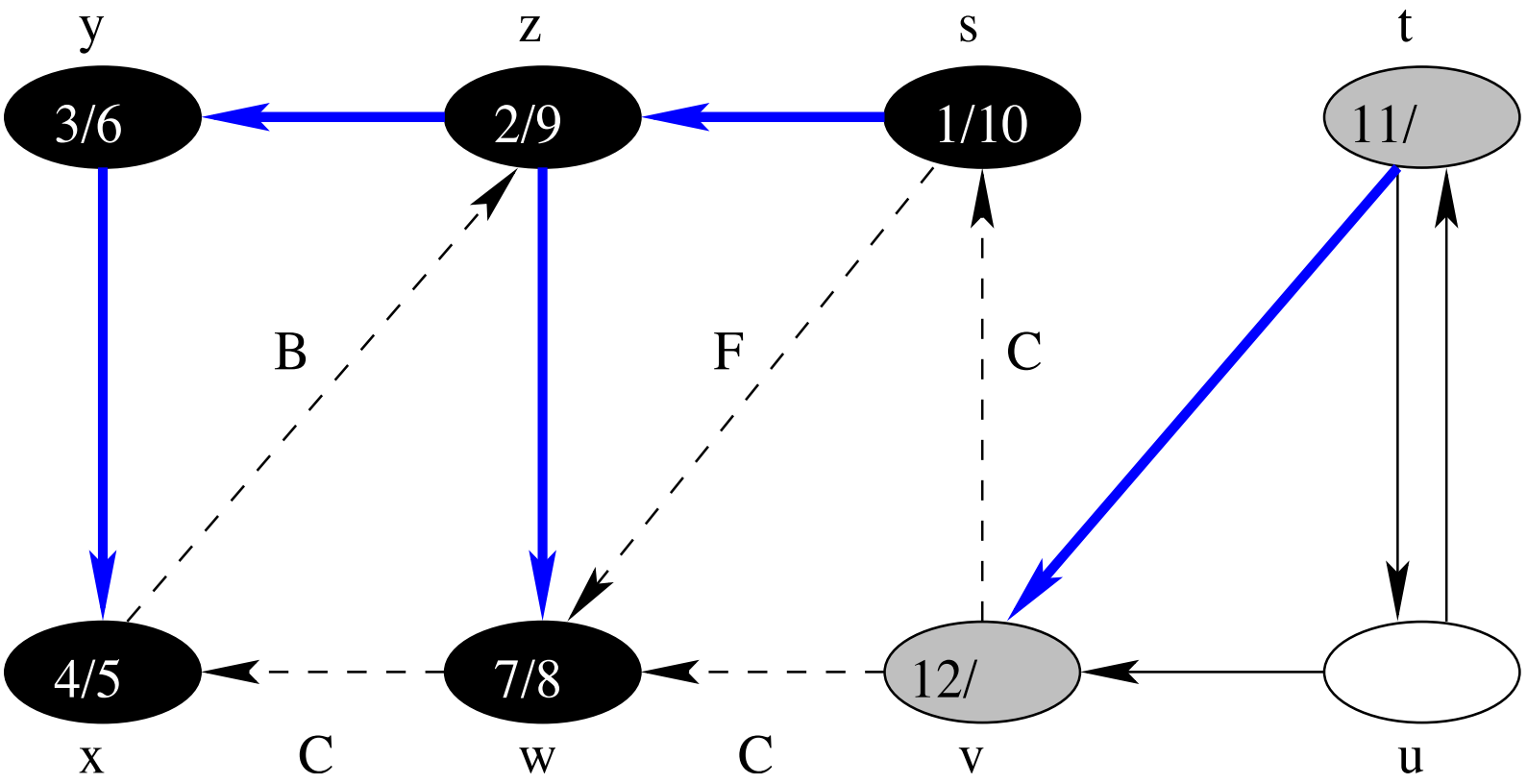
Exemplo



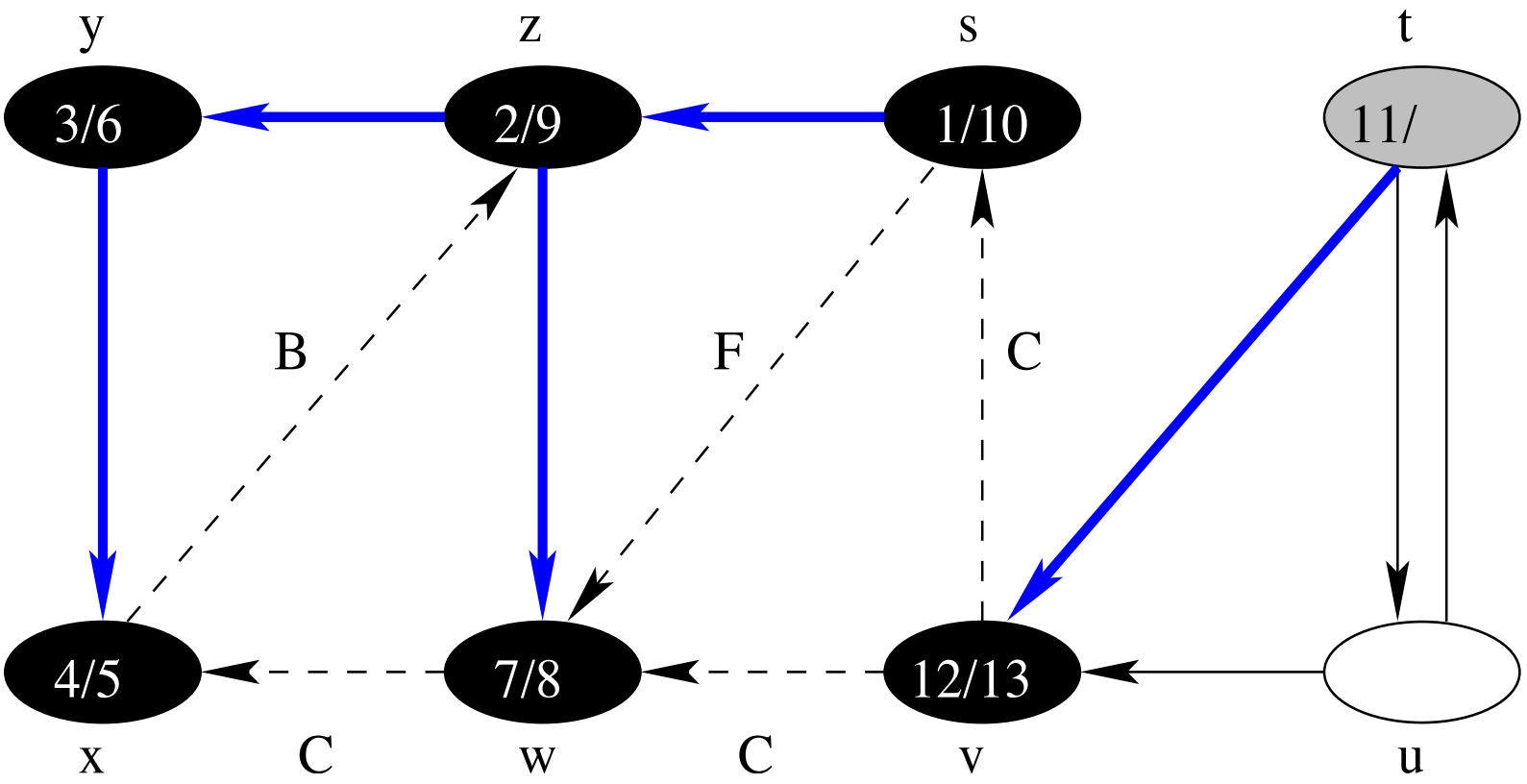
Exemplo



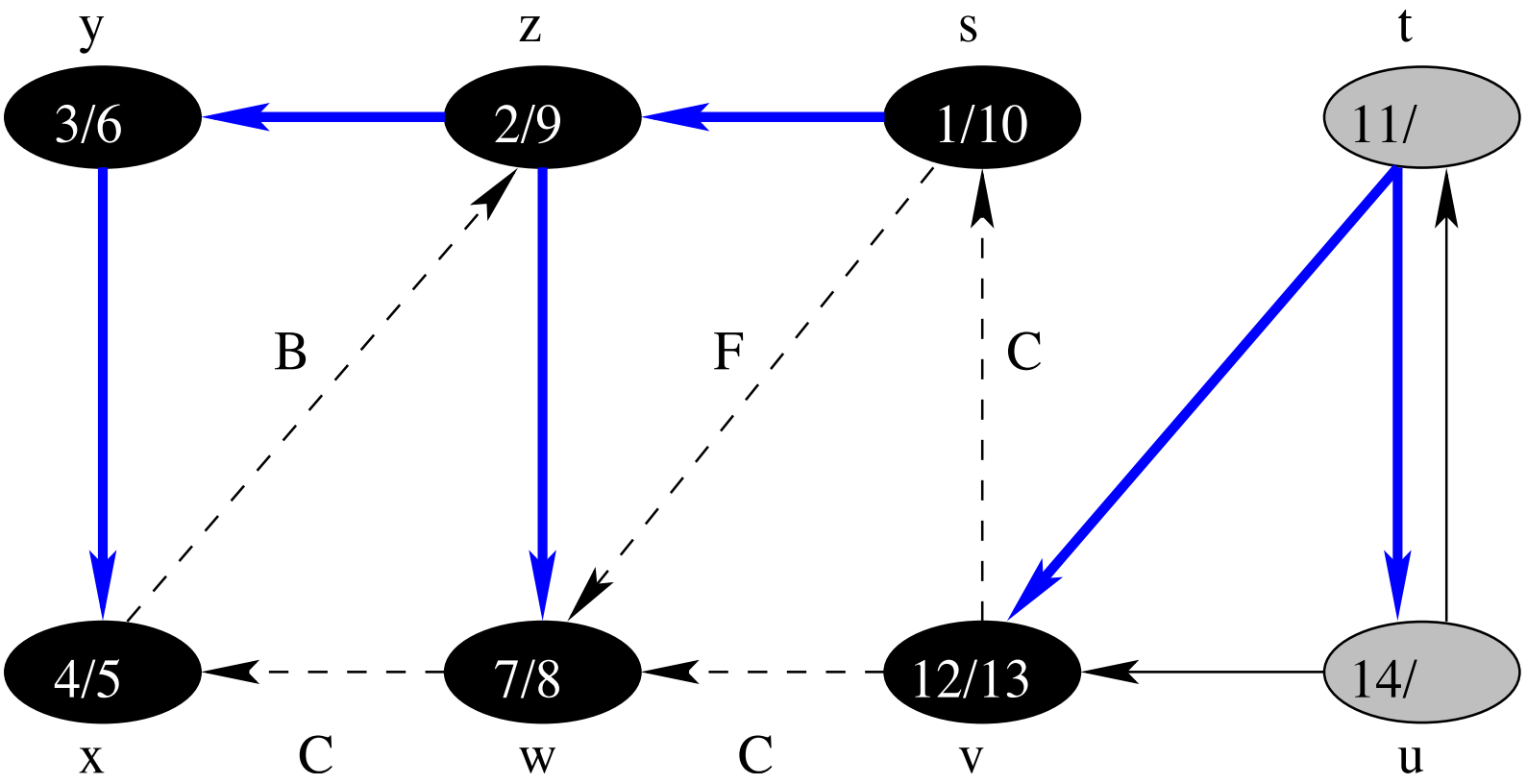
Exemplo



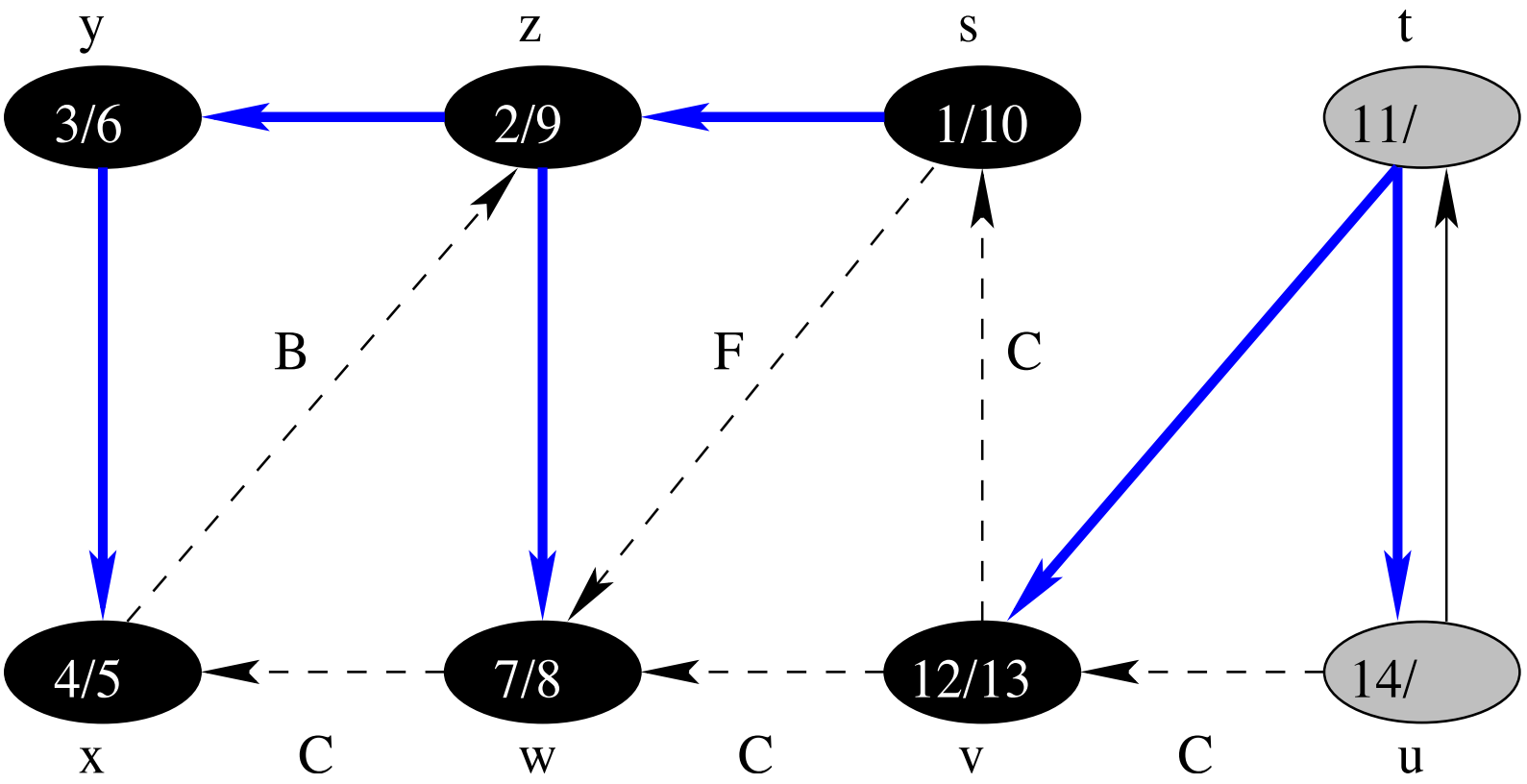
Exemplo



