Conceitos de grafos

Algoritmos em Grafos - Motivação prática

- 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

- *Problema do 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.
- *Problema da 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 todos os computadores que use a menor quantidade de fibra ótica possível.
- *Problema do 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 conjunto de associações pessoa-emprego que tenha o maior número possível de pessoas.

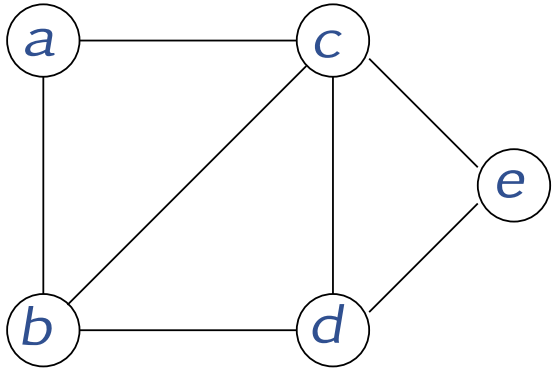
Aplicações

- *Problema do Caixeiro Viajante:* dado um conjunto de cidades, encontrar um ciclo que passa por todas as cidades tal que a distância total percorrida seja menor possível.
- *Problema Chinês do Correio:* dado o conjunto das ruas de um bairro, encontrar um passeio fechado que passa por todas as ruas tal que a distância total percorrida seja menor possível.

Definição de Grafo

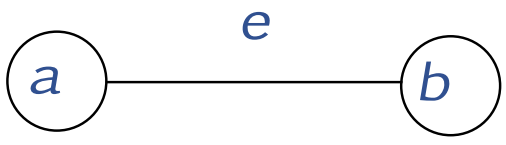
Um *grafo* é um par $G = (V, E)$ onde

- 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.

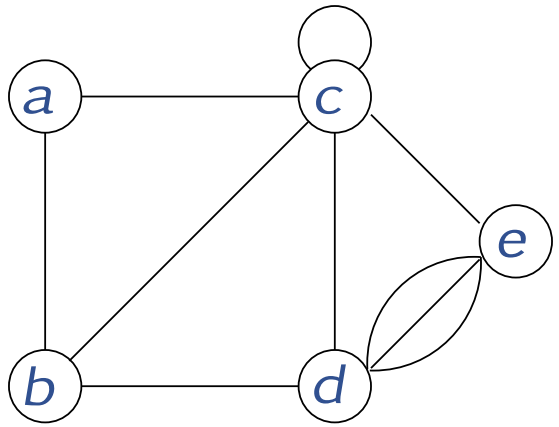


- Quando a ordem dos extremos de uma aresta não importa, temos $(a, b) = (b, a)$.

Multigrafo

Um multigrafo é um generalização de grafos que pode conter:

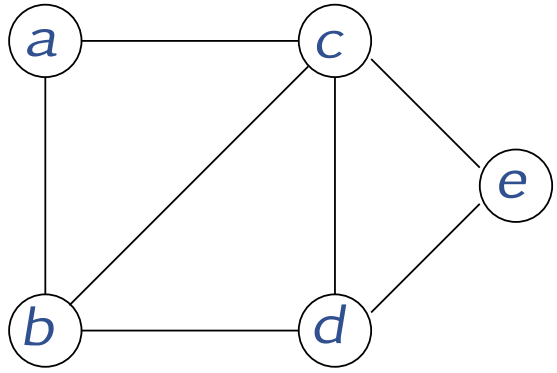
- *laço*: uma aresta com extremos idênticos
- *arestas múltiplas*: duas ou mais arestas com o mesmo par de extremos



Dizemos que um grafo é *simples* quando ele não possui laços ou arestas múltiplas.

Tamanho do Grafo

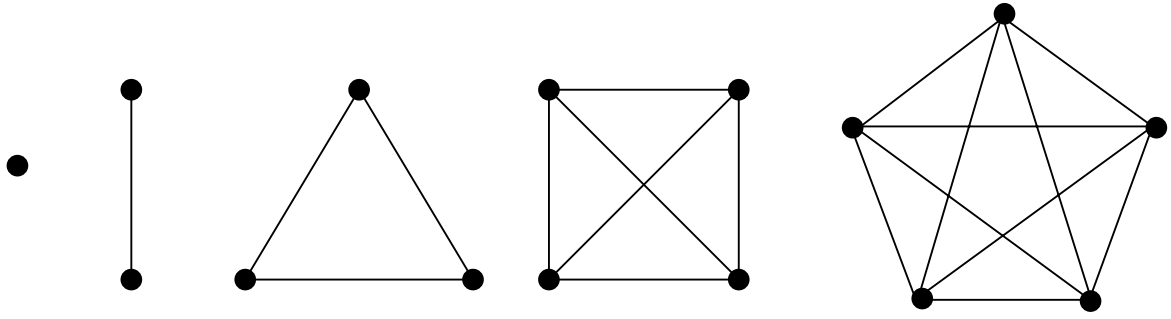
- Denotamos por $|V|$ e $|E|$ a cardinalidade dos conjuntos de vértices e arestas de um grafo $G = (V, E)$, respectivamente.
- No exemplo abaixo temos $|V| = 5$ e $|E| = 7$.



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

Grafos completos

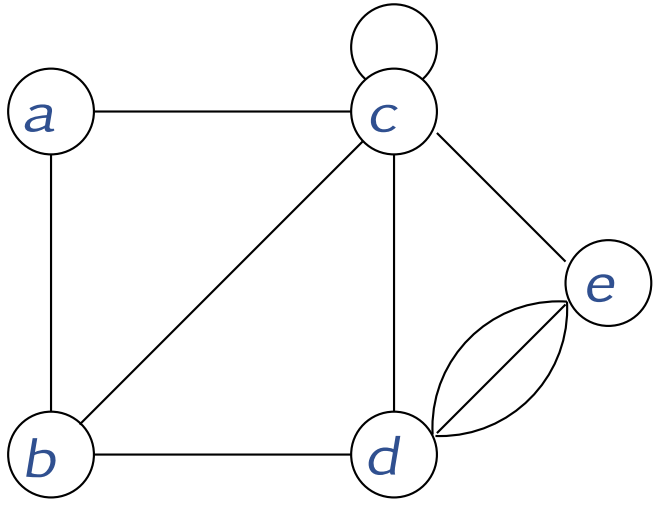
- *Grafo completo:* para todo par de vértices u, v, a aresta (u, v) pertence ao grafo.



- O número de arestas de um grafo completo com n vértices é $\binom{n}{2}$.
- O número de arestas de um grafo simples com n vértices é no máximo $\binom{n}{2}$.

Grau de um vértice

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



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

Teorema (*Handshaking lemma*)

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

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

Grafo complementar

Seja $G = (V, E)$ um grafo simples. O *complemento* de G é o grafo $\bar{G}$ com conjunto de vértices V tal que $(u, v) \in E(\bar{G})$ se e somente se $(u, v) \notin E(G)$.

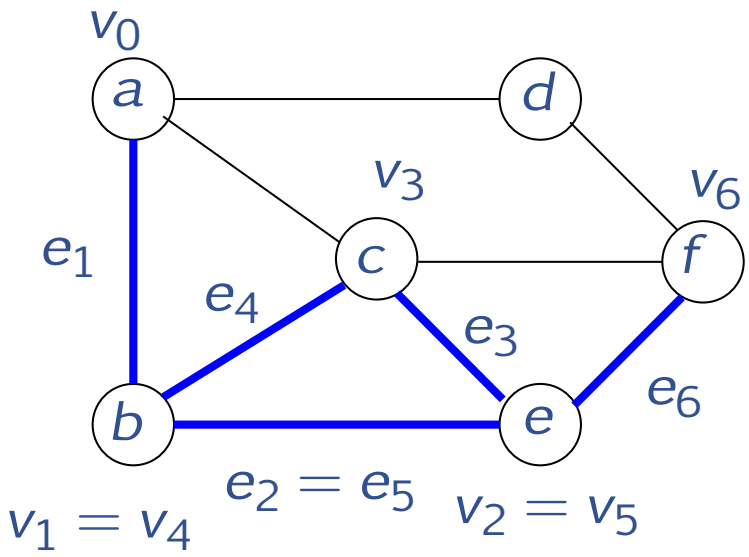
Note que $d_{\bar{G}}(v) = |V| - 1 - d_G(v)$.

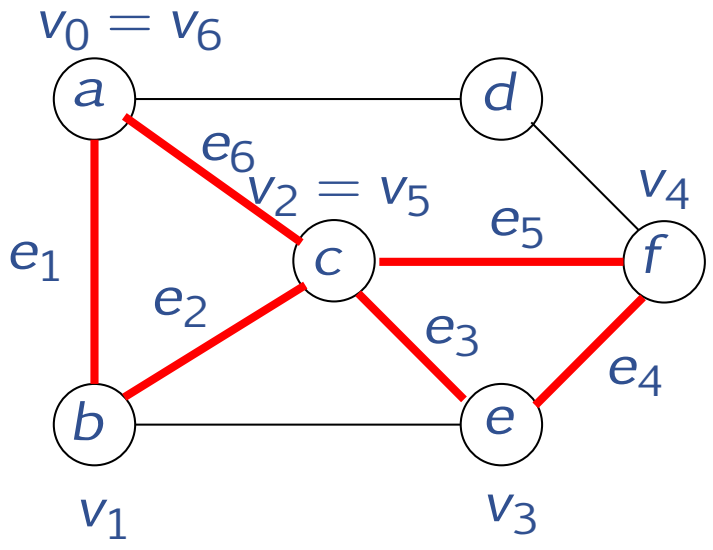
Exercício. Mostre que em uma festa com pelo menos $n \geq 6$ pessoas, existem três pessoas que se conhecem mutuamente ou três pessoas que não se conhecem mutuamente.

Exercício. Suponha que em um grupo S de n pessoas, com $n \geq 4$, vale o seguinte: em qualquer grupo $X \subseteq S$ de 4 pessoas, existe uma que conhece as demais pessoas de X.
Mostre que existe uma pessoa em S que conhece todas as demais pessoas de S.

Passeios em Grafos

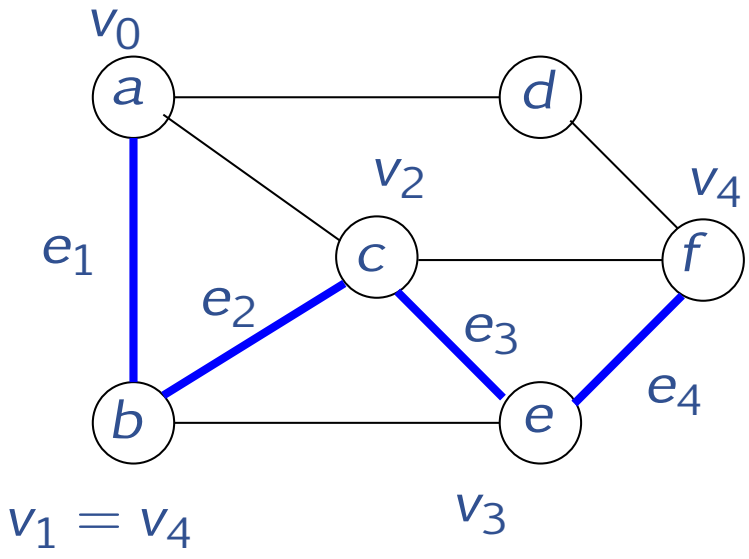
- Um *passeio* P de um vértice v_0 a um vértice v_k em um grafo G é uma sequência finita e não vazia $(v_0, e_1, v_1, \ldots, e_k, v_k)$ cujos elementos são alternadamente vértices e arestas e tal que, para todo $1 \leq i \leq k$, v_{i-1} e v_i são os extremos de e_i.
- Dizemos que P é um passeio de v_0 a v_k e que v_k é *alcançável* a partir de v_0 através de P.
- Dizemos que P é *fechado* se $v_0 = v_k$.

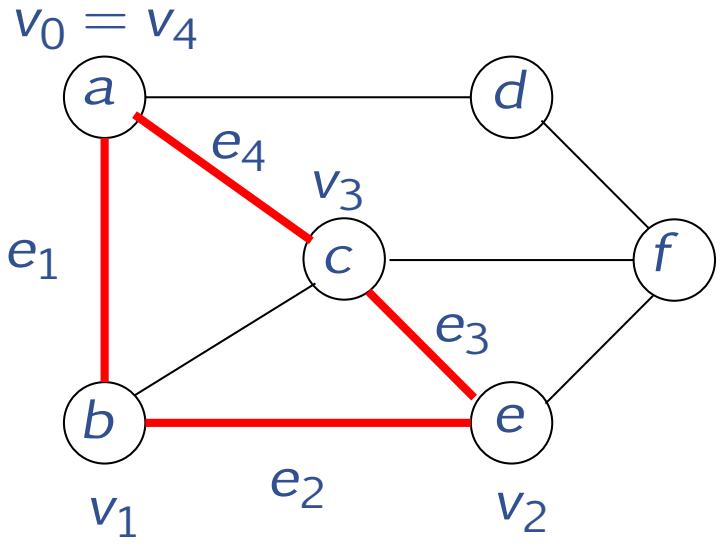




Caminhos e Ciclos

- O *comprimento* do passeio P é igual ao seu número de arestas, ou seja, k.
- Um *caminho* é um passeio em que todos seus vértices são distintos.
- Um *ciclo* é um passeio fechado que possui pelo menos uma aresta e tal que $v_2, \ldots, v_k$ são distintos e todas as arestas são distintas.





Exercícios

(1) Sejam G um grafo e u, v vértices de G. Mostre que se existe um passeio de u a v em G, então existe um caminho de u a v em G.

(2) Sejam G um grafo e u, v, w vértices de G. Mostre que se em G existem um caminho de u a v e um caminho de v a w então existe um caminho de u a w em G.

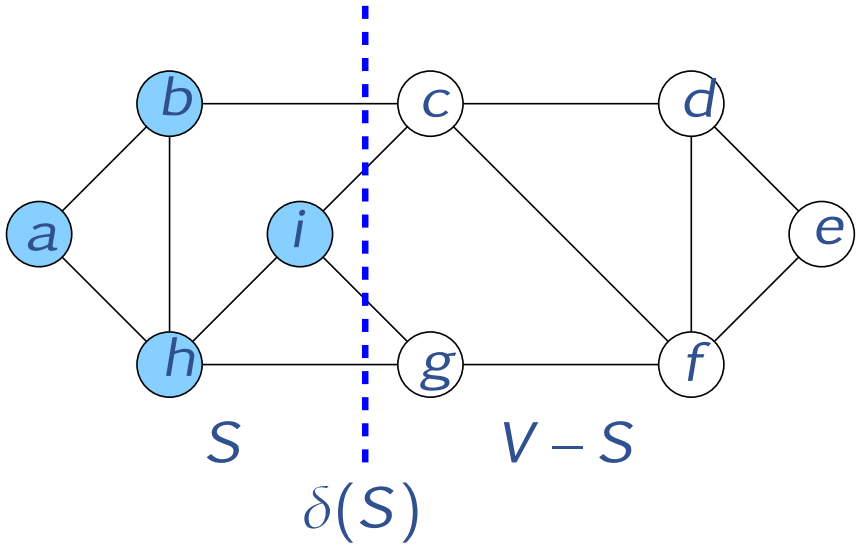
(3) É verdade que todo passeio fechado contém um ciclo?

Cortes

Seja $G = (V, E)$ um grafo e seja $S \subset V$.

Denote por $\delta_G(S)$ o conjunto de arestas de G com um extremo em S e outro em $V - S$. Dizemos que $\delta_G(S)$ é um *corte* determinado por S.

Se $s \in S$ e $t \in V - S$ dizemos que $\delta_G(S)$ *separa* s de t.



Caminhos versus Cortes

Lema. *Seja G um grafo e sejam s, t vértices distintos de G. Então exatamente um dos seguintes ocorre:*

(a) *existe um caminho de s a t em G, ou*

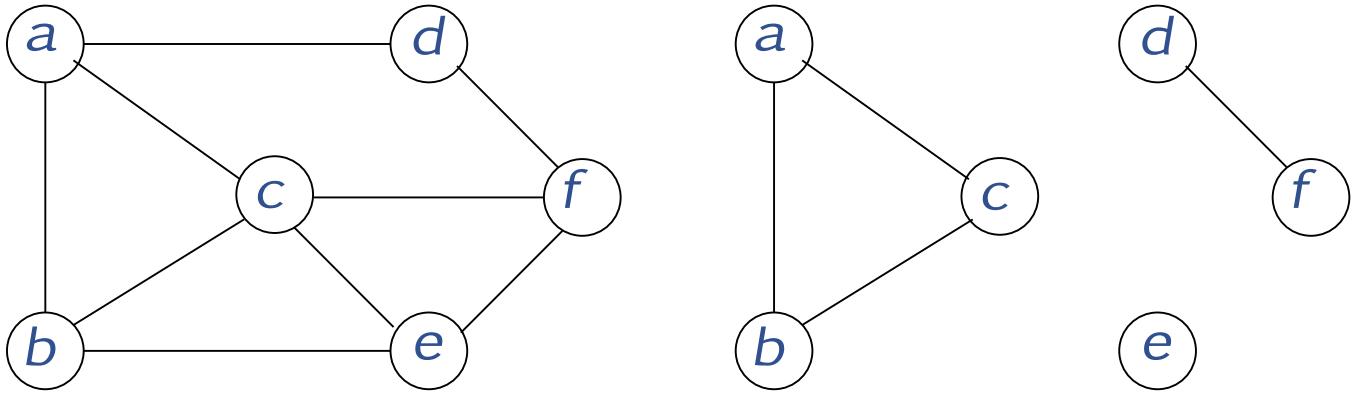
(b) *existe um corte $\delta_G(S)$ que separa s de t tal que $\delta_G(S) = \emptyset$.*

Prova: Claramente (a) e (b) não podem valer simultaneamente (Por quê?).

Suponha que (a) não vale. Seja S o conjuntos dos vértices que são alcançáveis por s em G. Obviamente, $t \in V - S$ e $\delta_G(S) = \emptyset$. □

Conexidade

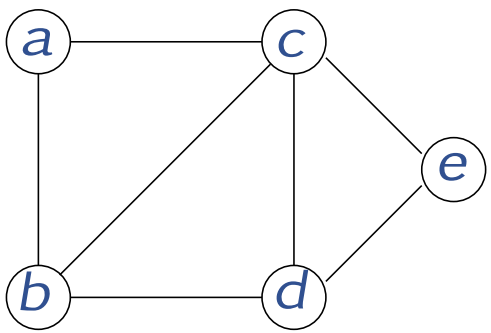
- Dizemos que um grafo é *conexo* se, para qualquer par de vértices u e v de G, existe um caminho de u a v em G. Caso contrário, dizemos que é *desconexo*.
- Quando o grafo G não é conexo, podemos particioná-lo em *componentes*. Dois vértices u e v de G estão no mesmo componente de G se existe um caminho de u a v em G.



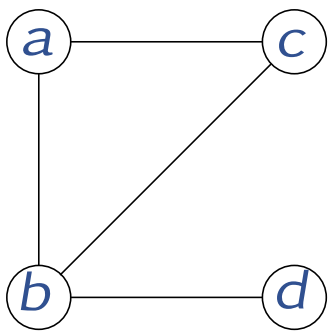
G G'

Subgrafo e Subgrafo Gerador

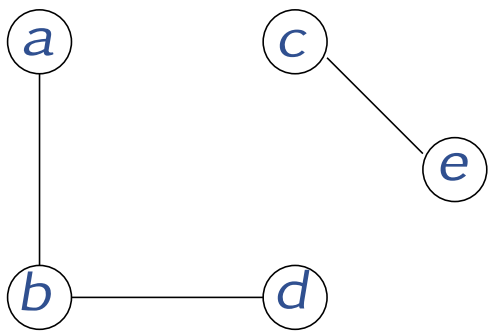
- Um *subgrafo* $H = (V', E')$ de um grafo $G = (V, E)$ é um grafo tal que $V' \subseteq V$, $E' \subseteq E$.
- Um *subgrafo gerador* de G é um subgrafo H com $V' = V$.



G



H



H

Grafos obtidos a partir de outros grafos

Seja $G = (V, E)$, e uma aresta de G e v um vértice de G.
Então

- $G - e$ é o grafo obtido de G removendo-se e. Formalmente,

$$G - e = (V, E - \{e\}).$$

- $G - v$ é o grafo obtido de G removendo-se v e todas as arestas que incidem em v. Formalmente,

$$G - v = (V - \{v\}, E - \delta(\{v\})).$$

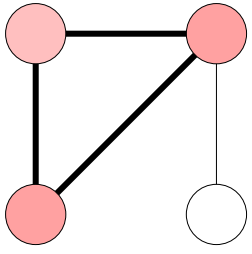
Subgrafo induzido

Seja $G = (V, E)$ e S um subconjunto de vértices.

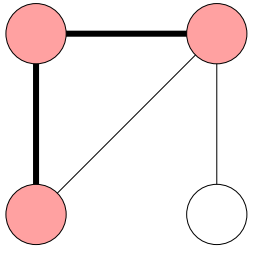
Então $G[S]$ é o subgrafo de G induzido por S, que é formado por S e todas as arestas entre vértices S.
Formalmente,

$$G[S] = (S, \{(uv) \in E | u, v \in S\}).$$

Exemplo de um subgrafo que é induzido e de outro subgrafo que não é induzido em um grafo G:



INDUZIDO

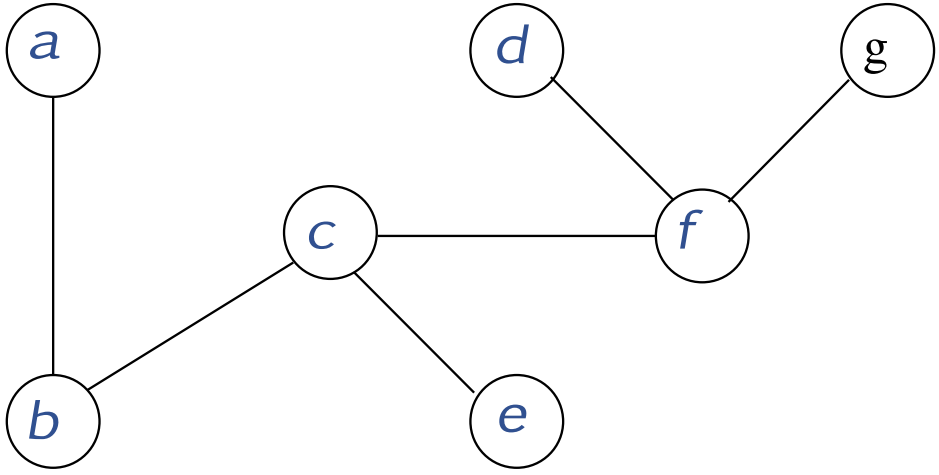


NÃO INDUZIDO

Fatos básicos de grafos

Árvores

- Um grafo $G = (V, E)$ é uma *árvore* se é conexo e não possui ciclos (acíclico).



- Uma *folha* de uma árvore G é um vértice de grau 1.
- *Toda árvore com pelo menos dois vértices possui uma folha.*(Por quê?)

Árvores

Teorema. *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 (conexo minimal).*
- *Para todo par de vértices* u, v *de* G, *existe um único caminho de* u *a* v *em* G *(e* G *não tem laços).*

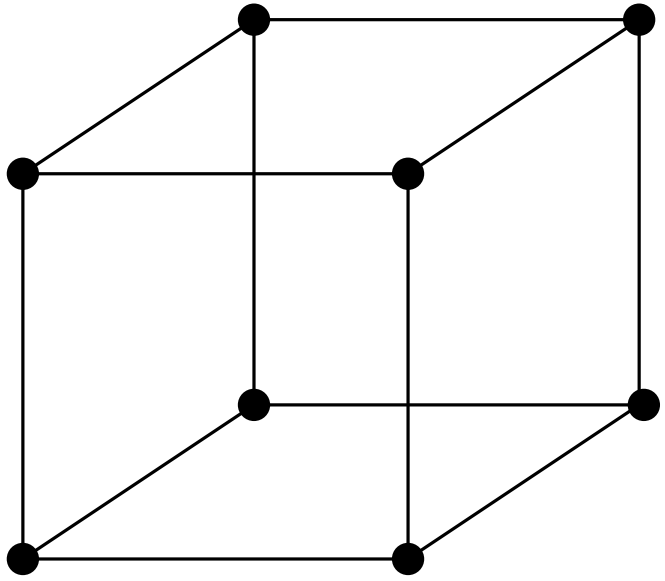
Grafos bipartidos

Uma *partição* de um conjunto V é um conjunto de subconjuntos não-vazios de V, $\{V_1, \ldots, V_k\}$ (ou $(V_1, \ldots, V_k)$) tal que:

- $V_i \cap V_j = \emptyset$, para todo $1 \leq i < j \leq k$, e
- $V_1 \cup \ldots \cup V_k = V$.

Um grafo $G = (V, E)$ é *bipartido* se existe uma partição (A, B) de V tal que toda aresta de G tem um extremo em A e outro em B.

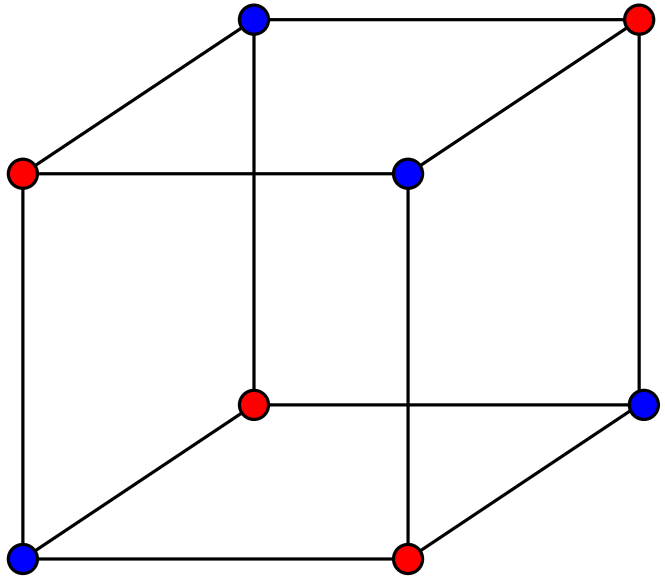
Grafos bipartidos



É bipartido?