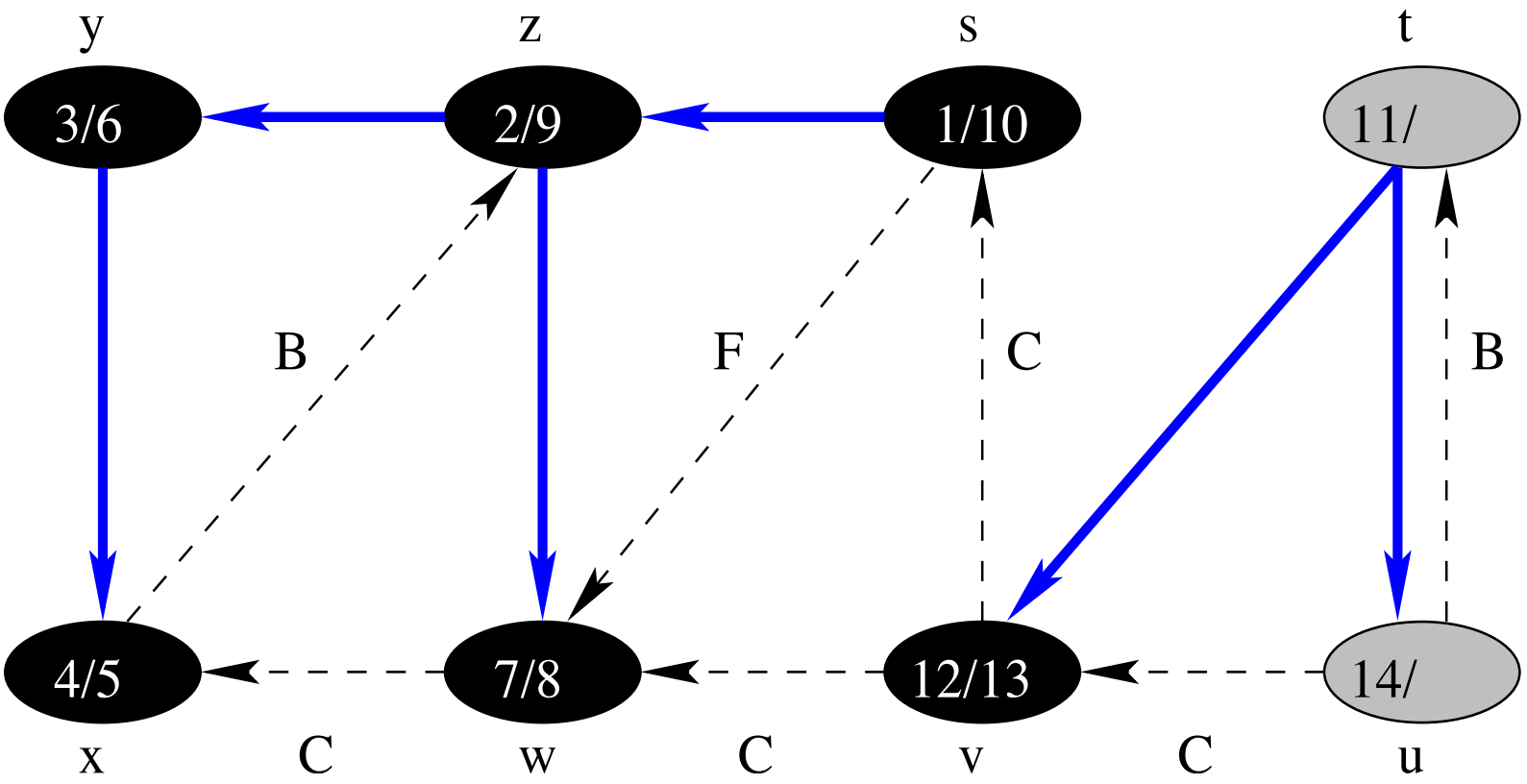
Exemplo



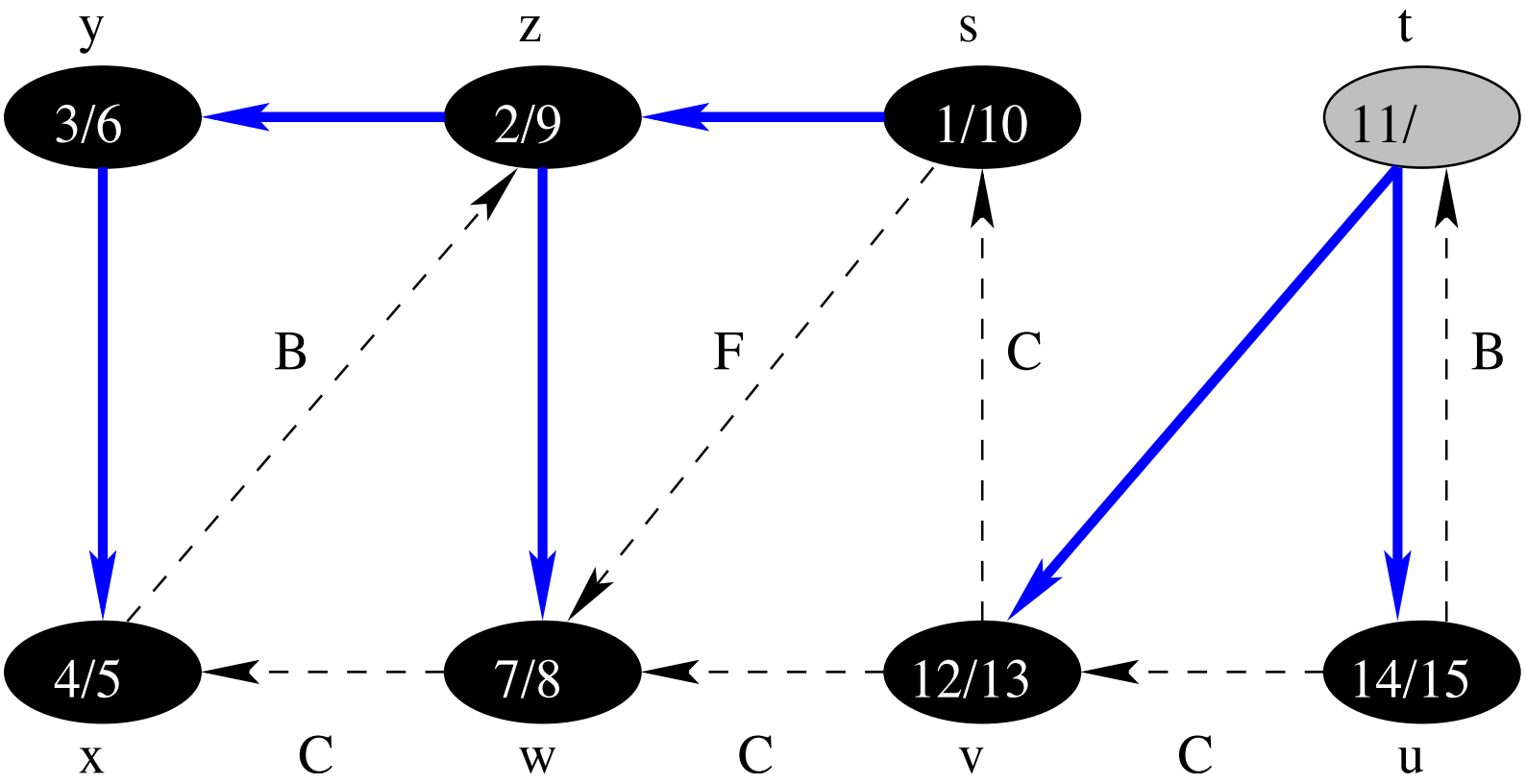
Exemplo



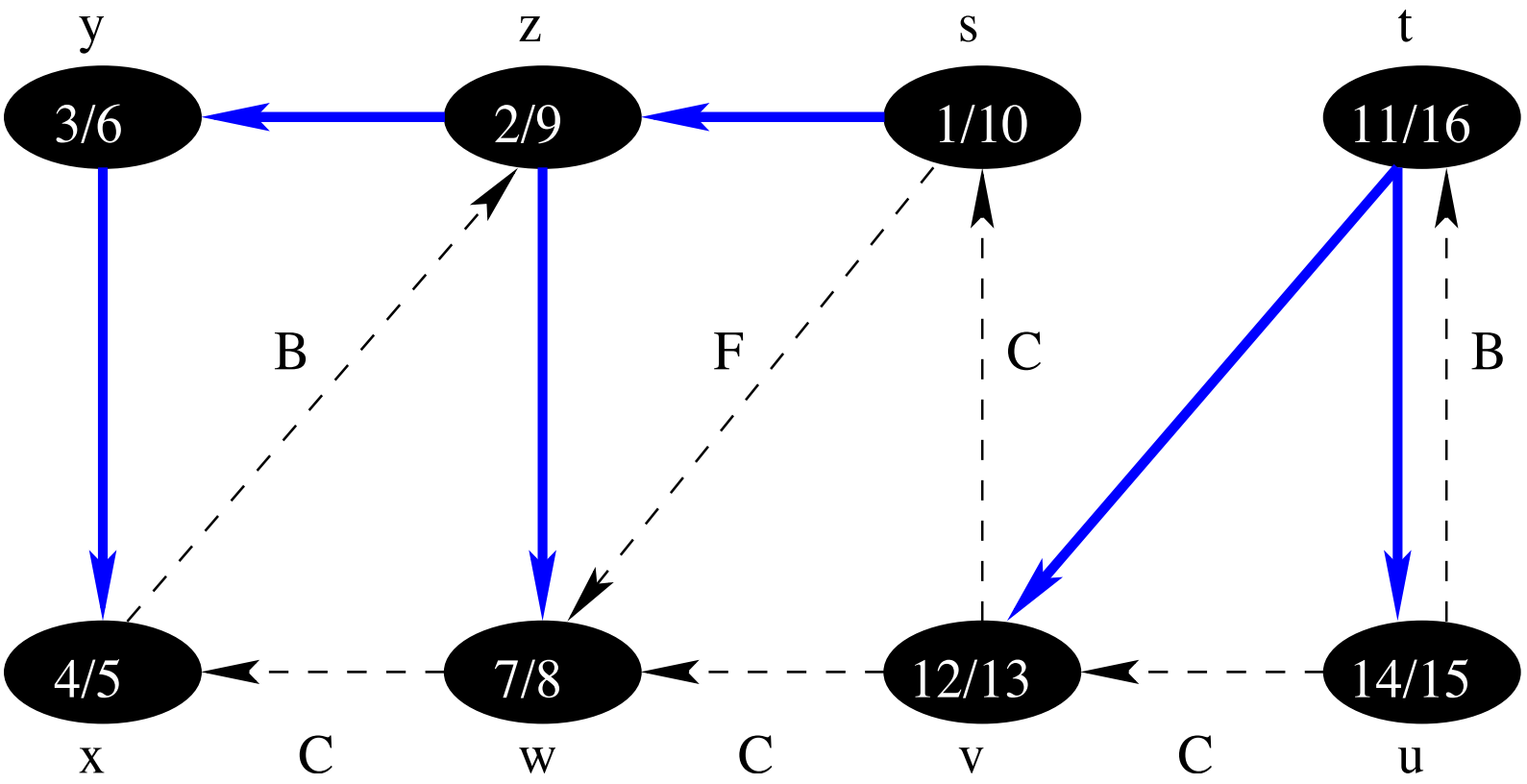
Exemplo



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 ∈ V[G] faça