Vamos indicar cada parte com uma cor: vermelho ou azul.

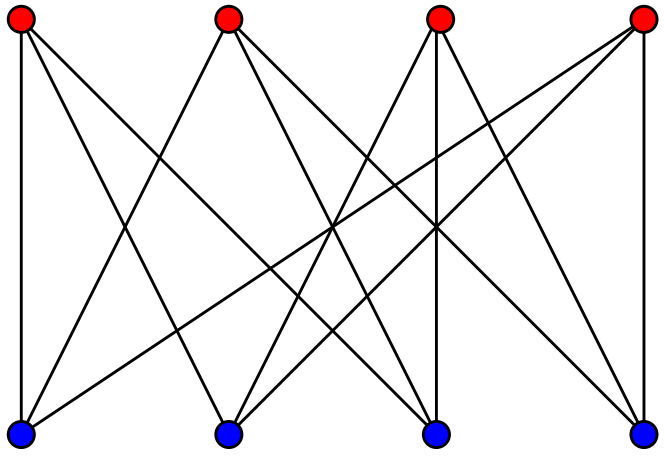
Grafos bipartidos



É bipartido!

Um grafo $G = (V, E)$ é *bipartido* se é possível colorir os vértices de G com **duas cores** de modo que vértices adjacentes tenham cores distintas.

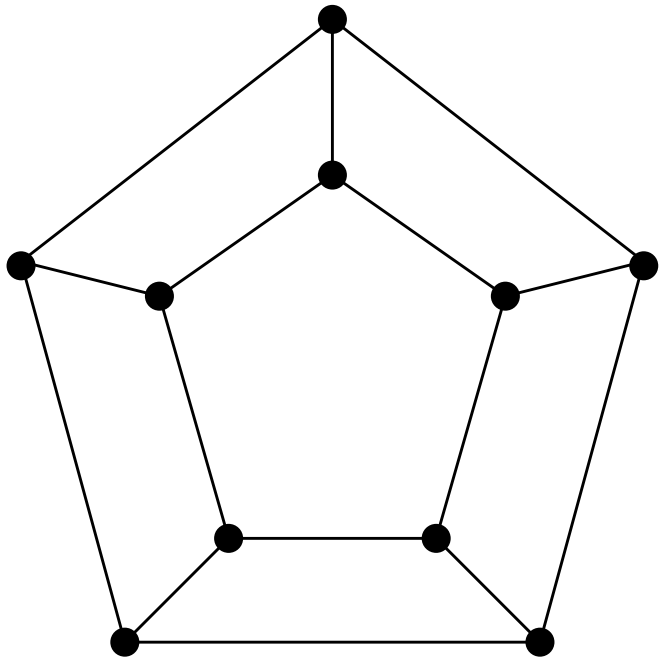
Grafos bipartidos



Isto pode ser visto melhor com outro desenho.

Um grafo $G = (V, E)$ é *bipartido* se é possível colorir os vértices de G com **duas cores** de modo que vértices adjacentes tenham cores distintas.

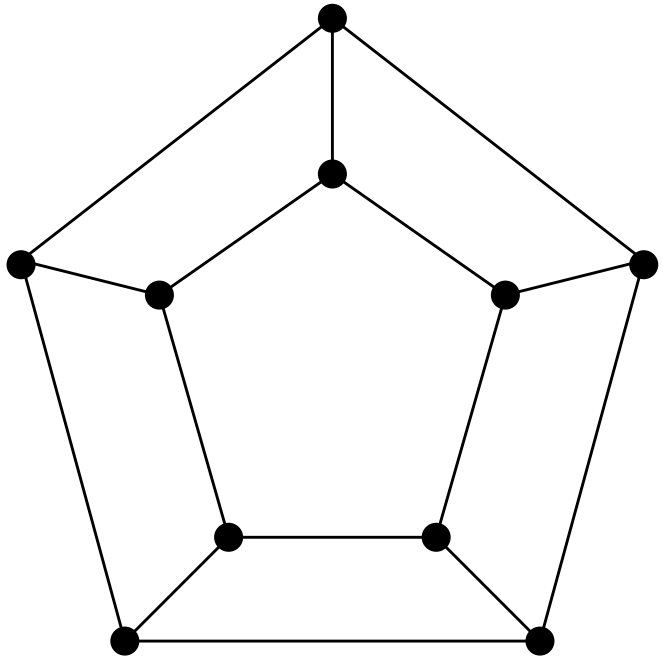
Grafos bipartidos



Este grafo **não** é bipartido.

Você consegue apresentar uma justificativa simples deste fato?

Grafos bipartidos



Um grafo bipartido **não** pode conter *ciclos ímpares* (de comprimento ímpar). Claramente, esta é uma condição necessária.

Ela é suficiente? Ou seja, é verdade que se G não contém ciclos ímpares então G é bipartido? SIM!

Grafos bipartidos

Queremos demonstrar o seguinte.

Teorema. *Seja G um grafo. Então G é bipartido se e somente se G não contém um ciclo ímpar.*

Já vimos que se G contém um ciclo ímpar então G não é bipartido provando a implicação "$\Rightarrow$".

Falta provar a recíproca "$\Leftarrow$". Para provar isto precisaremos de alguns fatos.

Podemos supor que G é conexo. (Por quê?)

Árvore geradora

Fato 1. *Todo grafo conexo contém uma árvore geradora.*

Isto segue facilmente do próximo resultado.

Lema. *Seja G um grafo conexo e seja C um ciclo de G. Se e é uma aresta de C então $G - e$ é conexo.* (Exercício!)

A recíproca também vale.

Lema. *Seja G um grafo conexo e seja e uma aresta de G. Se $G - e$ é conexo então e pertence a algum ciclo de G.* (Exercício!)

Árvores e grafos bipartidos

Fato 2. *Toda árvore $T = (V, E)$ é um grafo bipartido.*

A prova é por indução em $|V|$ e do fato de que toda árvore com pelo menos dois vértices contém uma folha. (Exercício!)

Árvore geradora

Fato 3. *Seja $T = (V, E')$ uma árvore geradora de um grafo $G = (V, E)$. Então para toda aresta $e \in E - E'$ existe um único ciclo em $T + e := (V, E' \cup \{e\})$.*

Prova: Sejam u, v os extremos de e. Como T é uma árvore, existe um único caminho P de u a v em T. Logo, $P + e$ é o único ciclo em $T + e$. □

É comum chamar o único ciclo de $T + e$ de ciclo fundamental de $T + e$.

Grafos bipartidos

Agora estamos prontos para demonstrar o seguinte.

Teorema. *Seja G um grafo. Então G é bipartido se e somente se G não contém um ciclo ímpar.*

Prova: Já vimos que se G contém um ciclo ímpar então G não é bipartido provando a implicação $"\Rightarrow"$.

Agora provaremos a recíproca $"\Leftarrow"$. Suponha que G não contém um ciclo ímpar. Construiremos uma bipartição (A, B) de V tal que toda aresta de G tem um extremo em A e outro em B.

Podemos supor que G é conexo.

Grafos bipartidos

Pelo Fato 1, G contém uma árvore geradora $T = (V, E')$.

Pelo Fato 2, T possui uma uma bipartição (A, B) de V tal que toda aresta de T tem um extremo em A e outro em B.

Mostraremos que toda aresta de $E - E'$ tem um extremo em A e outro em B, o que implica que G é bipartido.

Grafos bipartidos

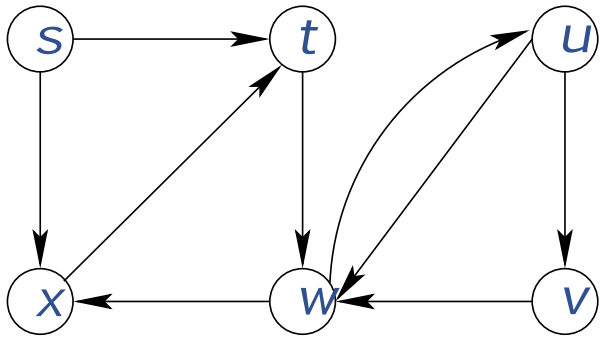
Seja e uma aresta de $E - E'$. Pelo Fato 3, existe um único ciclo C em $T + e$ (que contém e).

- Se e tem extremos em partes distintas, então nada há a fazer.
- Se os extremos de e pertencem a mesma parte (A ou B), então C é um ciclo ímpar (contradição)!

O resultado segue. □

Grafo direcionado

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



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

Grafo direcionado

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

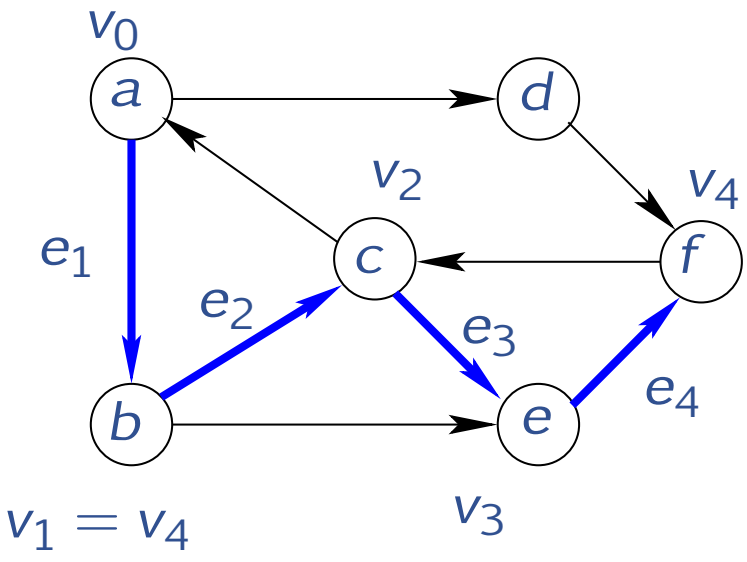
Teorema.

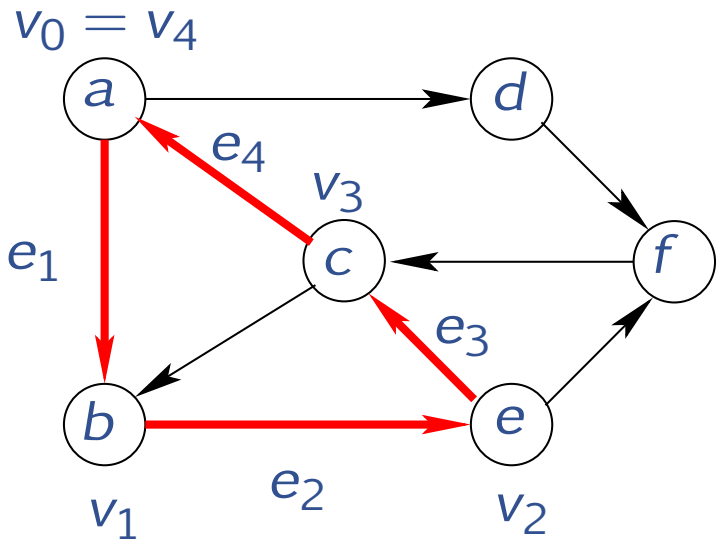
Para todo grafo direcionado $G = (V, E)$ temos:

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

Passeios em grafos direcionados.

- Supõe-se que passeios, caminhos e ciclos de grafos direcionados são direcionados (todas as arestas "seguem o mesmo sentido").





- Podemos definir subgrafo (gerador) de um grafo direcionado de modo análogo ao caso não direcionado. Há também um conceito de conexidade para grafos direcionados que veremos mais tarde.

Exercícios

(1) Sejam G um grafo direcionado e u, v vértices de G. Mostre que se existe um passeio de u a v em G, então existe um caminho de u a v em G.

(2) Sejam G um grafo direcionado e u, v, w vértices de G. Mostre que se em G existem um caminho de u a v e um caminho de v a w então existe um caminho de u a w em G.

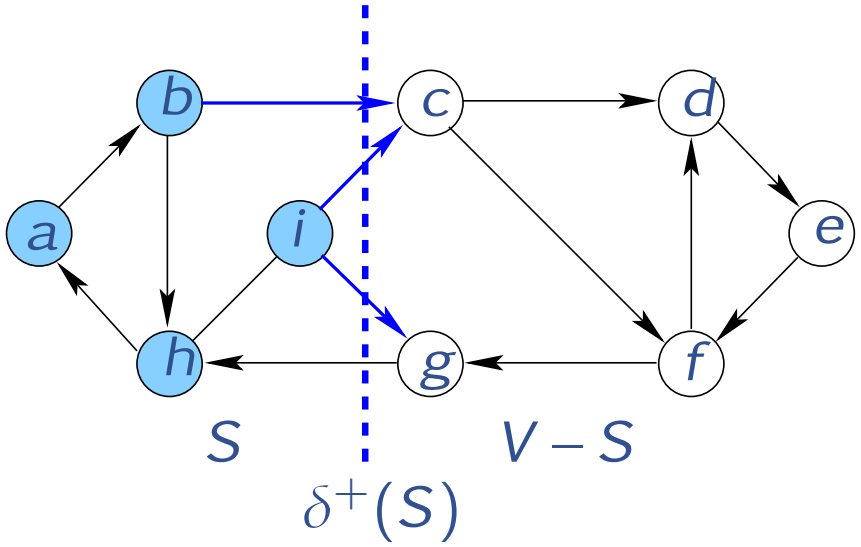
(3) É verdade que todo passeio fechado em um grafo direcionado contém um ciclo (direcionado)?

Cortes em grafos direcionados

Seja $G = (V, E)$ um grafo direcionado e seja $S \subset V$.

Denote por $\delta^+_G(S)$ o conjunto de arestas de G com cauda em S e cabeça em $V - S$. Dizemos que $\delta^+_G(S)$ é um *corte direcionado*.

Se $s \in S$ e $t \in V - S$ dizemos que $\delta^+_G(S)$ *separa* s de t.



Note que (g, h) não pertence a $\delta^+_G(S)$.

Caminhos versus Cortes

Lema. *Seja G um grafo direcionado e sejam s, t vértices distintos de G. Então exatamente um dos seguintes ocorre:*

(a) *existe um caminho de s a t em G, ou*

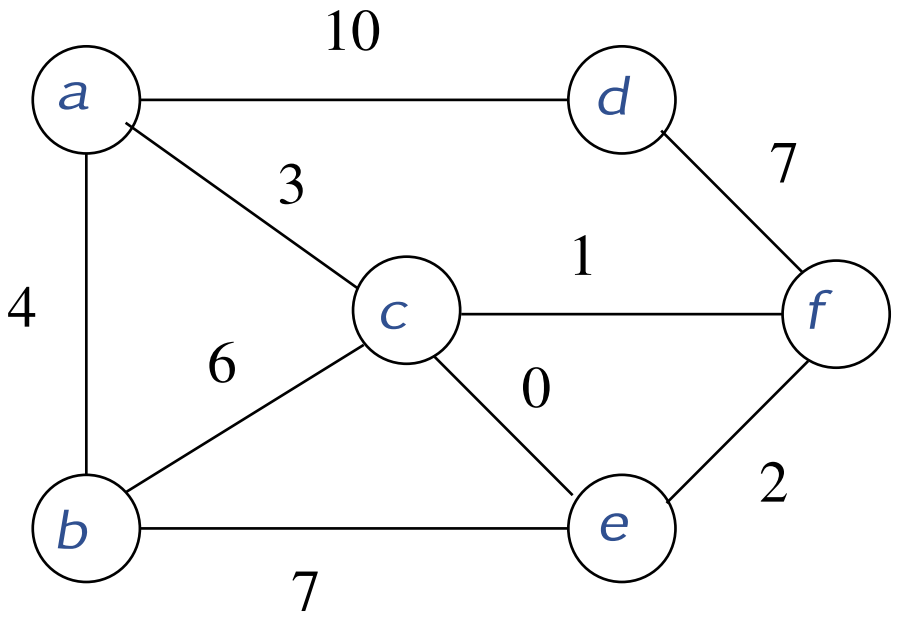
(b) *existe um corte direcionado $\delta^+_G(S)$ que separa s de t tal que $\delta^+_G(S) = \emptyset$.*

Prova: Claramente (a) e (b) não podem valer simultaneamente (Por quê?).

Suponha que (a) não vale. Seja S o conjuntos dos vértices que são alcançáveis por s em G. Obviamente, $t \in V - S$ e $\delta^+_G(S) = \emptyset$. □

Grafo Ponderado

- Um grafo (direcionado 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.



Representação de grafos

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.

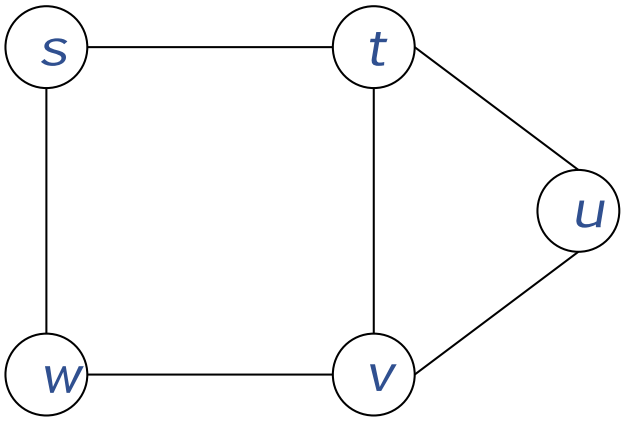
Matriz de adjacência

- Seja $G = (V, E)$ um grafo simples (direcionado 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:

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

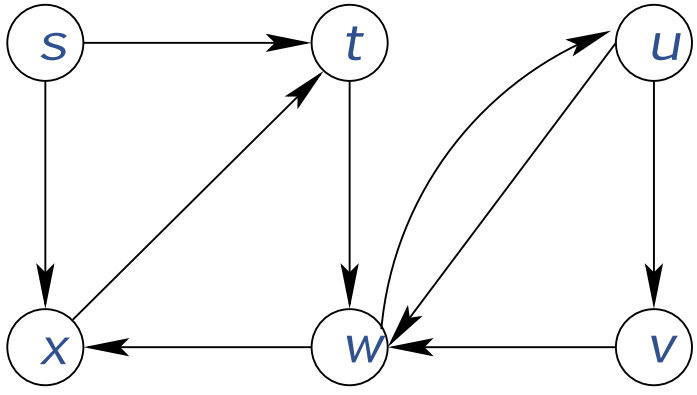
- Note que se G é não direcionado, então a matriz A correspondente é simétrica.

Matriz de adjacência



	s	t	w	v	u
s	0	1	1	0	0
t	1	0	0	1	1
w	1	0	0	1	0
v	0	1	1	0	1
u	0	1	0	1	0

Matriz de adjacência



	s	t	u	x	w	v
s	0	1	0	1	0	0
t	0	0	0	0	1	0
u	0	0	0	0	1	1
x	0	1	0	0	0	0
w	0	0	1	1	0	0
v	0	0	0	0	1	0

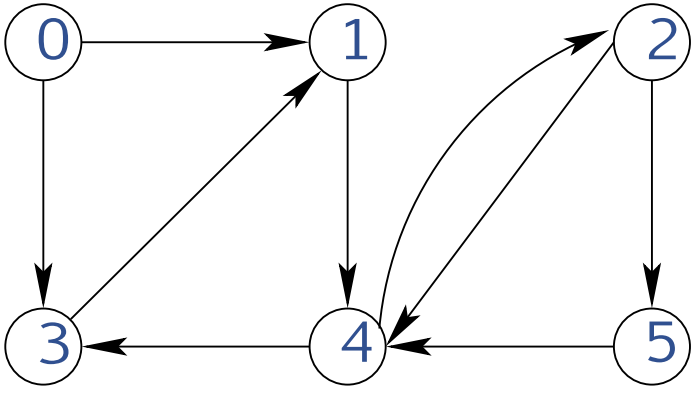
Matriz de adjacência em C

Em C para representar os vértices de um grafo G com n vértices usamos $0, 1, 2, \ldots, n-2, n-1$.
A declaração de uma matriz poderia ser:

```
#define NMAX 100
unsigned char A[NMAX][NMAX]; /* matriz estática */
```

ou

```
unsigned char **A; /* matriz dinâmica */
```



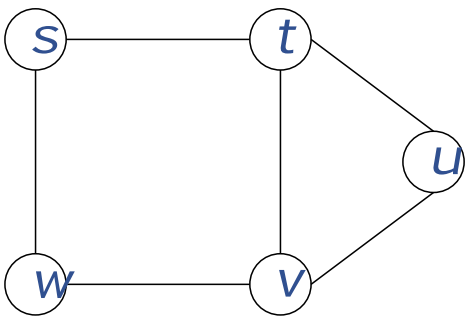
	0	1	2	3	4	5
0	0	1	0	1	0	0
1	0	0	0	0	1	0
2	0	0	0	0	1	1
3	0	1	0	0	0	0
4	0	0	1	1	0	0
5	0	0	0	0	1	0

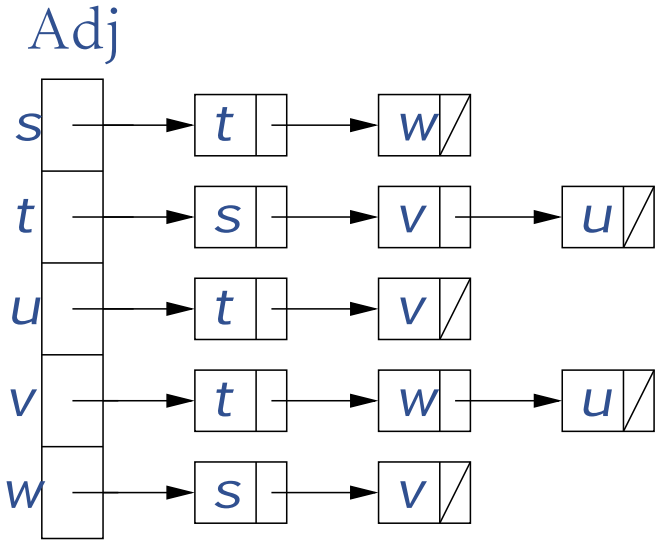
Listas de adjacência

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

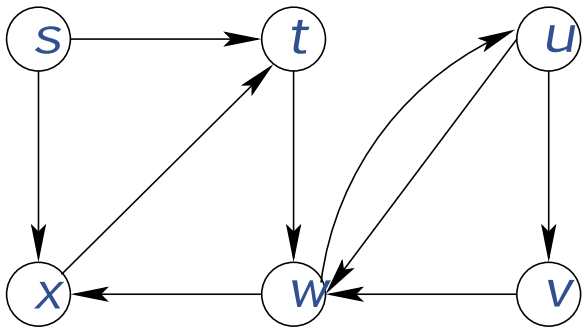
  Para cada vértice v, temos uma lista ligada $\mathrm{Adj}[v]$ dos vértices *adjacentes* a v, ou seja, w aparece em $\mathrm{Adj}[v]$ se (v, w) é uma aresta de G. Os vértices podem estar em qualquer ordem em uma lista.

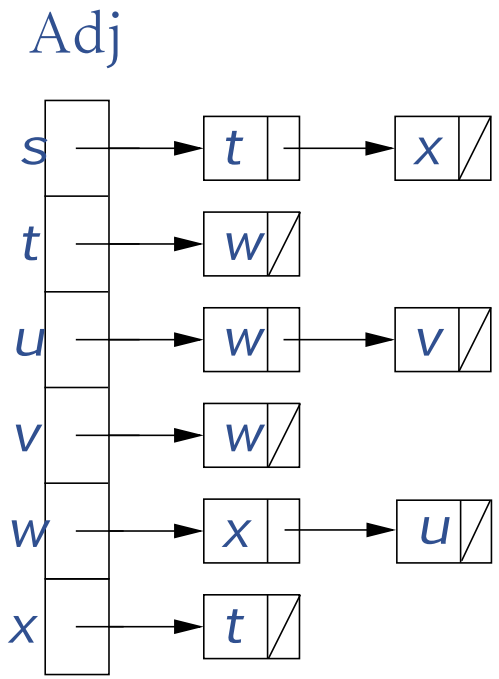
Listas de adjacência





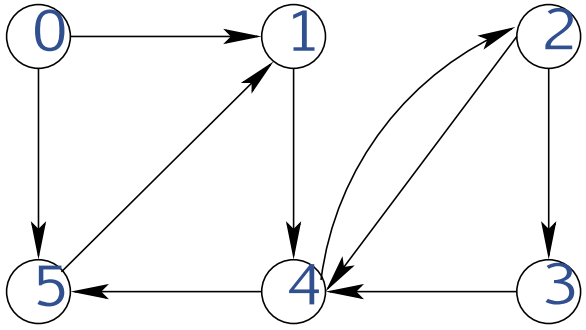
Lista de adjacências

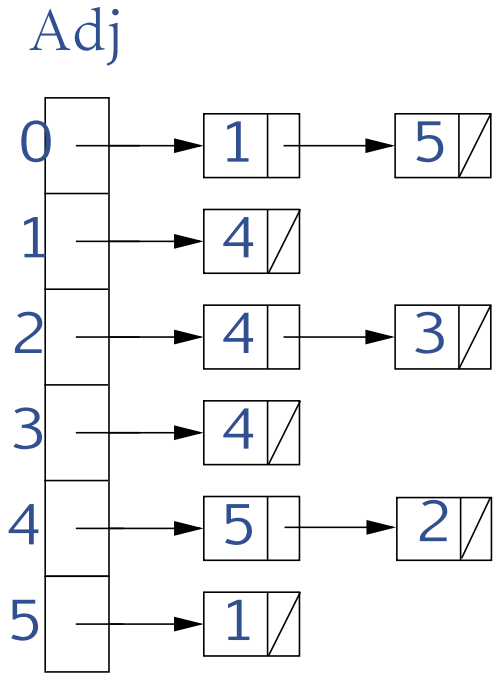




Listas de adjacências em C

```
#define NMAX 100
typedef struct SVertex SVertex;
struct SVertex {
 int vert;
 SVertex *next;
}
SVertex *Adj[NMAX]; /* Vetor estático */
```





Notação para complexidade de algoritmos

Quando analisarmos a complexidade de um algoritmo envolvendo um grafo $G = (V, E)$ usaremos V e E na notação assintótica, em vez de $|V|$ e $|E|$.

Por exemplo, escrevemos $O(E^2 \lg V)$ em vez de $O(|E|^2 \lg |V|)$.

Matriz × Lista de adjacência

O que é melhor? **Depende**.

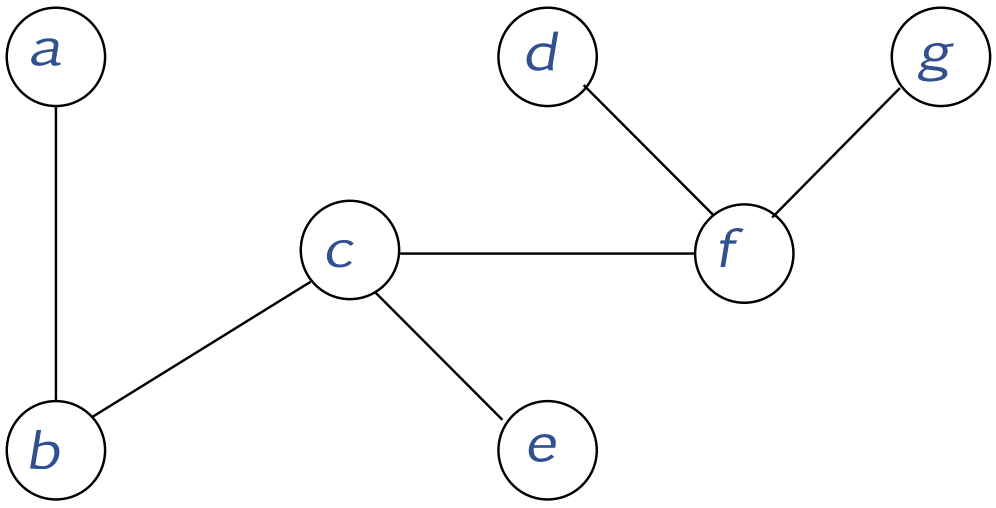
- Matriz de adjacência: é fácil verificar se (i, j) é uma aresta de G.
- Lista de adjacência: é fácil descobrir os vértices adjacentes a um dado vértice v (ou seja, listar $\mathrm{Adj}[v]$).
- Matriz de adjacência: espaço $\Theta(V^2)$.
  Adequada a grafos densos ($|E| = \Theta(V^2)$).
- Lista de adjacência: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

Representação de árvores

Uma *árvore enraizada* é uma árvore com um vértice especial chamado *raiz*.

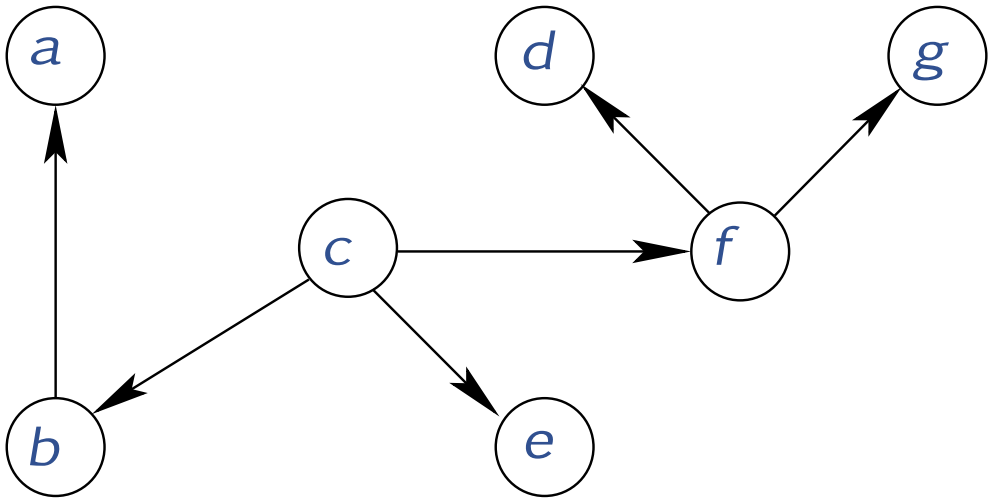


raiz c

Representação de árvores

Uma *árvore direcionada com raiz r* é um grafo direcionado **acíclico** $T = (V, E)$ tal que:

1. $g^-(r) = 0$,
2. $g^-(v) = 1$ para $v \in V - \{r\}$.



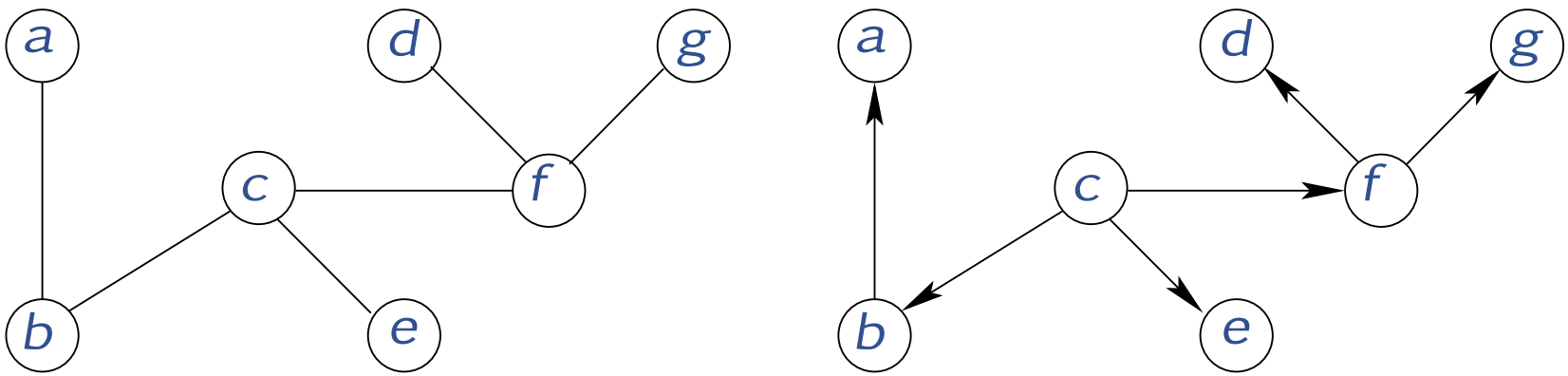
raiz c

Representação de árvores

Vetor de predecessores π (ou outro nome):

v	a	b	c	d	e	f	g
$\pi[v]$	b	c	N	f	c	c	f

N é um símbolo usado para indicar não existência.



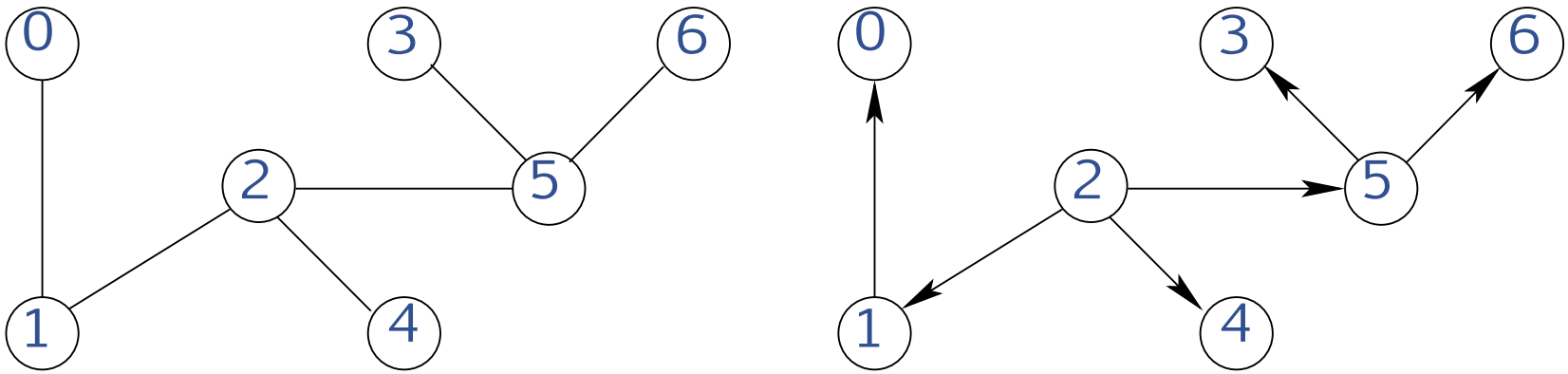
raiz c

Representação de árvores em C

Vetor de predecessores π:

v	0	1	2	3	4	5	6
$\pi[v]$	1	2	N	5	2	2	5

N é um símbolo usado para indicar não existência. Por exemplo -1, NULL etc.



raiz 2

Alguns detalhes de implementação

- Nos algoritmos que veremos durante o semestre, usa-se a representação de um grafo (direcionado ou não) por listas de adjacências.
- Em uma implementação de verdade de um algoritmo, o mais provável é que esta representação **não** seja dada a priori.
- Em geral é necessário construir tal representação a partir da entrada dada. Como fazer isto depende do formato da entrada.

Alguns detalhes de implementação

Suponha que a entrada (arquivo texto ou teclado) tenha a seguinte forma:

5 7 0 /* respectivamente, $|V|$, $|E|$, 0/1 (direcionado ou não) */
0 1
0 2
1 2
1 3
2 3
2 4
3 4

Alguns detalhes de implementação

```
leia n, m e b
enquanto houver arestas faça
    leia a próxima aresta (u, v)
    insira v na lista Adj[u]
    se b = FALSE então
        insira u na lista Adj[v]
```

Exercício. (extremamente fácil) Suponha que a entrada tenha esta mesma forma. Escreva um pseudocódigo para construir a matriz de adjacências a partir dela.

Calculando o quadrado

O *quadrado* de um grafo direcionado $G = (V, E)$ é o grafo $G^2 = (V, E^2)$ onde $(u, v) \in E^2$ se existe um caminho de comprimento no máximo 2 de u a v em G.

(1) Dado $G = (V, E)$ representado por uma matriz de adjacência, mostre como calcular G^2 (i.e. sua matriz de adjacência).

(2) Dado $G = (V, E)$ representado por listas de adjacência, mostre como calcular G^2 (i.e. suas listas de adjacência).

Calculando G^2 a partir da matriz de adjacência

Suponha que M é a matriz de adjacência de $G = (V, E)$.

```
QUADRADO(M)
 1 N ← M
 2 para cada u ∈ V faça
 3    para cada v ∈ V faça
 4       para cada w ∈ V faça
 5          se N[u, v] = 0 então
 6             N[u, v] ← M[u, w] · M[w, v]
 7 devolva N
```

Complexidade: $\Theta(V^3)$

Calculando G^2 a partir da lista de adjacências

Suponha que Adj guarda as listas de adjacências de $G = (V, E)$.

```
QUADRADO(Adj)
 1 Crie uma cópia Adj' de Adj
 2 para cada v ∈ V faça
 3     para cada u ∈ Adj[v] faça
 4         concatene uma cópia de Adj[u] a Adj'[v]
 5 para cada v ∈ V faça
 6     ordene Adj'[v] em tempo linear (Counting-Sort)
 7     elimine as repetições de Adj'[v] em tempo linear
 8 devolva Adj'
```

Complexidade: $\Theta(VE)$

Busca em Largura

Busca em grafos

- Grafos são estruturas mais complicadas do que listas, vetores e árvores (binárias).
  Precisamos de métodos para explorar/percorrer um grafo (direcionado ou não direcionado).
- 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 (direcionado 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 $O(V^2)$.

Representação de árvores

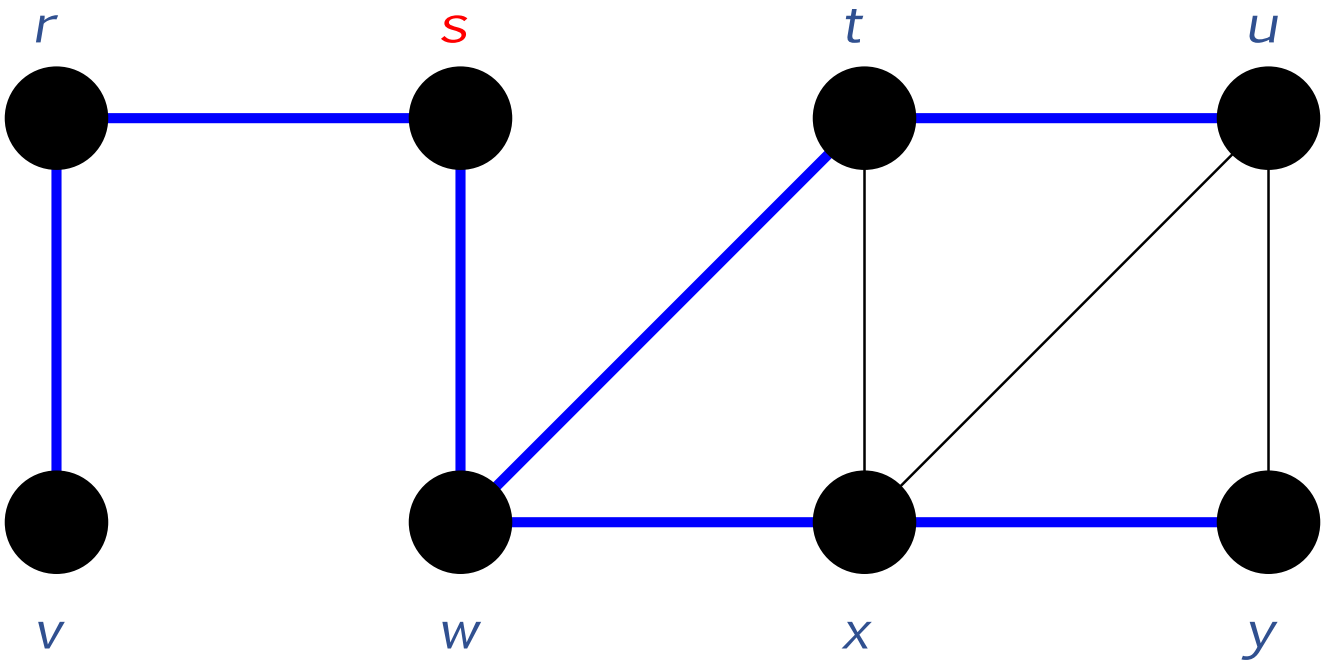
- Os algoritmos de busca que veremos constroem uma árvore (ou floresta) que é um subgrafo do grafo de entrada.
- Usualmente supomos que há um vértice s que será a raiz da árvore.
- É conveniente ter uma representação desta árvore que facilite certas operações. Por exemplo, determinar o caminho que vai da raiz s a um vértice v.

Representação de árvores

- Para representar uma árvore com raiz s usamos um vetor $\pi[\,]$.
- Convencionamos que $\pi[s] = NIL$.
- Cada vértice $v (\neq s)$ possui um pai $\pi[v]$.
- O caminho de s a v na Árvore é dado por:

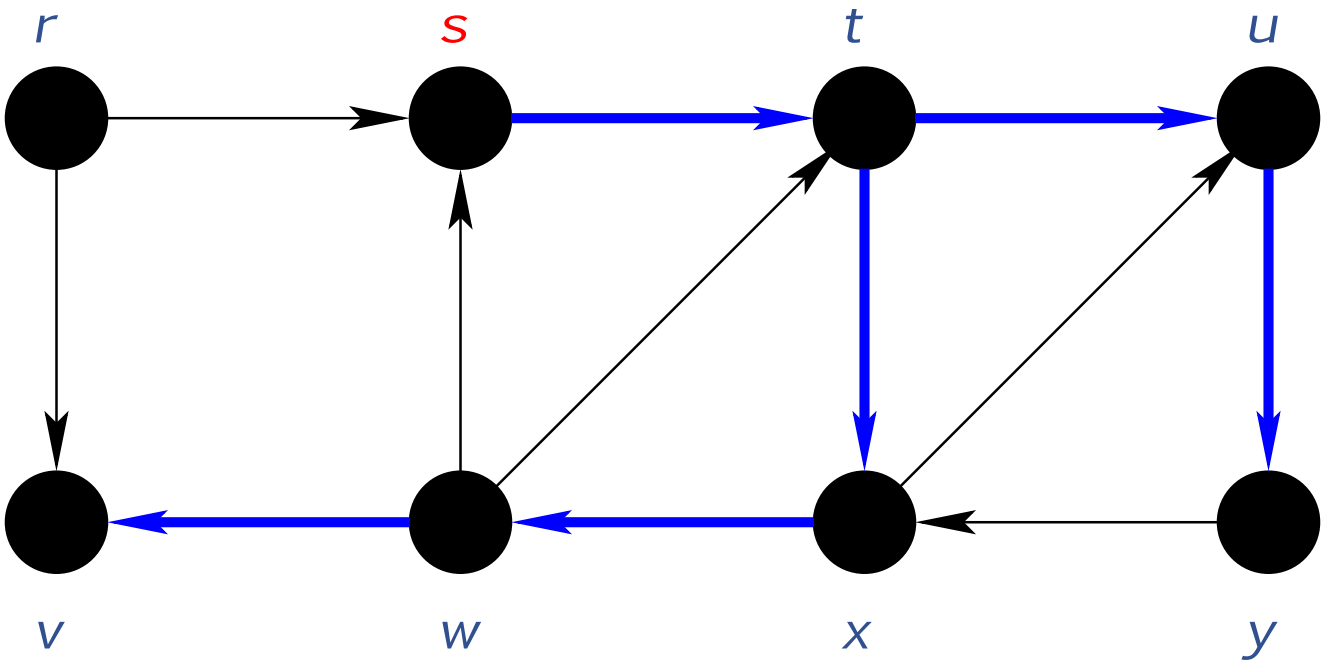
  $v, \pi[v], \pi[\pi[v]], \pi[\pi[\pi[v]]], \ldots, s.$

Exemplo 1



v	r	s	t	u	v	w	x	y
$\pi[v]$	s	N	w	t	r	s	w	x

Exemplo 2



v	r	s	t	u	v	w	x	y
$\pi[v]$	N	N	s	t	w	x	t	u

Como obter os caminhos na árvore

Imprime o caminho de s a v na árvore com raiz s representada por $\pi[\,]$.

```
PRINT-PATH(G, s, v)
1   se v = s então
2       imprima s
3   senão
4       se π[v] = NIL então
4           imprima Não existe caminho de s a v.
5       senão
6           PRINT-PATH(G, s, π[v])
7           imprima v.
```

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

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.
  Denotamos este valor por $\mathrm{dist}(s, v)$.
- Se v **não é alcançável** a partir de s, então $\mathrm{dist}(s, v) = \infty$ (*distância 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 à 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.

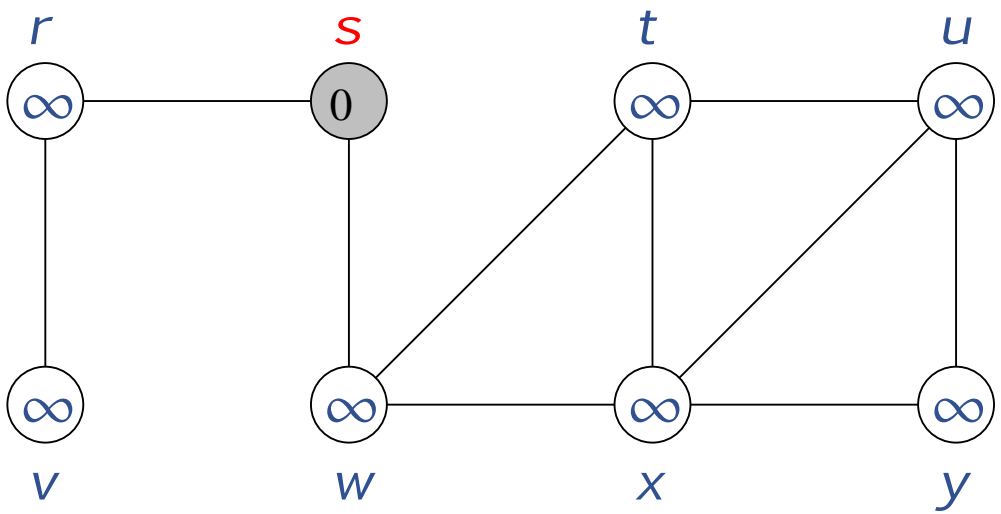
Busca em largura

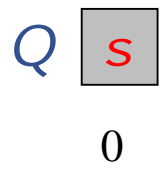
- Inicialmente a Árvore de Busca em Largura contém apenas o vértice fonte s.
- Para cada vizinho v de s, o vértice v e a aresta (s, v) são acrescentadas à árvore.
- O processo é repetido para os vizinhos dos vizinhos de s e assim por diante, até que todos os vértices atingíveis por s sejam inseridos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

Busca em largura

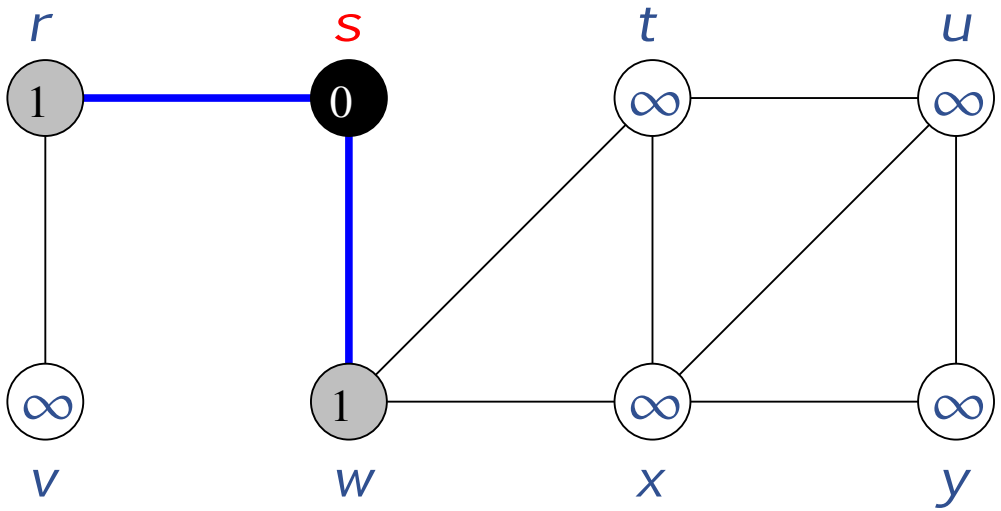
- Busca em largura atribui cores a cada vértice: branco, cinza e preto.
- Cor branca = "não visitado".
  Inicialmente todos os vértices são brancos.
- Cor cinza = "visitado pela primeira vez" (na fila).
- Cor Preta = "teve seus vizinhos visitados".