2       cor[u] ← branco
3       π[u] ← NIL
4   tempo ← 0
5   para cada u ∈ V[G] faça
6       se cor[u] = branco
7           então DFS-VISIT(u)
```

Algoritmo DFS-VISIT

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

DFS-VISIT(u)

```
1   cor[u] ← cinza
2   tempo ← tempo + 1
3   d[u] ← tempo
4   para cada v ∈ Adj[u] faça
5       se cor[v] = branco
6           então π[v] ← u
7                   DFS-VISIT(v)
8   cor[u] ← preto
9   f[u] ← tempo ← tempo + 1
```

Algoritmo DFS

```
DFS(G)
1   para cada u ∈ V[G] faça
2       cor[u] ← branco
3       π[u] ← NIL
4   tempo ← 0
5   para cada u ∈ V[G] faça
6       se cor[u] = branco
7           então DFS-VISIT(u)
```

Consumo de tempo

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

Algoritmo DFS-VISIT

```
DFS-VISIT(u)
1   cor[u] ← cinza
2   tempo ← tempo + 1
3   d[u] ← tempo
4   para cada v ∈ Adj[u] faça
5       se cor[v] = branco
6           então π[v] ← u
7                 DFS-VISIT(v)
8   cor[u] ← preto
9   f[u] ← tempo ← 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 $O(V + E)$.