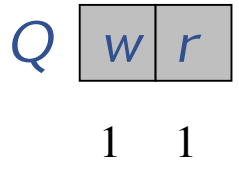
Exemplo (CLRS)



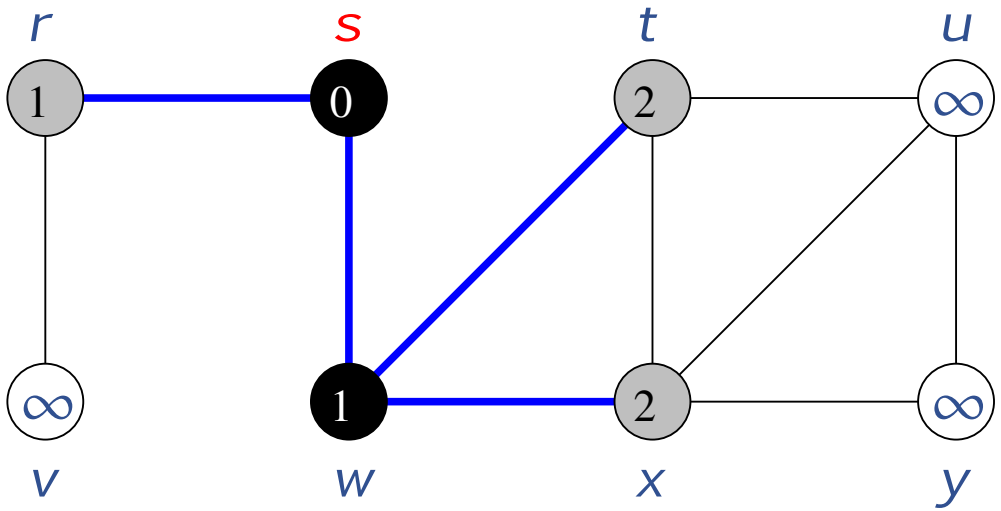


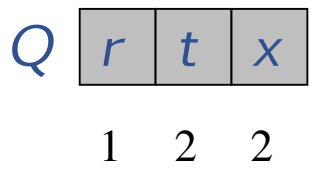
Exemplo (CLRS)



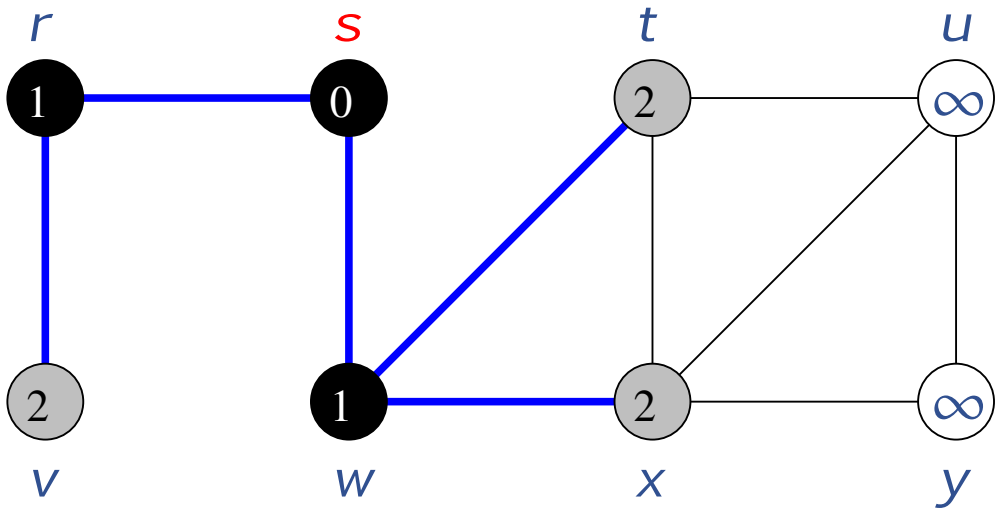


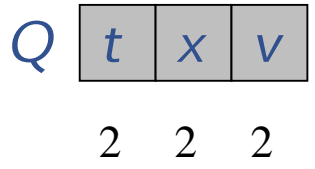
Exemplo (CLRS)



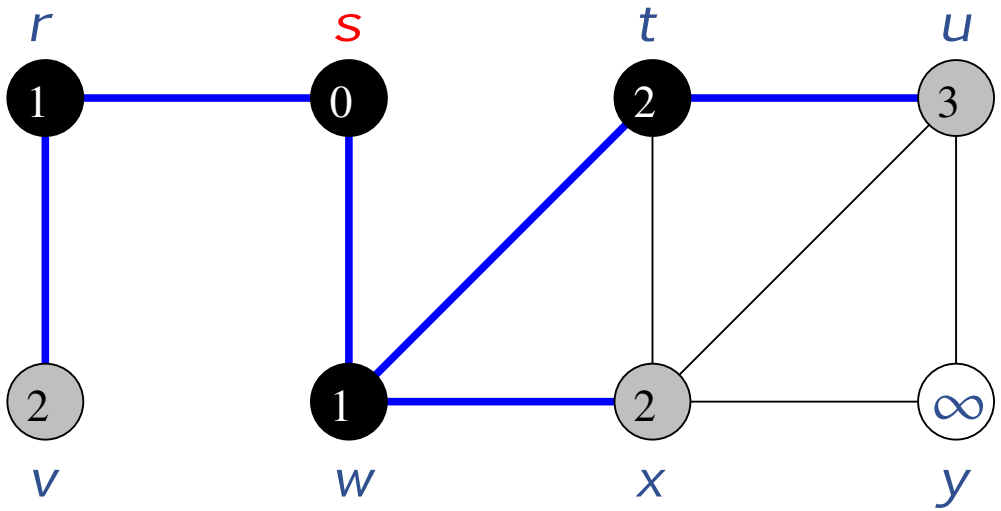


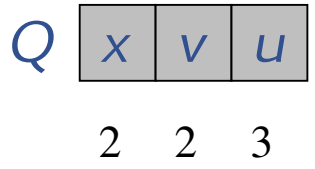
Exemplo (CLRS)



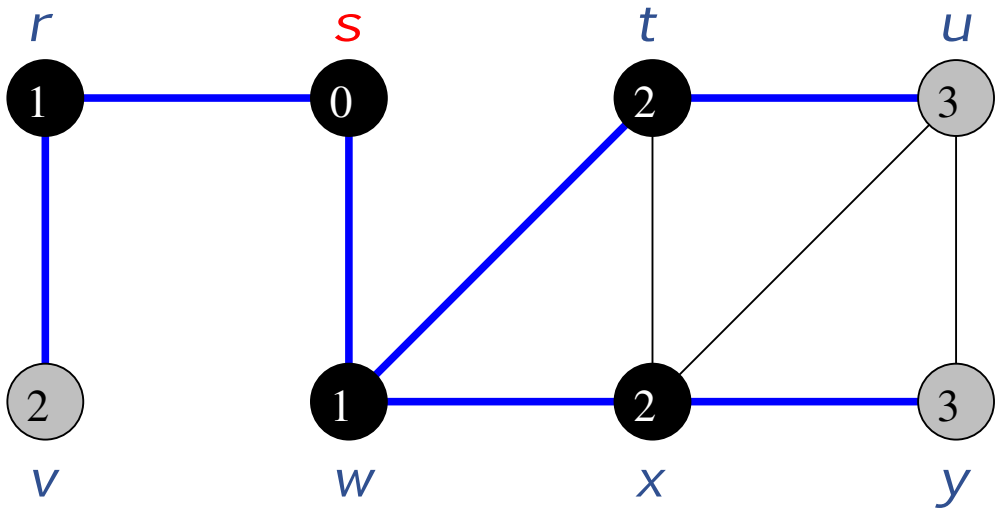


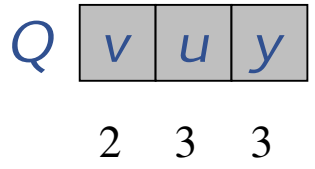
Exemplo (CLRS)



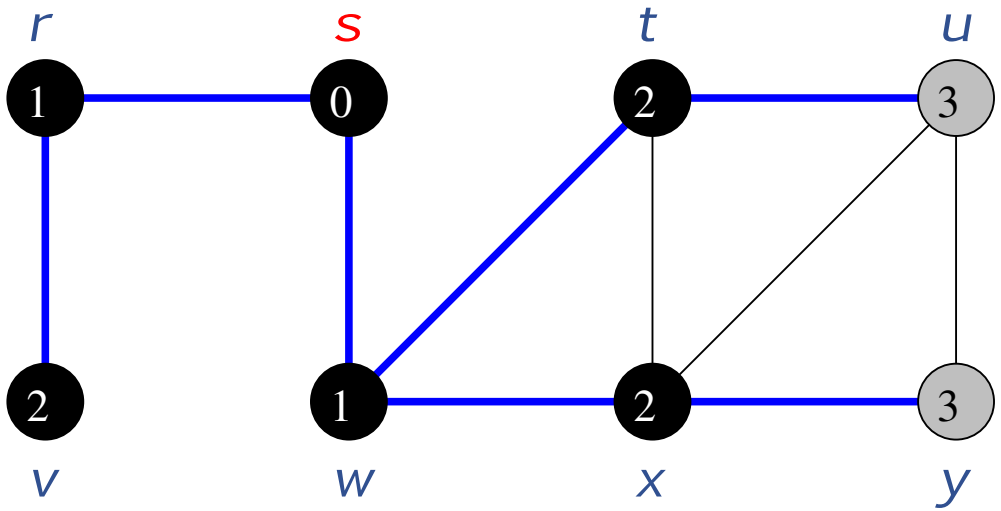


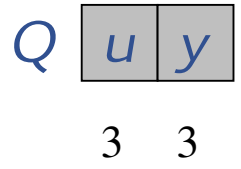
Exemplo (CLRS)



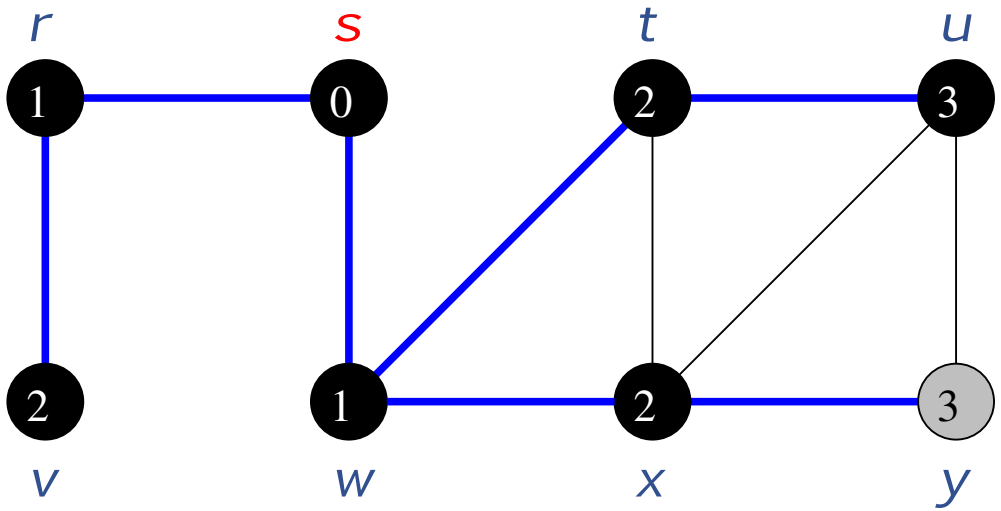


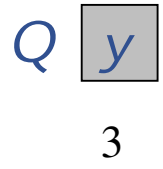
Exemplo (CLRS)



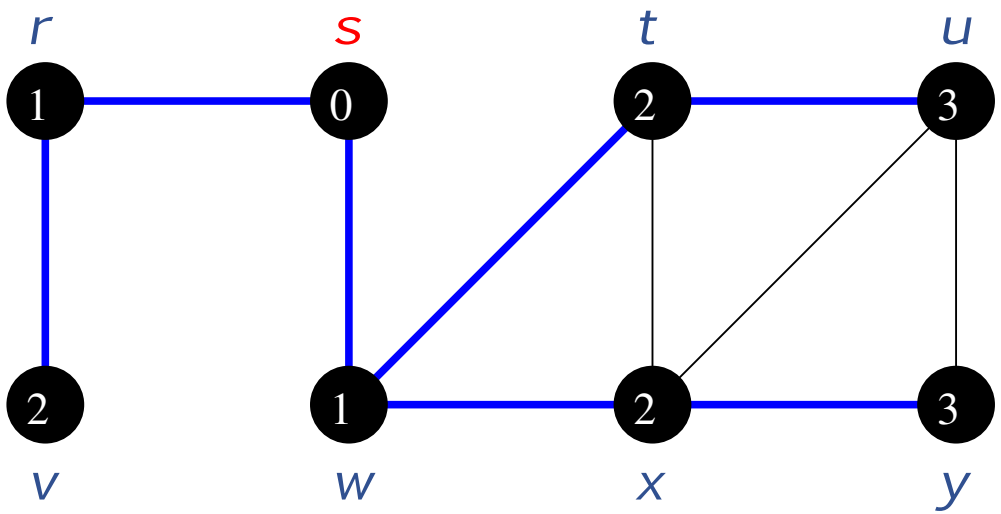


Exemplo (CLRS)





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.
- Representamos a árvore através de um vetor $\pi[\,]$.
- 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.

```
BFS(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 BFS é $O(V + E)$.

Agora falta mostrar que BFS funciona.

Correção

Lembre-se que $\mathrm{dist}(s, v)$ a distância de s a v.

Precisamos mostrar que:

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

Alguns lemas

Note que $d[v]$ e $\pi[v]$ nunca mudam após v ser inserido na fila.

Lema 1. Se $d[v] < \infty$ então v pertence à árvore T induzida por $\pi[]$ e o caminho de s a v em T tem comprimento $d[v]$.

Prova:

Indução no número de operações ENQUEUE.

Alguns lemas

Base: quando s é inserido na fila temos $d[s] = 0$ e s é a raiz da árvore T.

Passo de indução: v é descoberto enquanto a busca é feita em u (percorrendo $\mathrm{Adj}[u]$). Então por HI existe um caminho de s a u em T com comprimento $d[u]$.

Como tomamos $d[v] = d[u] + 1$ e $\pi[v] = u$, o resultado segue. □

Alguns lemas

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

Corolário 1. Durante a execução do algoritmo vale o seguinte invariante

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

□

Alguns lemas

Lema 2. 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 2

Prova: Indução no número de operações ENQUEUE e DEQUEUE.

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

Passo de indução:

Caso 1: 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.

Prova do Lema 2

Passo de indução:

Caso 2: $v = v_{r+1}$ 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 segue que $d[v_r] \leq d[u] + 1$. Logo

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

As outras desigualdades se mantêm. □

Árvore

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

Então após a execução de BFS,

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

e

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

Prova: Basta provar que $d[v] = \mathrm{dist}(s, v)$ para todo $v \in V[G]$.

Note que se $\mathrm{dist}(s, v) = \infty$ então $d[v] = \infty$ pelo Corolário 1.

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

Correção

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

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

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

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

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

Correção

- 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 2, $d[w] \leq d[u] = k - 1$ e segue que $d[v] = k$.

- se v é preto, então v já passou pela fila Q e pelo Lema 2, $d[v] \leq d[u] = k - 1$. Mas por outro lado, pelo Corolário 1, $d[v] \geq \mathrm{dist}(\mathrm{s}, \mathrm{v}) = \mathrm{k}$, o que é uma contradição. Este caso não ocorre.

Portanto, temos que $d[v] = \mathrm{dist}(\mathrm{s}, \mathrm{v})$. □

Busca em Profundidade

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 direcionados quanto não direcionados.
- 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 ideia 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 e volte ao primeiro passo.

Floresta de Busca em Profundidade

- A busca em profundidade associa a cada vértice x um pai $\pi[x]$.
- 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

À 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".

Observação: os vértices de cor cinza correspondem a um caminho na floresta (começando da raiz).

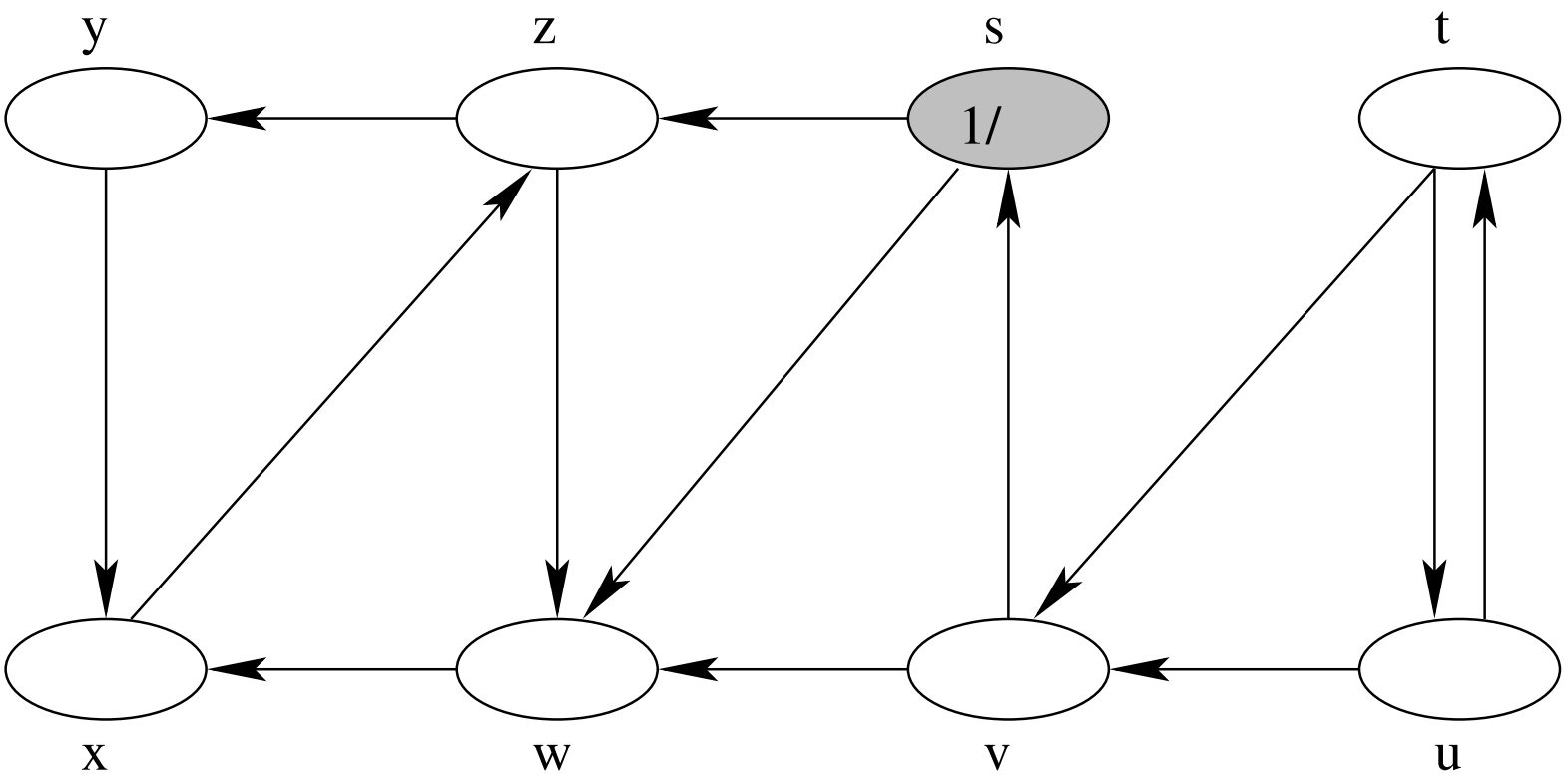
Estampas/rótulos

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

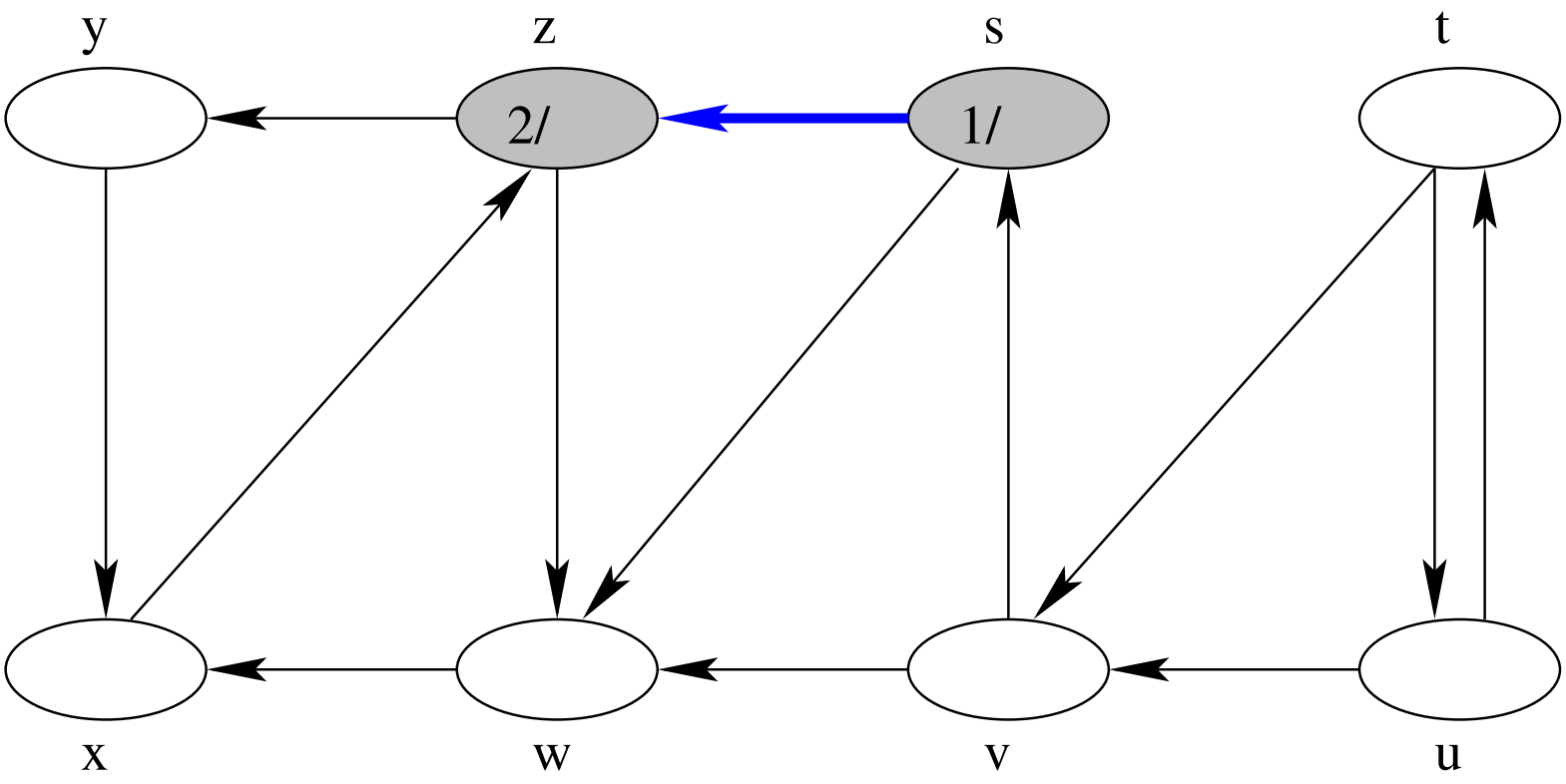
- $d[x]$: instante de descoberta de x.
  Neste instante x torna-se cinza.
- $f[x]$: instante de finalização de x.
  Neste instante x torna-se preto.

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

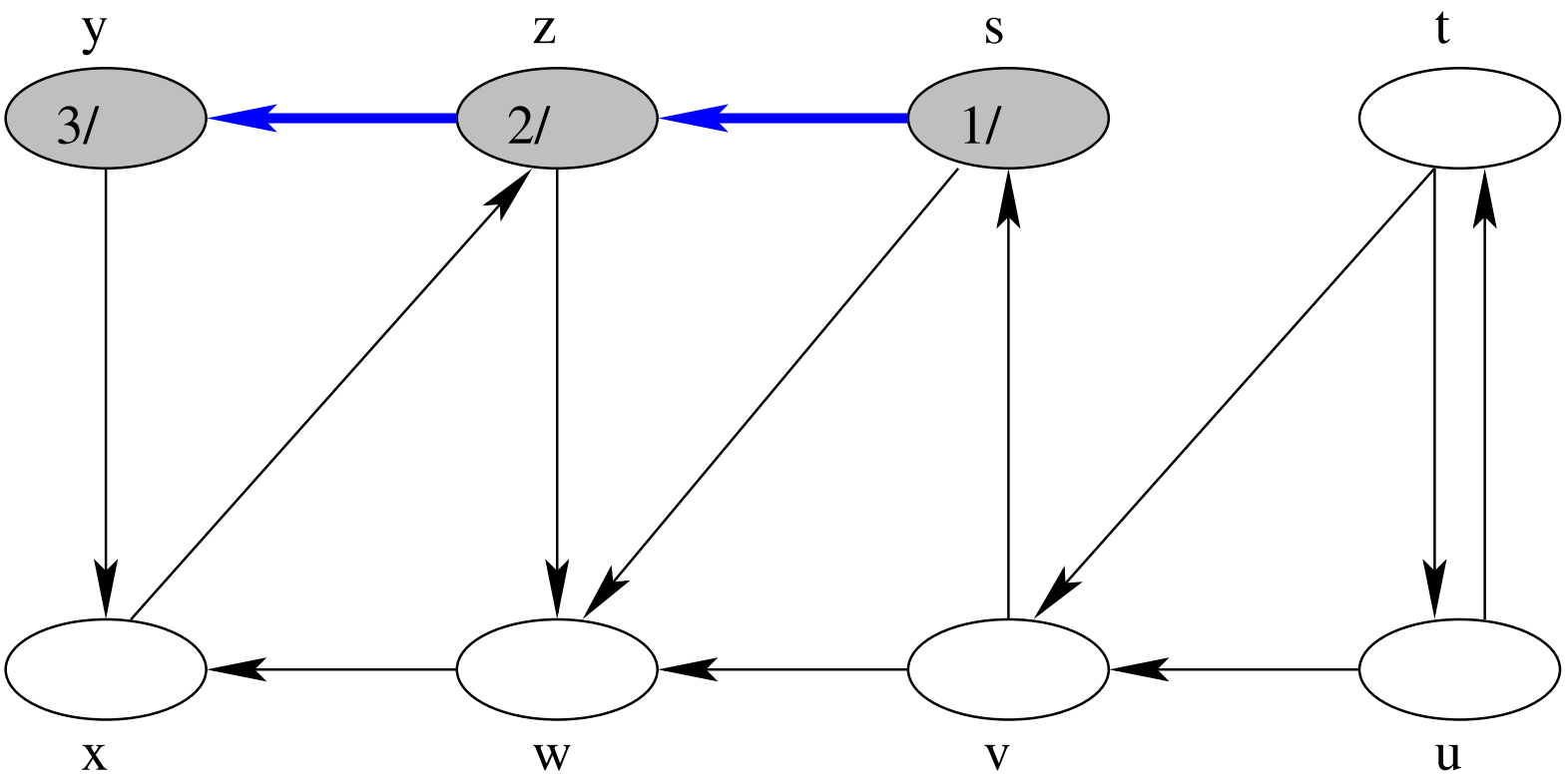
Exemplo DFS



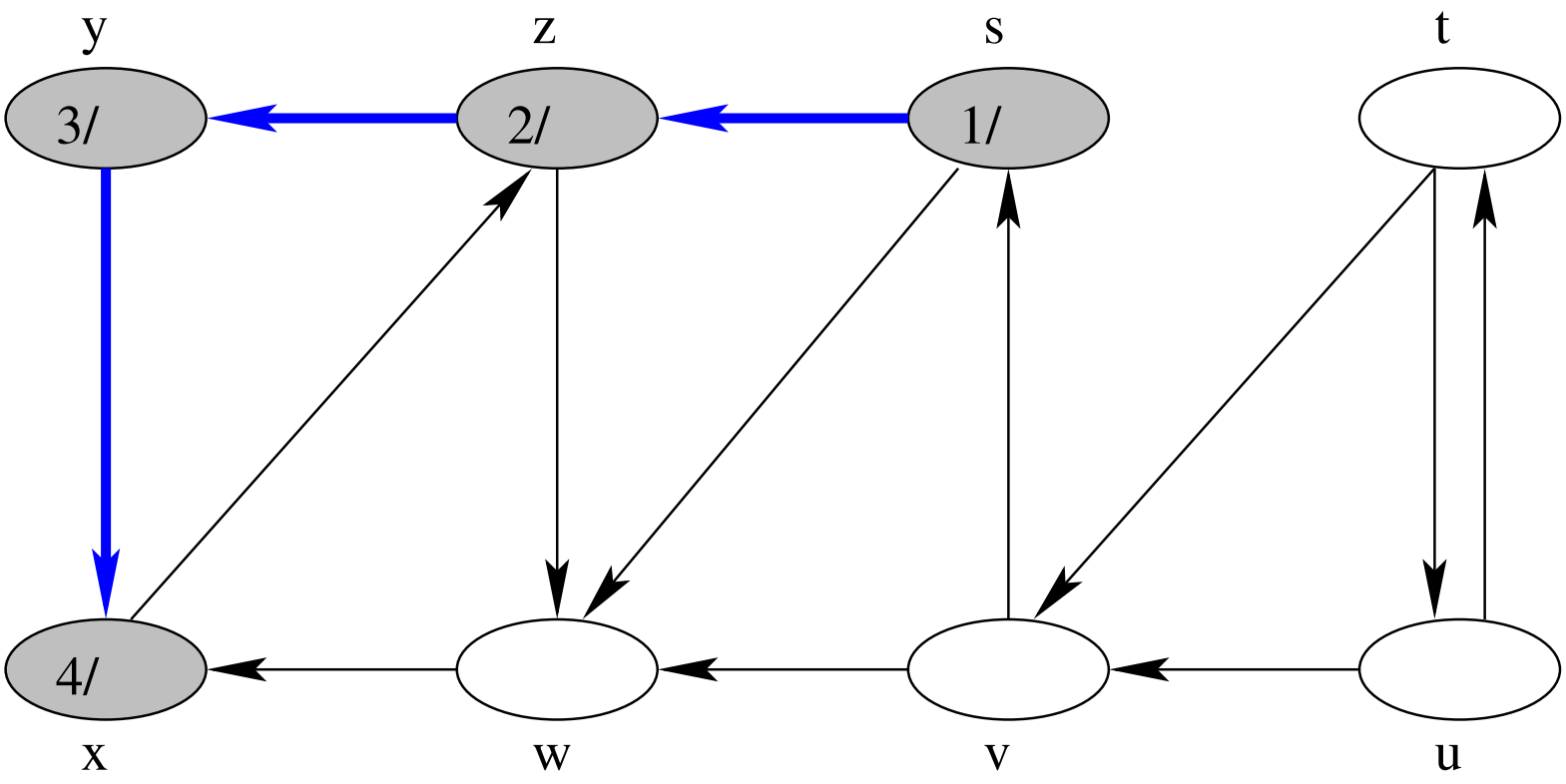
Exemplo DFS



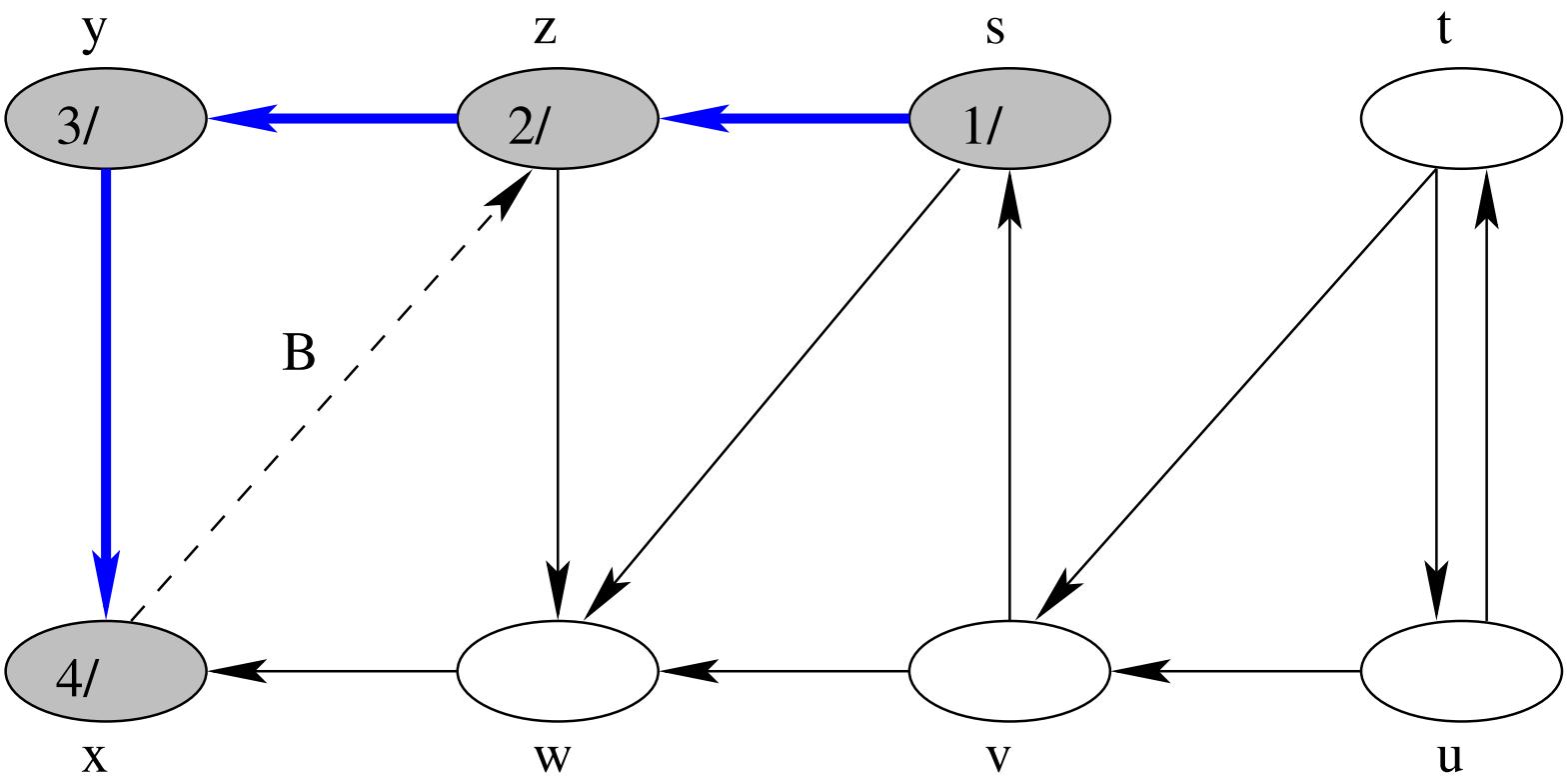
Exemplo DFS



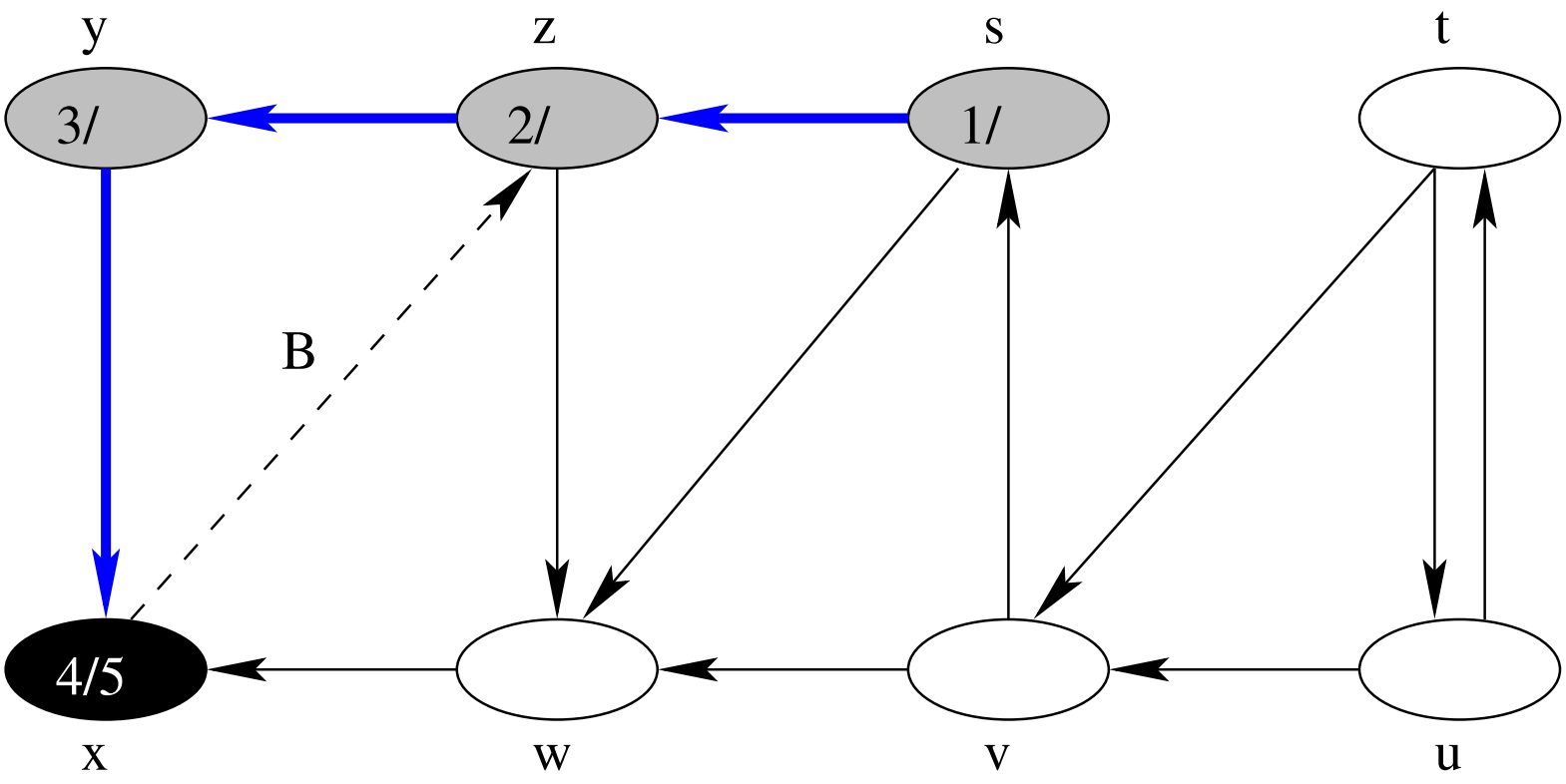
Exemplo DFS



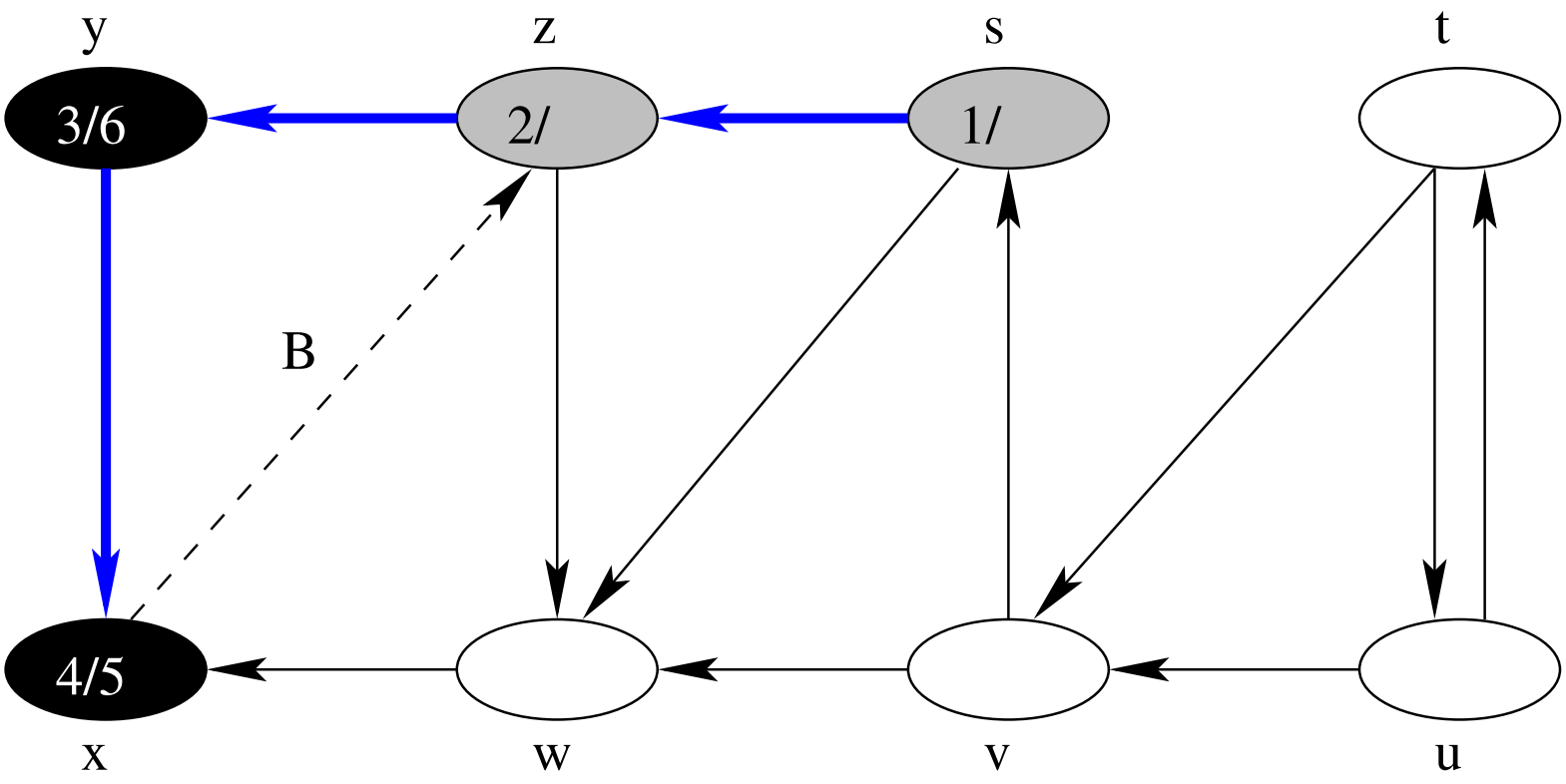
Exemplo DFS



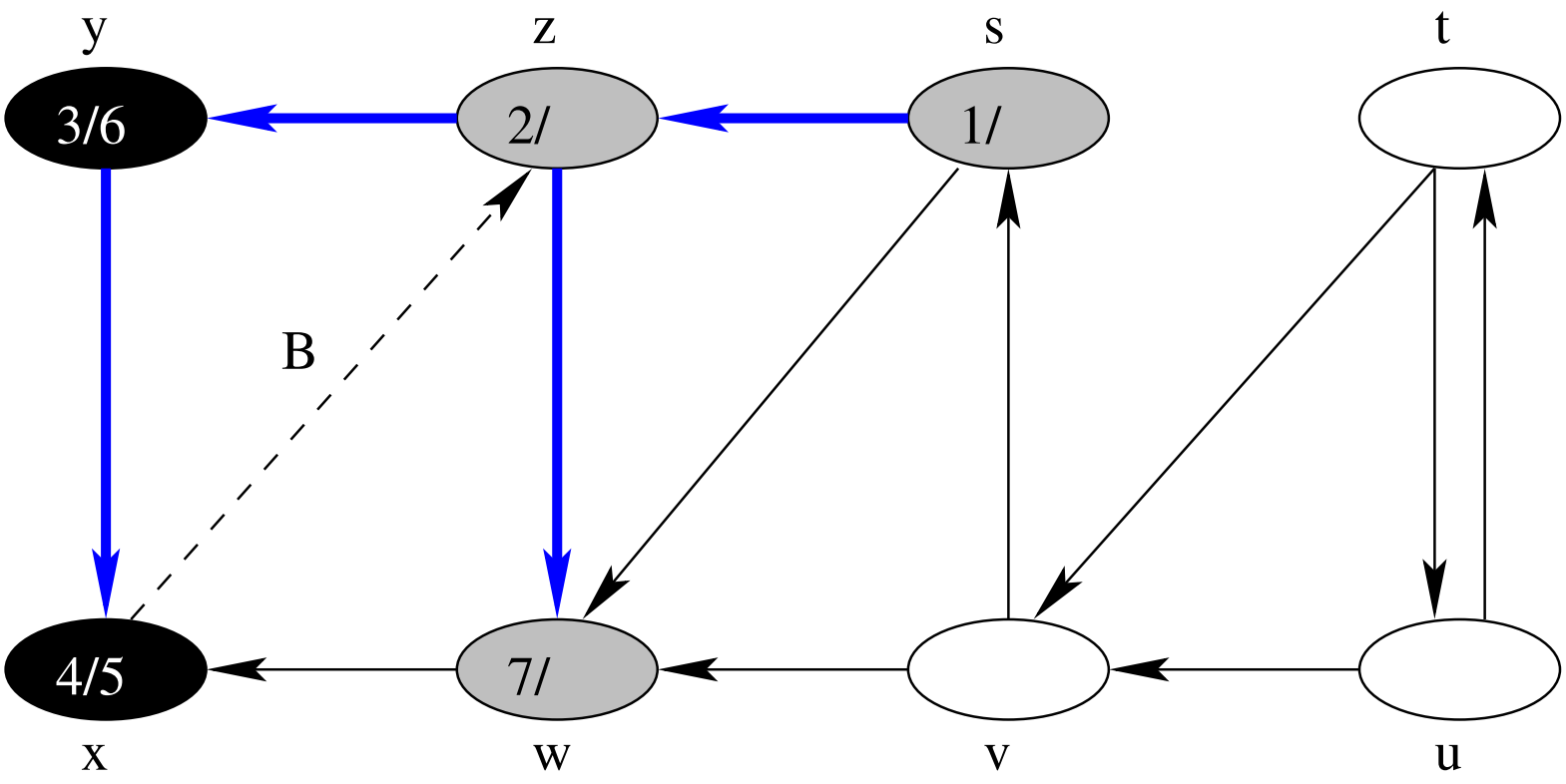
Exemplo DFS



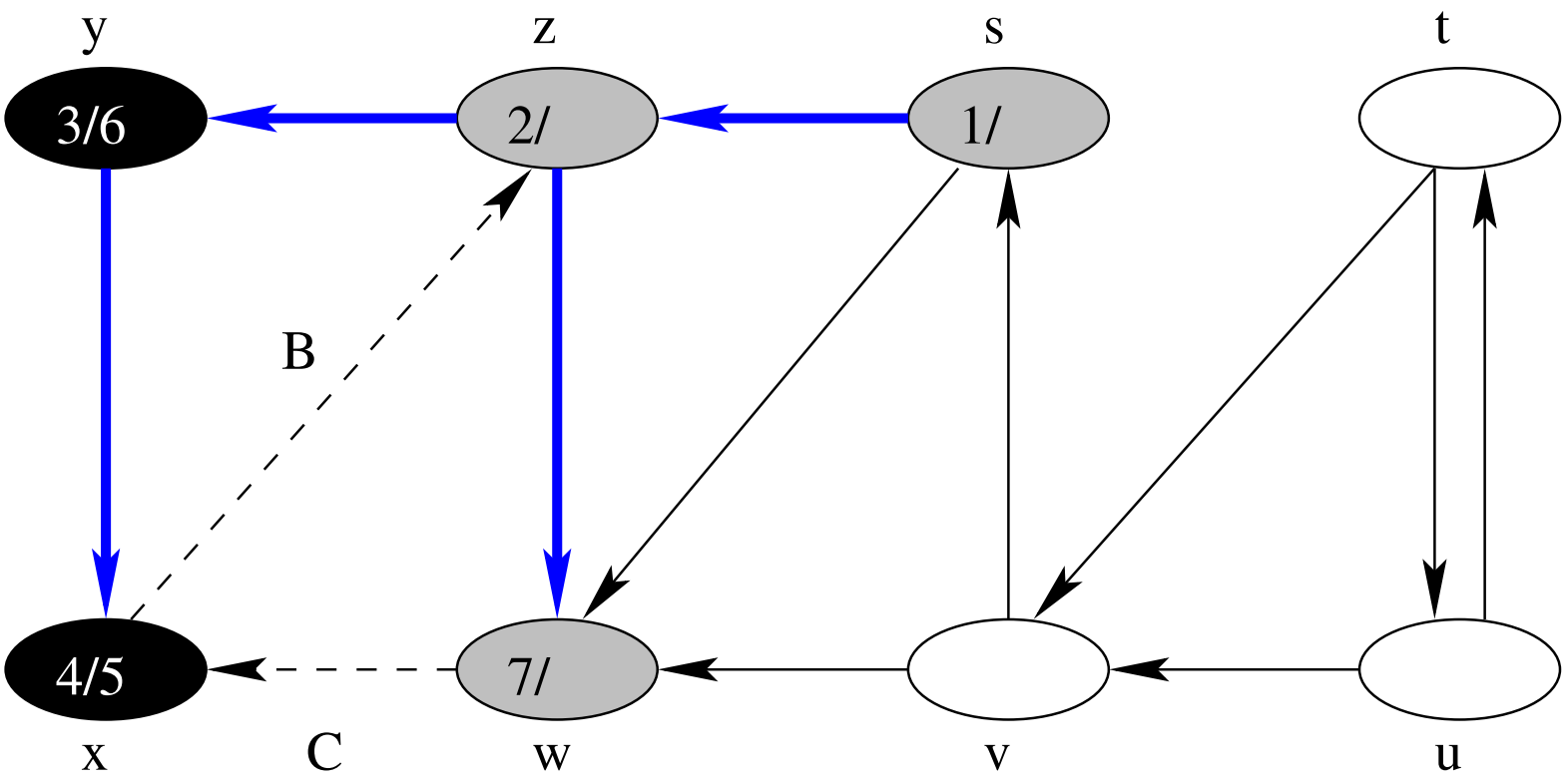
Exemplo DFS



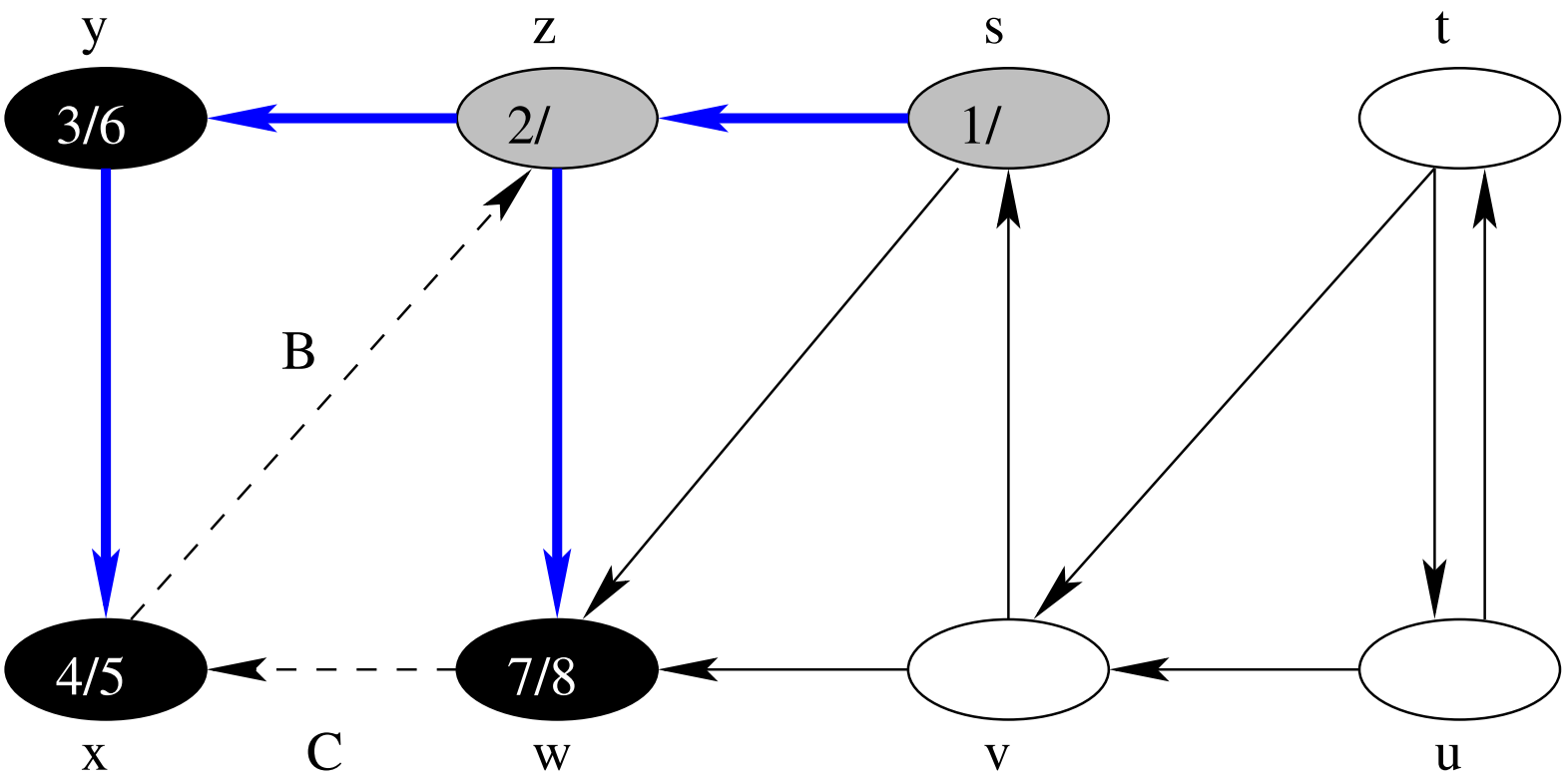
Exemplo DFS



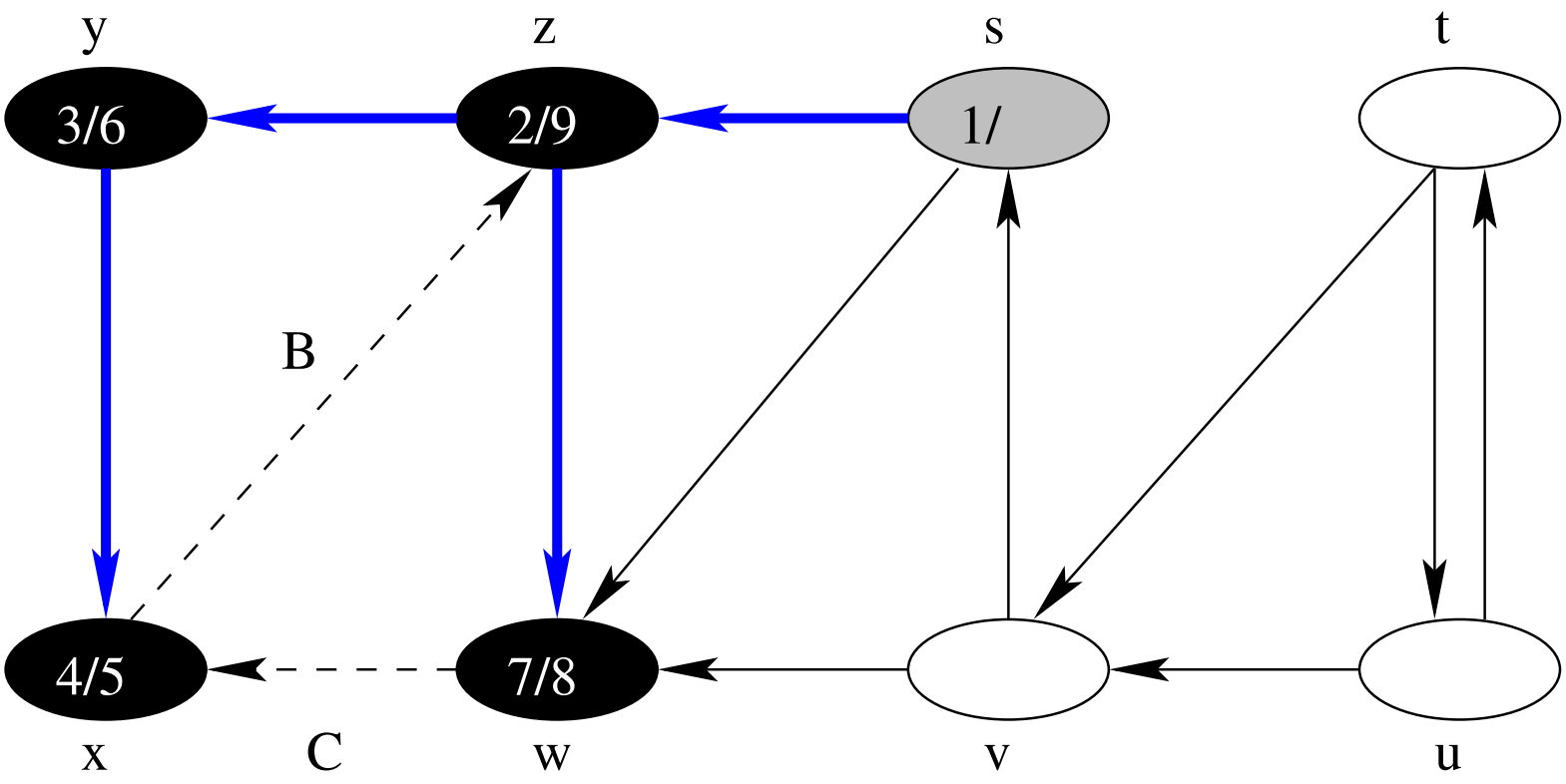
Exemplo DFS



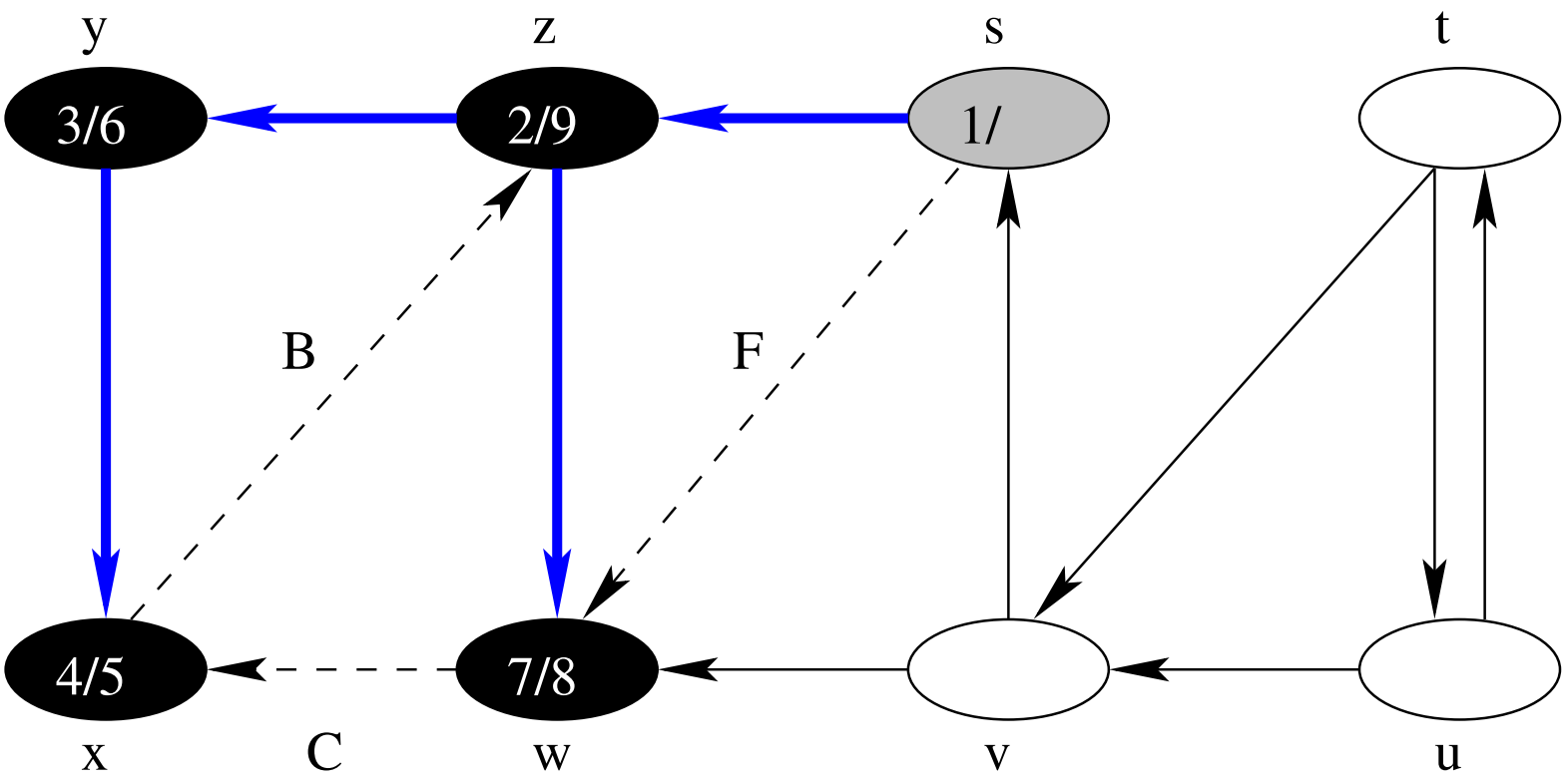
Exemplo DFS



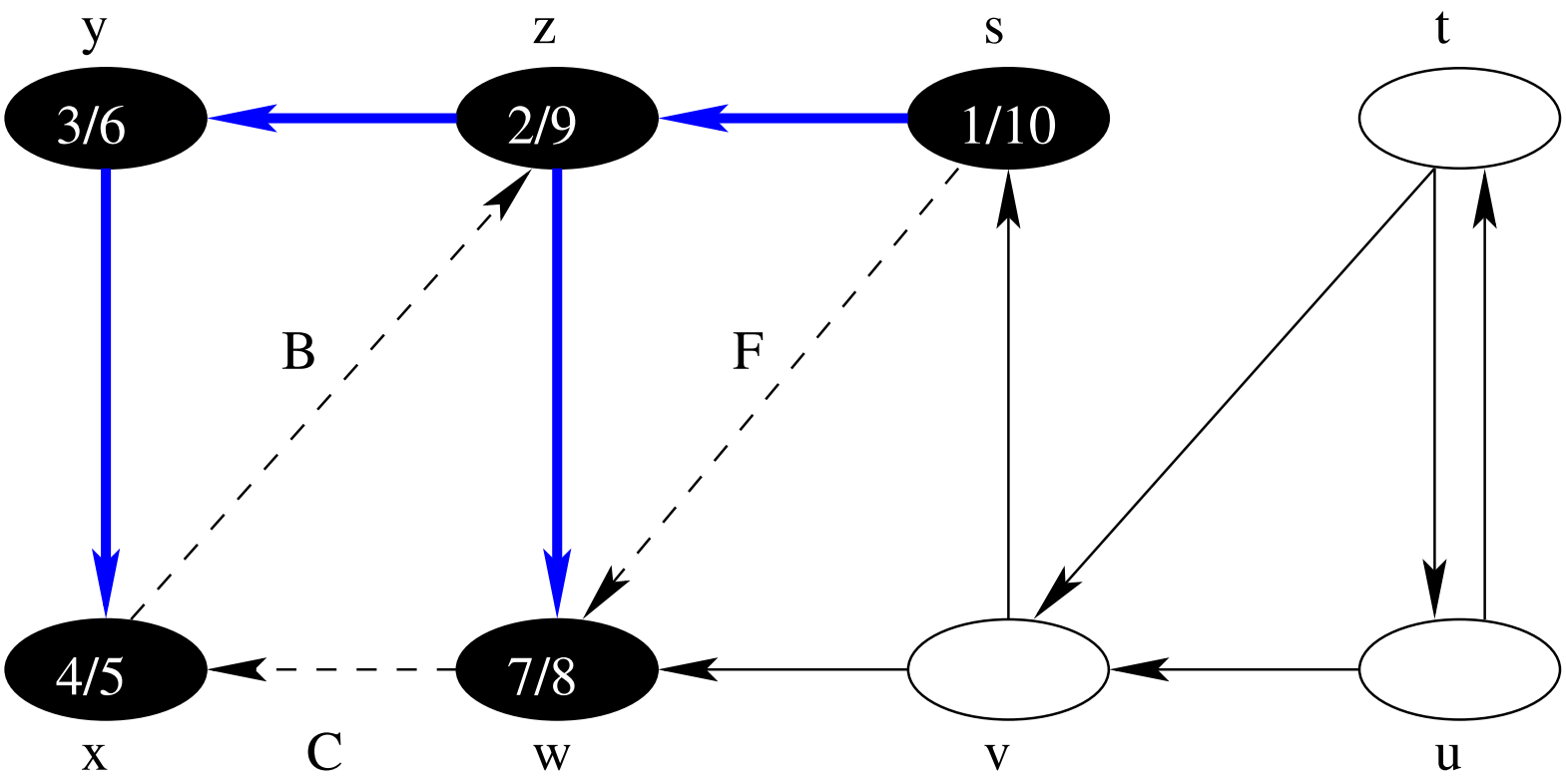
Exemplo DFS



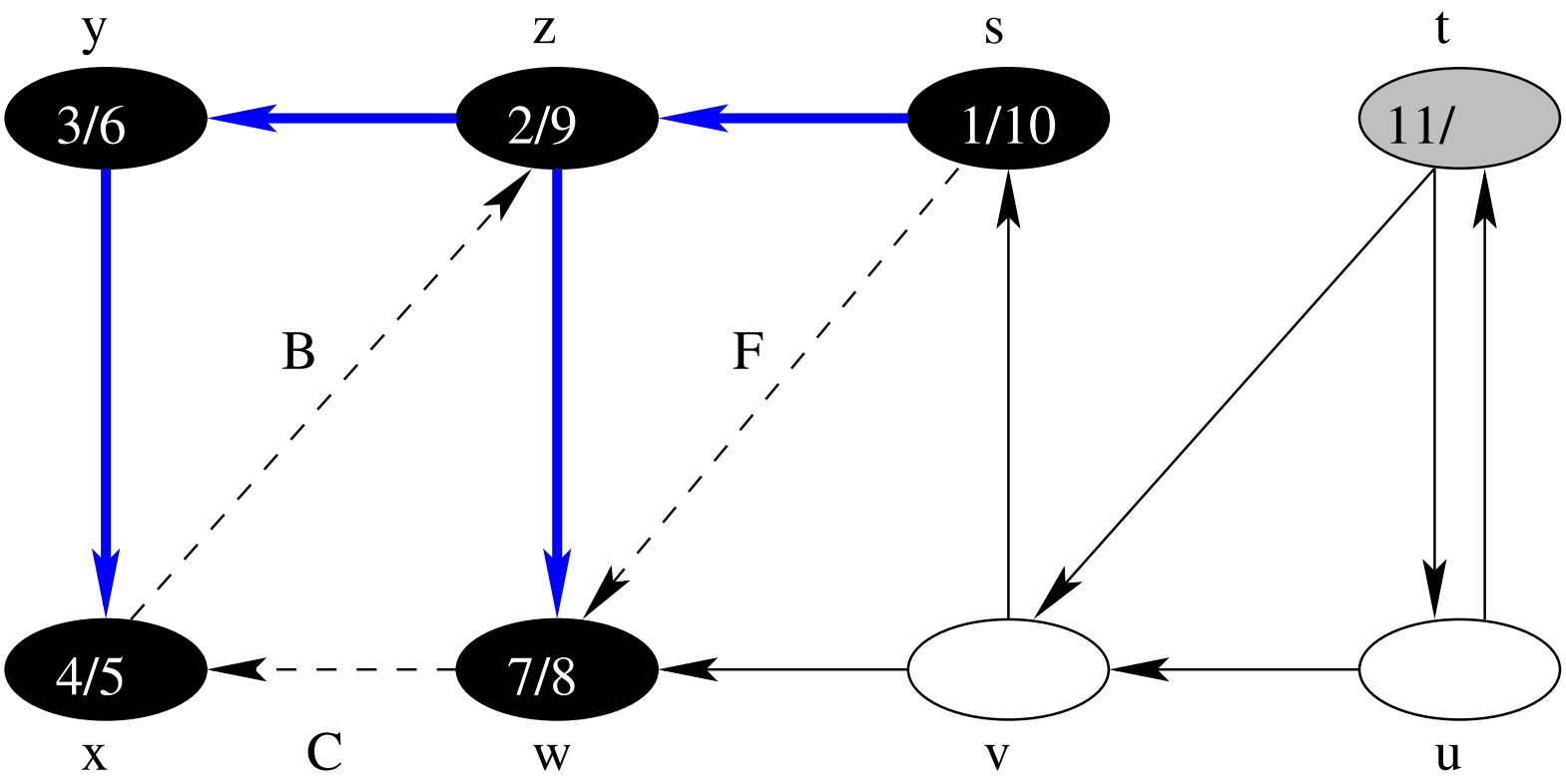
Exemplo DFS



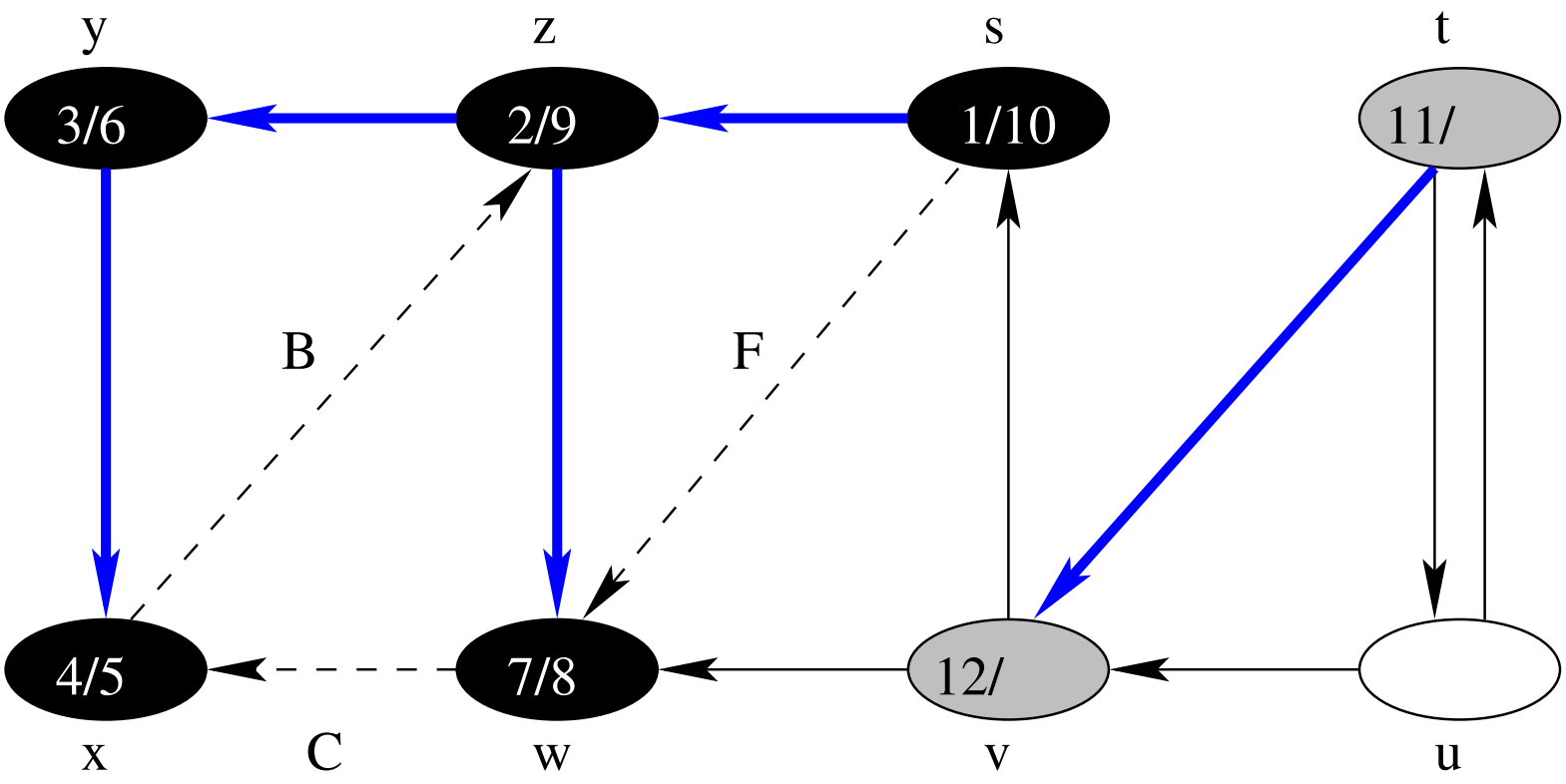
Exemplo DFS



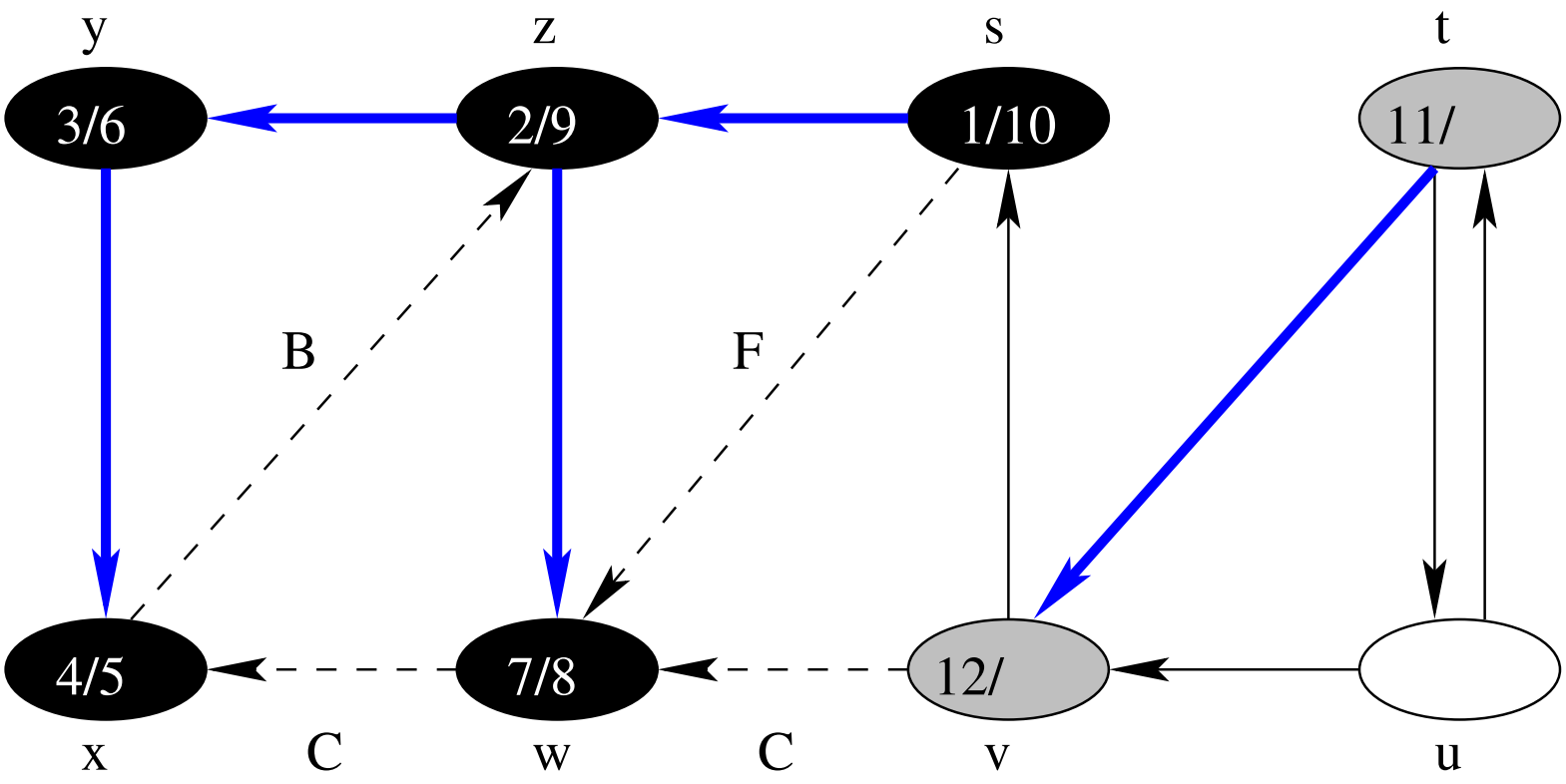
Exemplo DFS



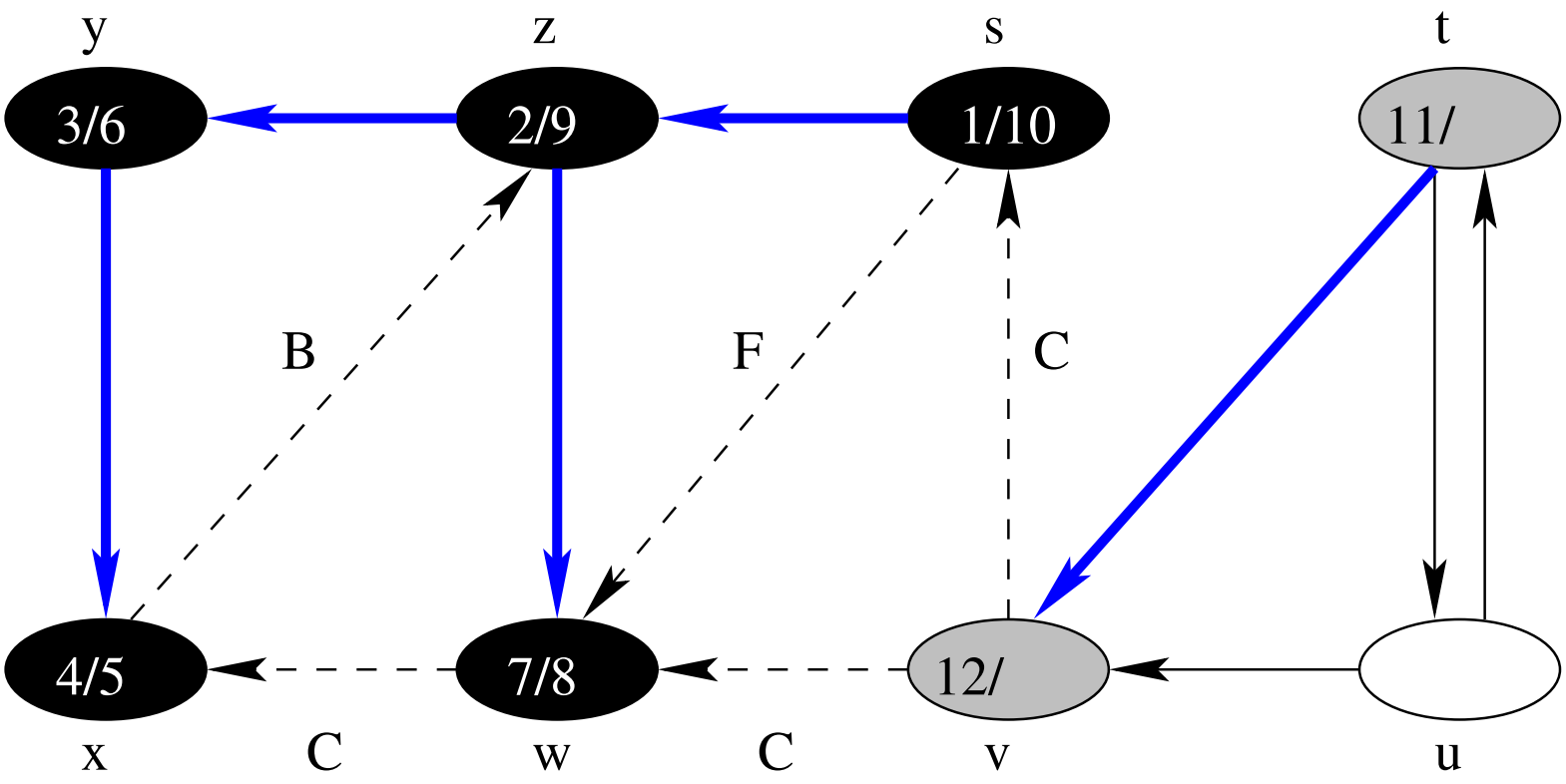
Exemplo DFS



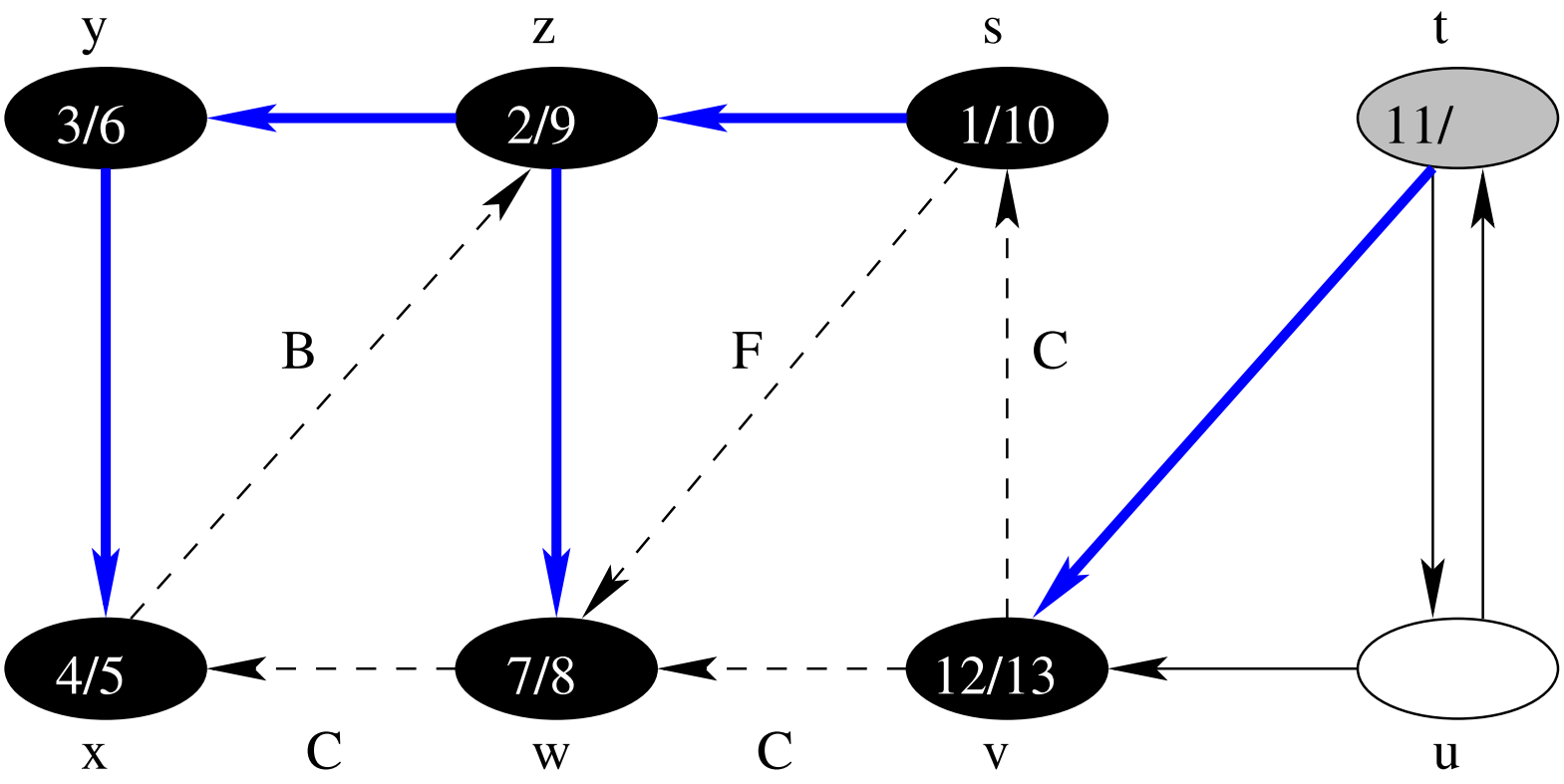
Exemplo DFS



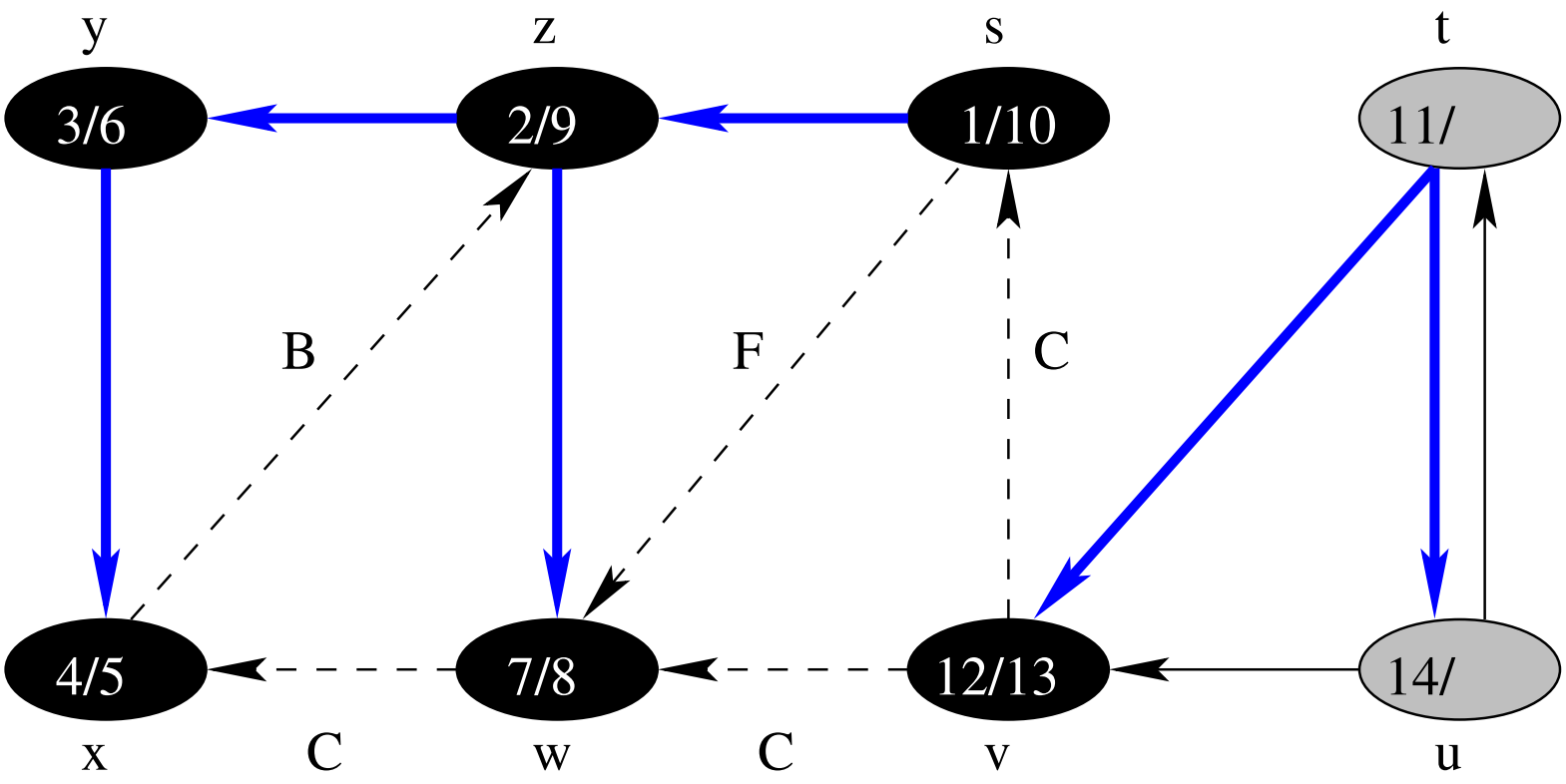
Exemplo DFS



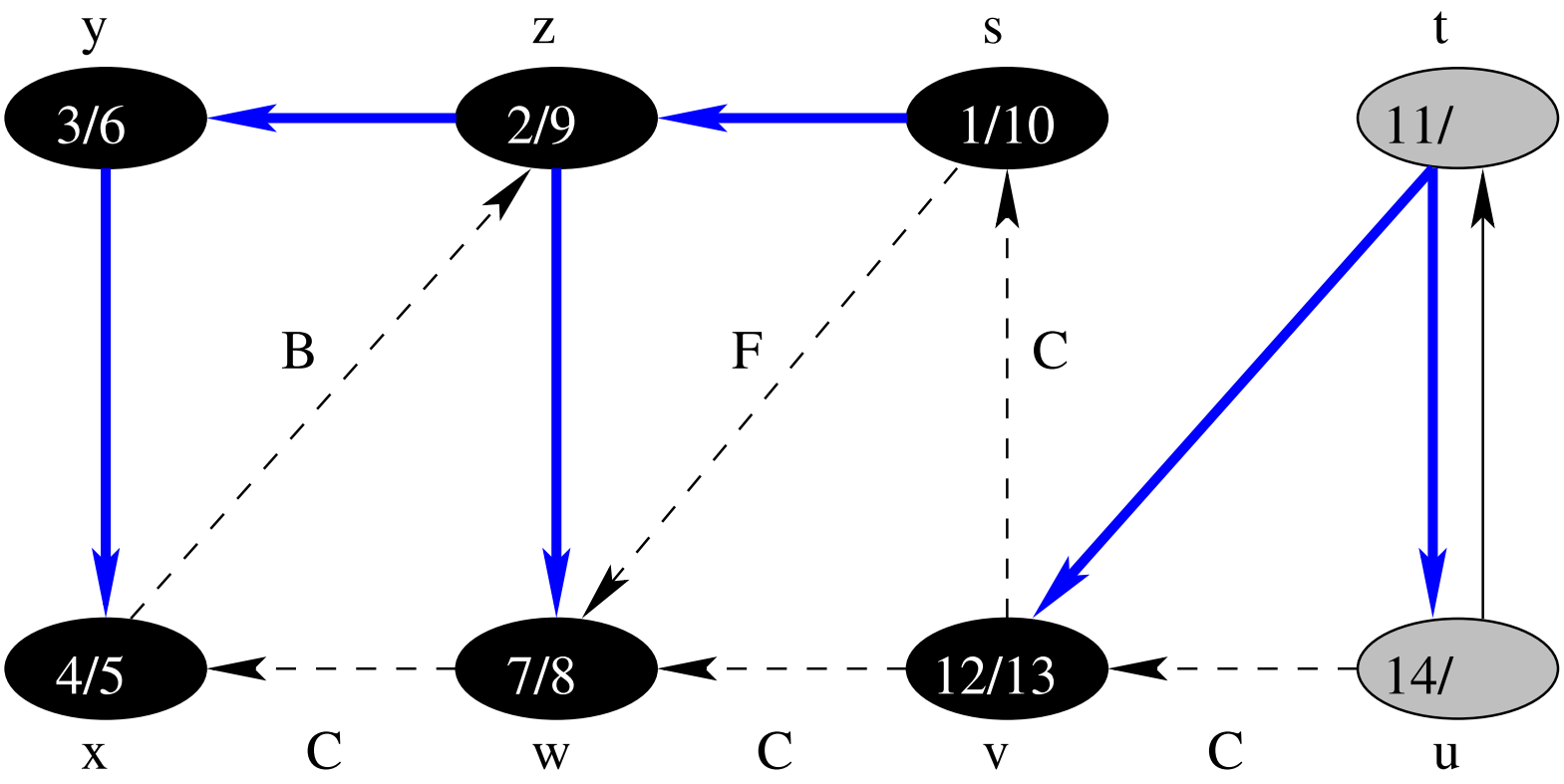
Exemplo DFS



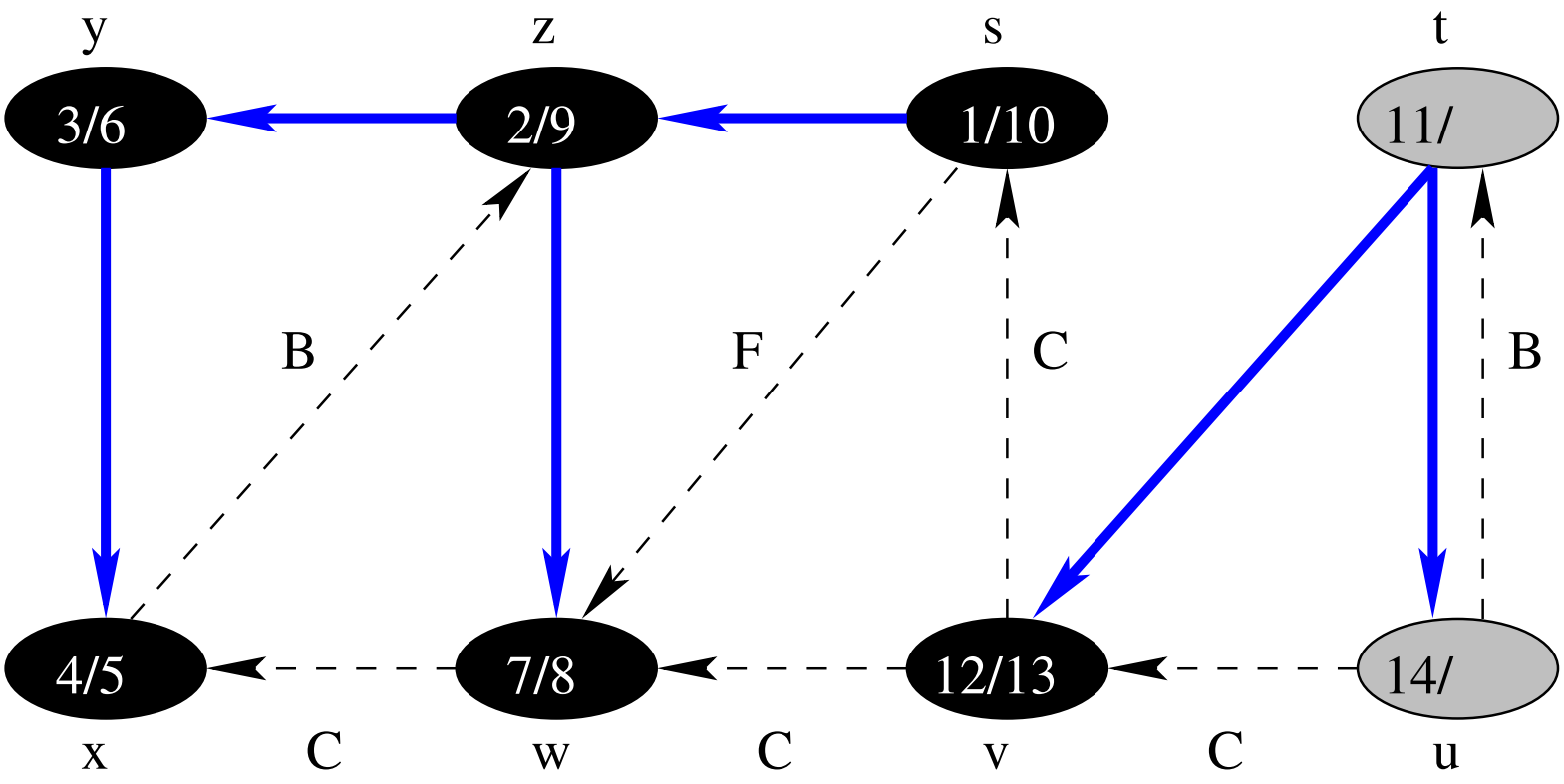
Exemplo DFS



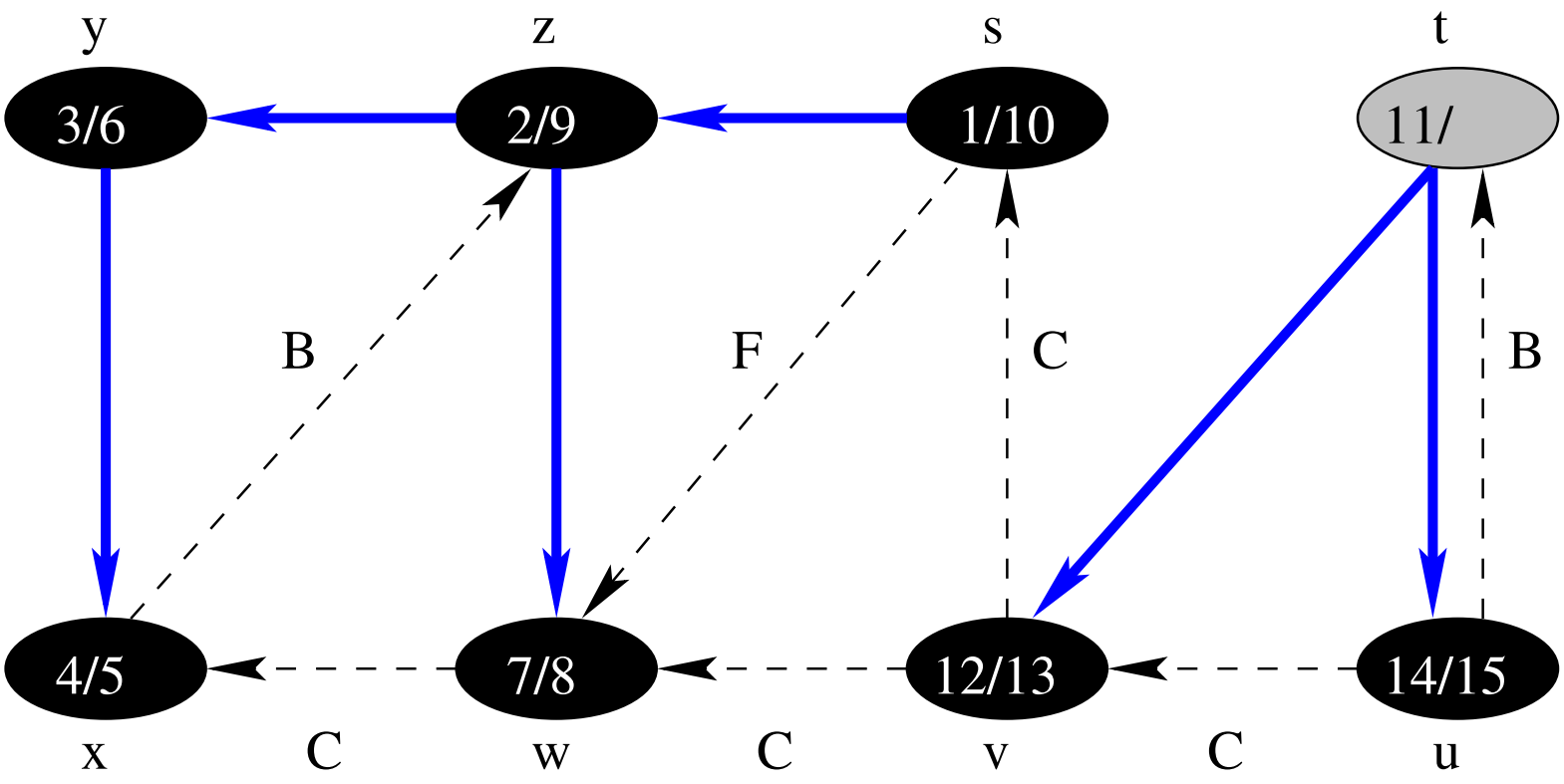
Exemplo DFS



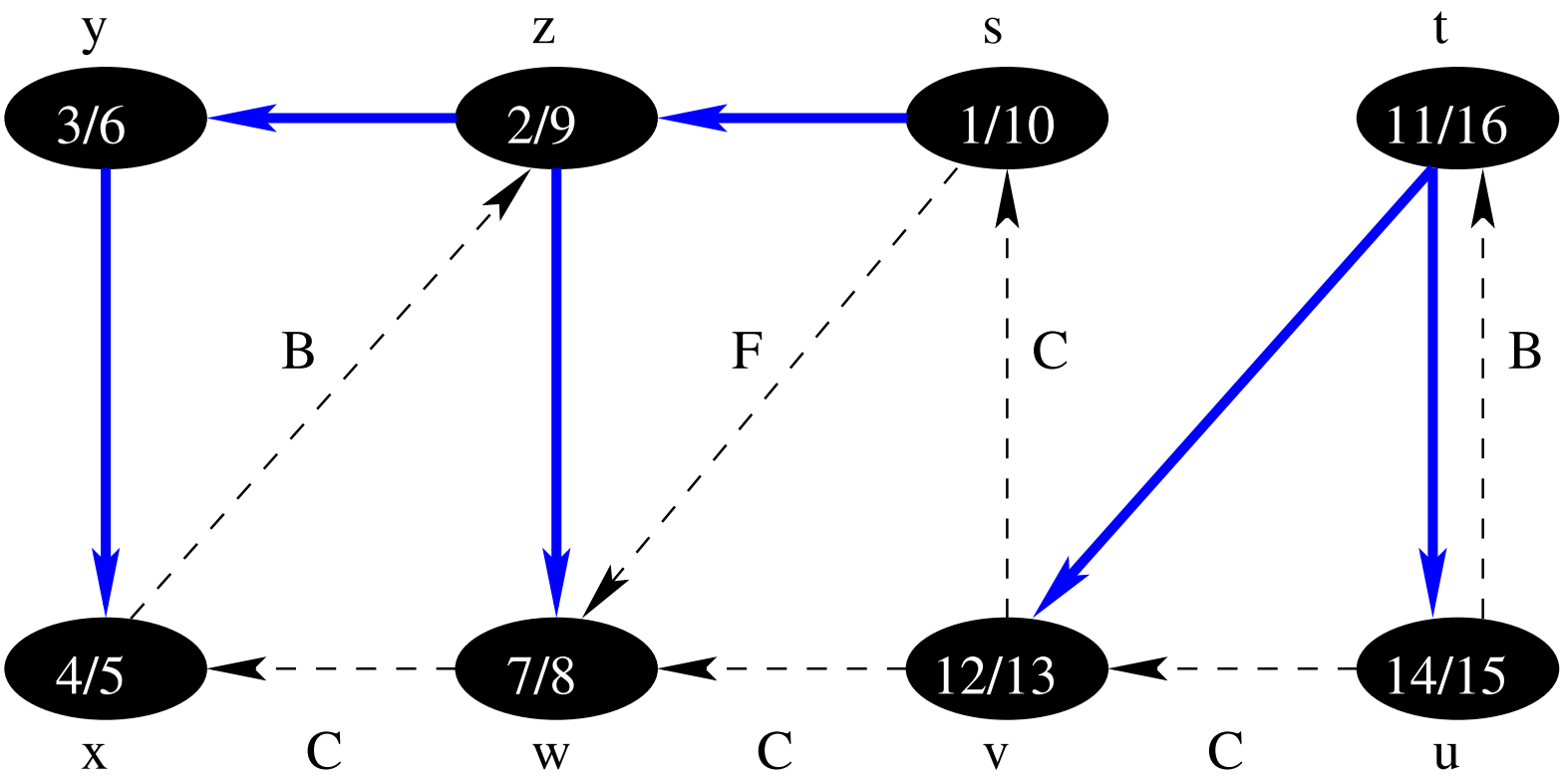
Exemplo DFS



Exemplo DFS



Exemplo DFS



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 $v \in V[G]$ 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   tempo ← tempo + 1
10   f[u] ← tempo
```

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   tempo ← tempo + 1
10   f[u] ← tempo
```

Consumo de tempo

linhas 4-7: executado $|\mathrm{Adj}[u]|$ 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

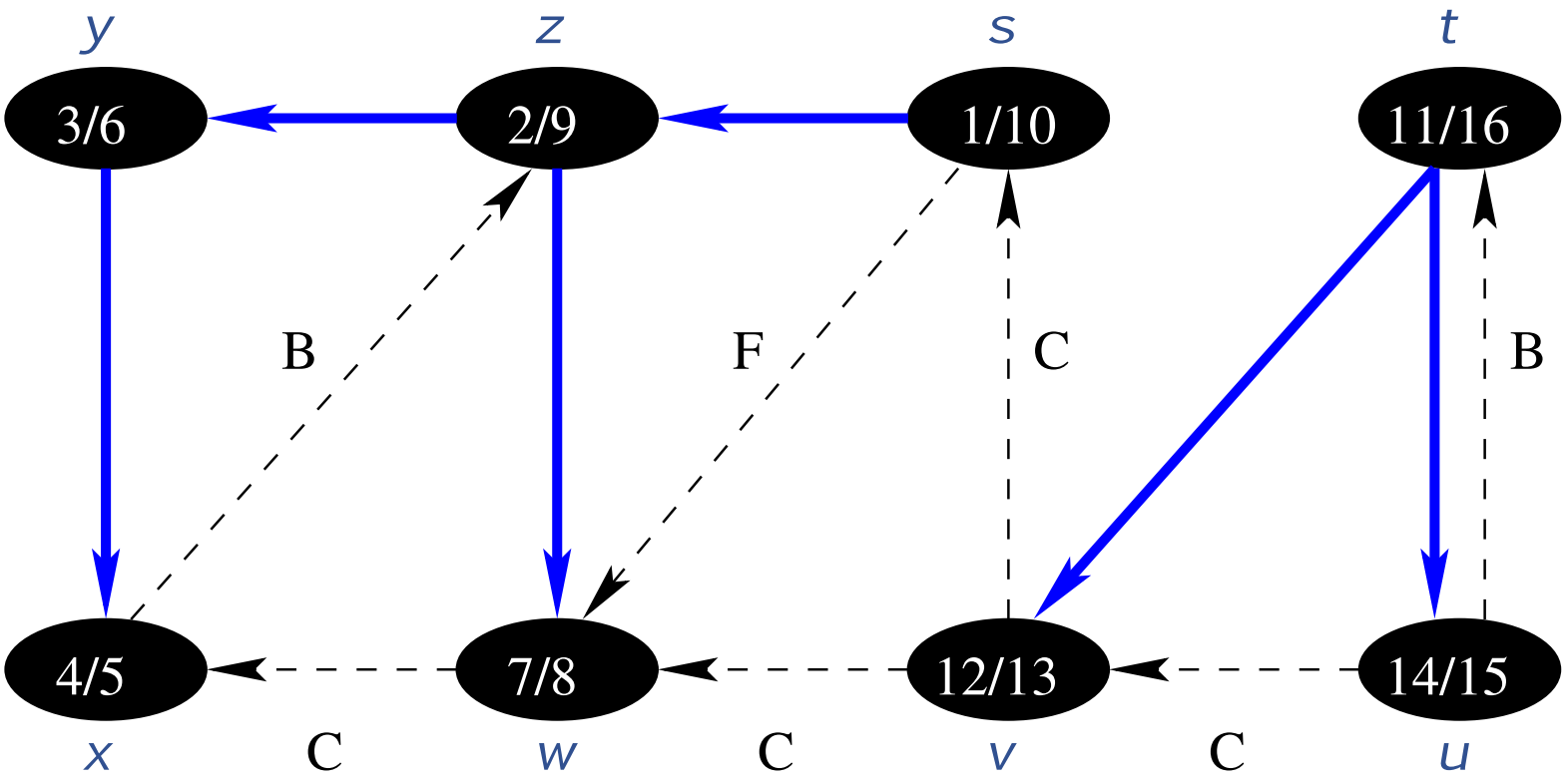
  $\sum_{v \in V} |\mathrm{Adj}(v)| = \Theta(E)$.

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

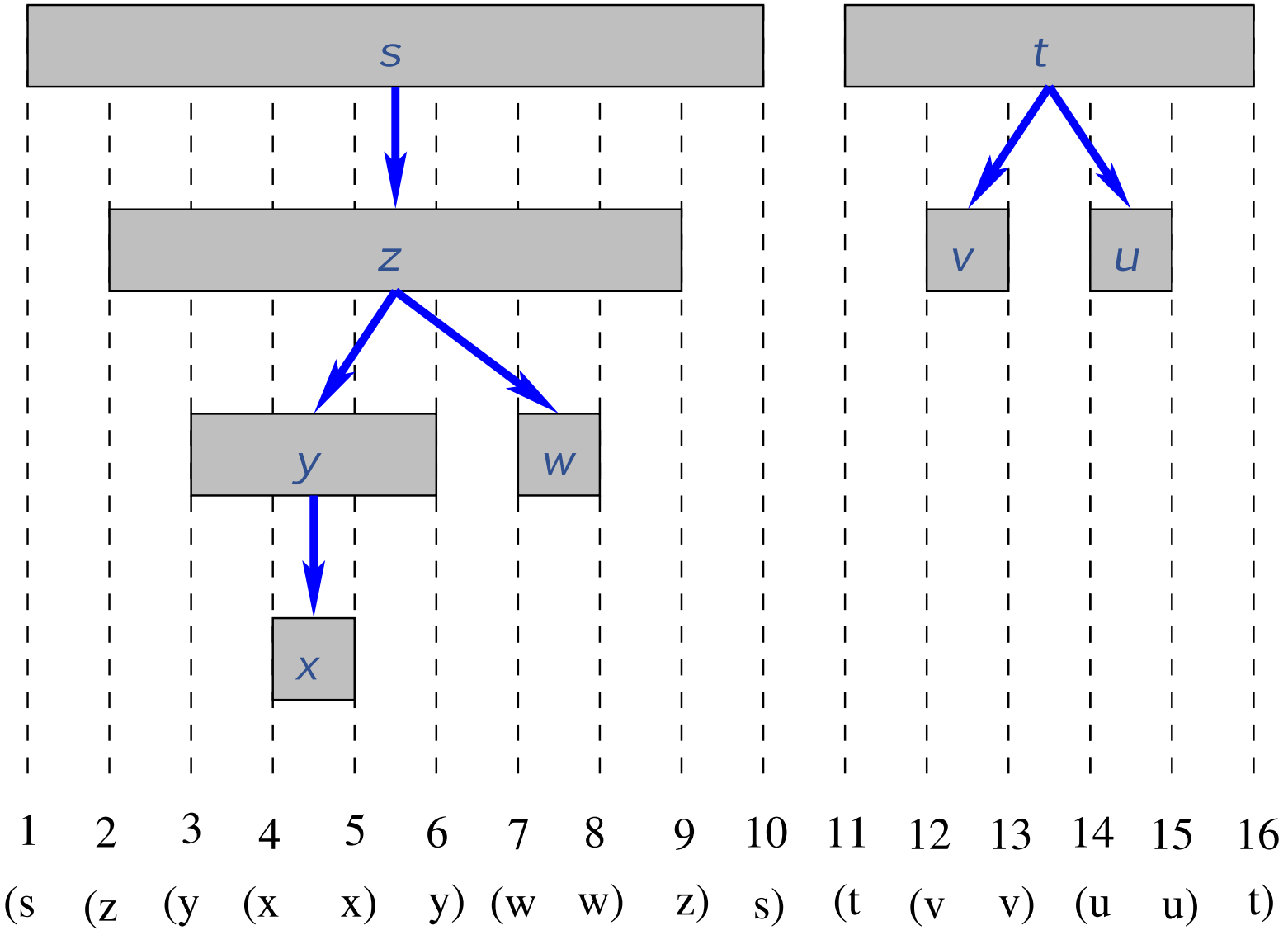
Estrutura de parênteses

- 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 situações abaixo:

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

Estrutura de parênteses

Prova (rascunho). Podemos supor que $d[u] < d[v]$. Temos dois casos: Caso 1: $d[v] < f[u]$ ou Caso 2: $f[u] < d[v]$

- Caso 1: $d[v] < f[u]$: v foi descoberto enquanto u era cinza. Logo, v é descendente de u. Além disso, as arestas que saem de v são exploradas e v é finalizado ($cor[v] = \text{preto}$) antes da busca voltar a u. Logo, o intervalo $[d[v], f[v]]$ está contido em $[d[u], f[u]]$.

Estrutura de parênteses

Prova (rascunho). Podemos supor que $d[u] < d[v]$. Temos dois casos: Caso 1: $d[v] < f[u]$ ou Caso 2: $f[u] < d[v]$

- Caso 2: $f[u] < d[v]$: u é finalizado antes de v ser descoberto. Logo,

$$d[u] < f[u] < d[v] < f[v]$$

e os intervalos $[d[u], f[u]]$ e $[d[v], f[v]]$ são disjuntos. Neste caso, nenhum desses vértices foi descoberto enquanto o outro estava cinza, e assim, nenhum é descendente do outro na Floresta de BP.

Estrutura de parênteses

Corolário. (Intervalos encaixantes para descendentes)

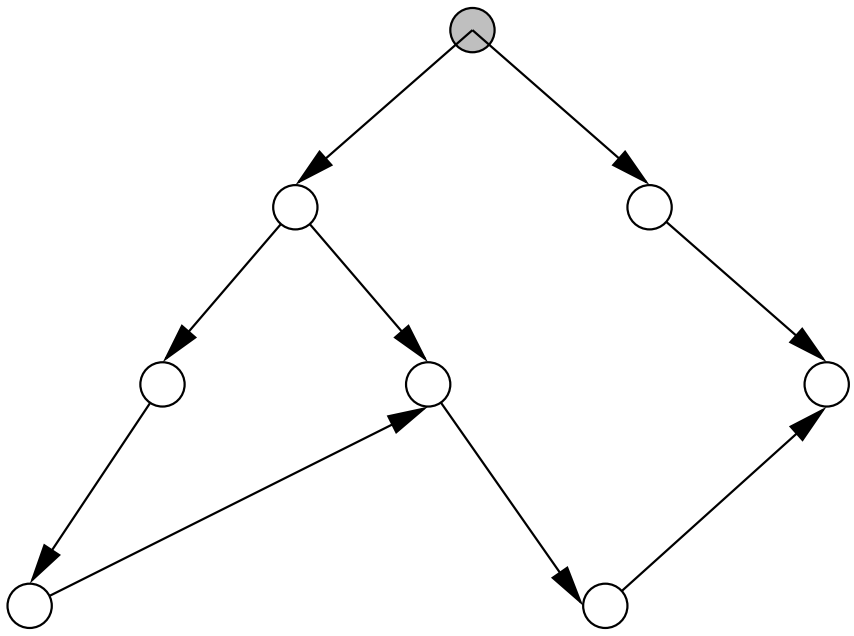
Um vértice v é um descendente próprio de u na Floresta de BP se e somente se $d[u] < d[v] < f[v] < f[u]$.

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 (com exceção de u).



Teorema do Caminho Branco

Prova (rascunho).
$\Rightarrow$: Se $u = v$ então o resultado é óbvio. Suponha que v é um descendente próprio de u na Floresta de BP. Como $d[u] < d[v]$, v ainda é branco no instante $d[u]$. Neste instante, o caminho de u a v na floresta é branco (com exceção de u).

Teorema do Caminho Branco

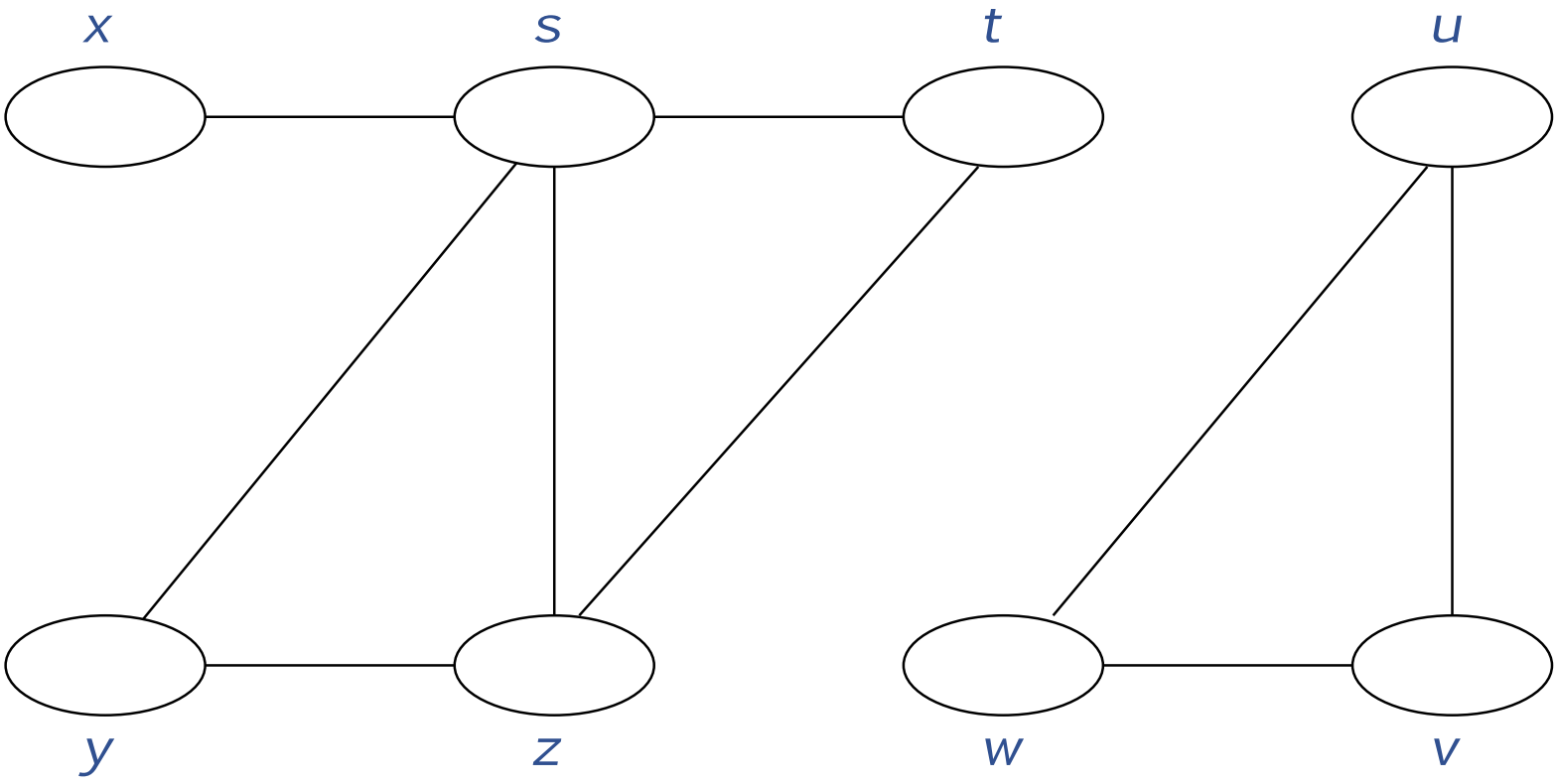
$\Leftarrow$: Suponha que no instante $d[u]$ existe um caminho branco de u a v (com exceção de u). Suponha por contradição que v não se torna descendente de u na Floresta de BP. Podemos supor que todos os vértices que precedem v tornam-se descendentes de u. Seja w o vértice que antecede imediatamente v nesse caminho. Assim, $f[w] \leq f[u]$. Como v é descoberto depois de u, mas antes de w ser finalizado, temos

$$d[u] < d[v] < f[w] < f[u].$$

Logo, o intervalo $[d[v], f[v]]$ está contido em $[d[u], f[u]]$ e v é descendente de u, uma contradição.

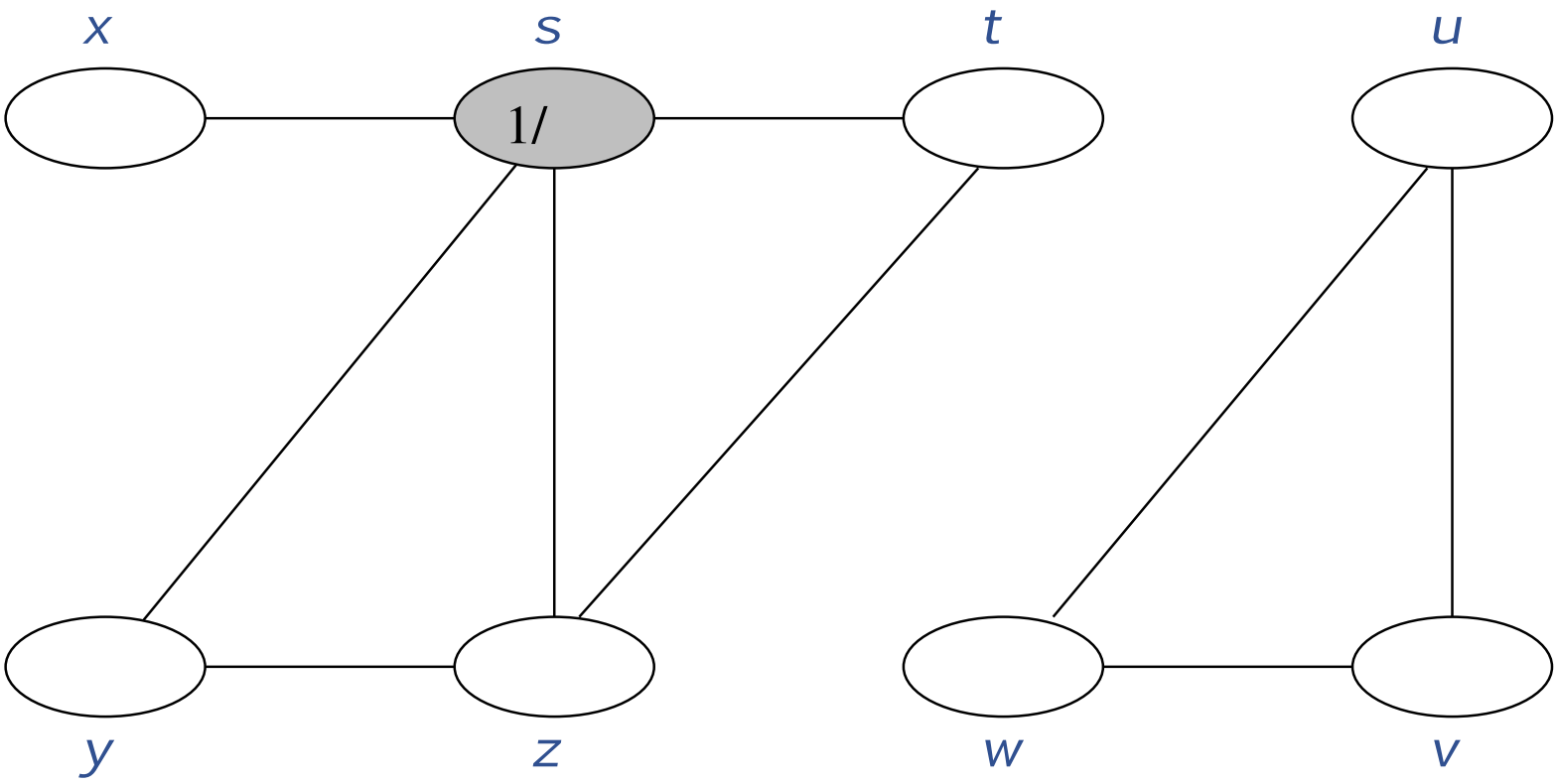
Componentes Conexos

Aplicação: componentes conexos

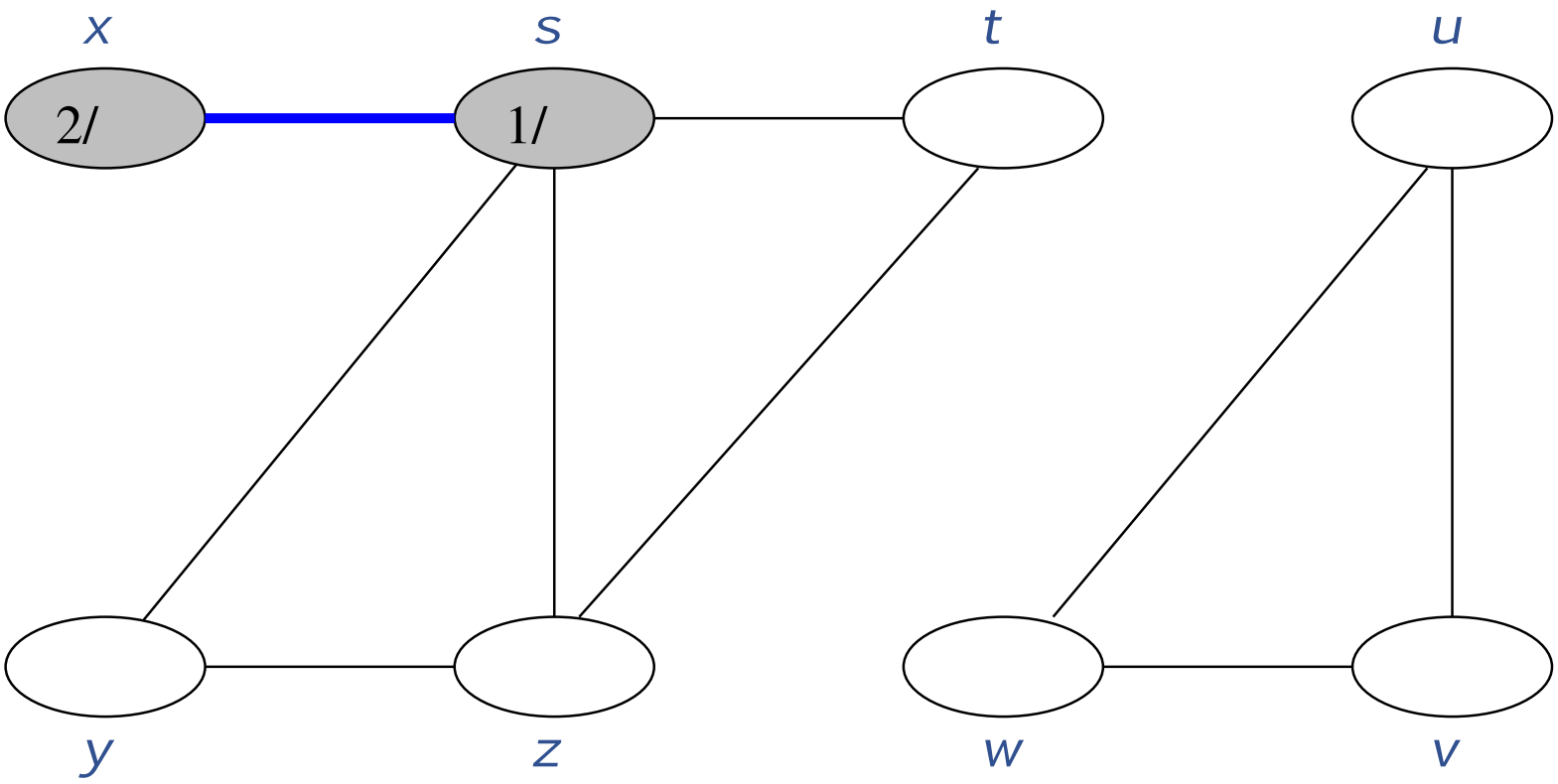


Problema: determinar os componentes conexos de um grafo.

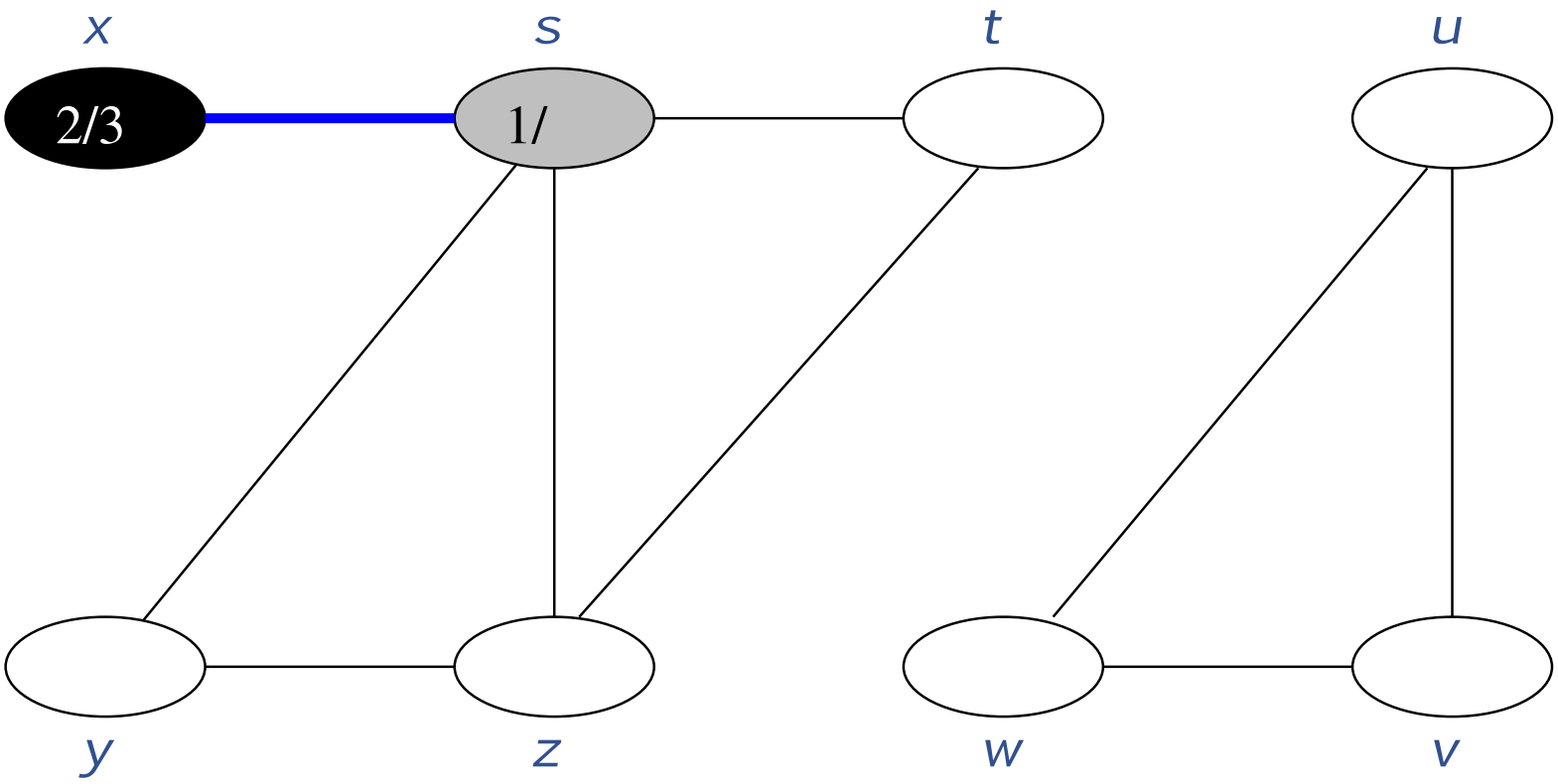
Aplicação: componentes conexos



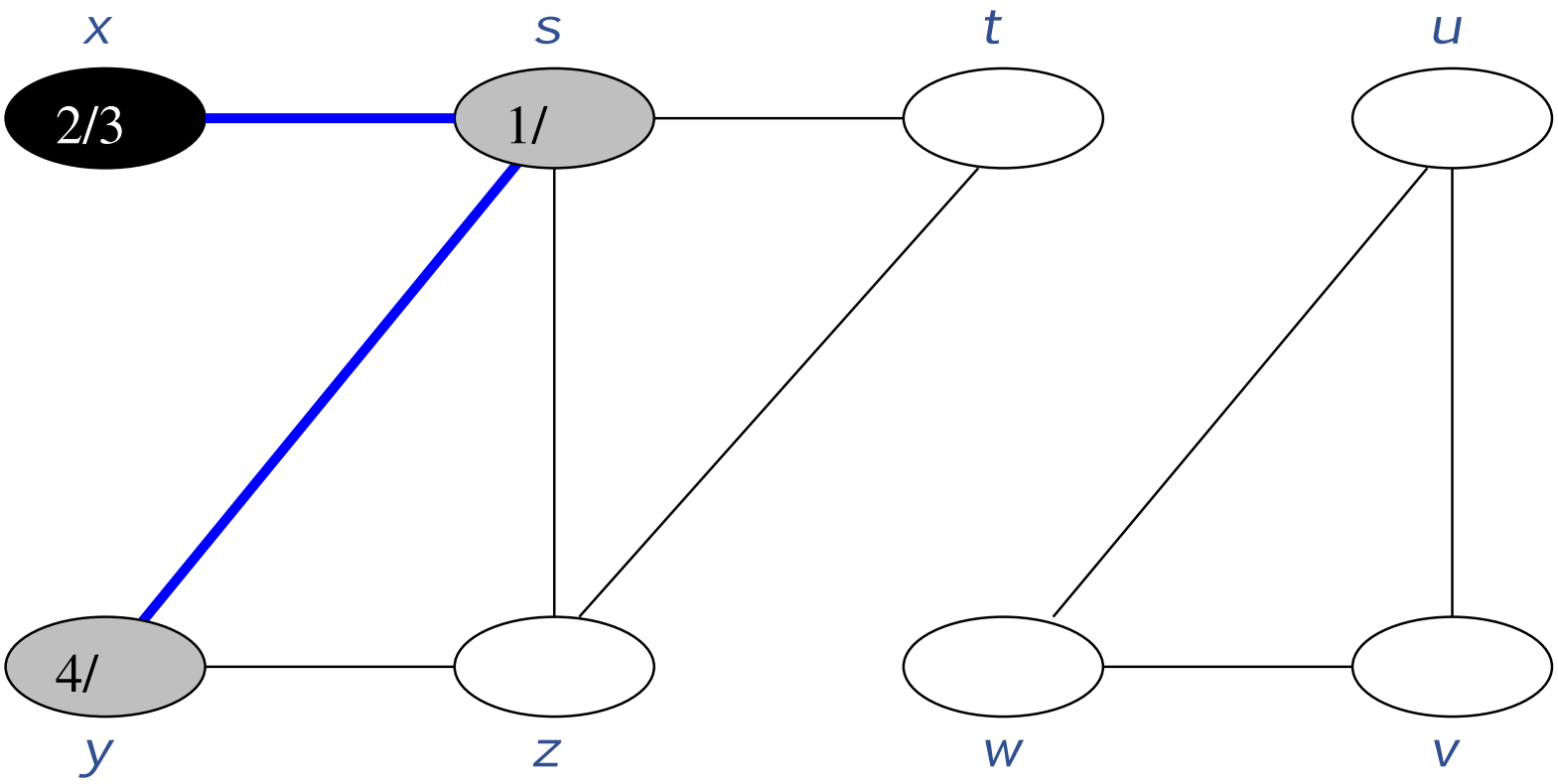
Aplicação: componentes conexos



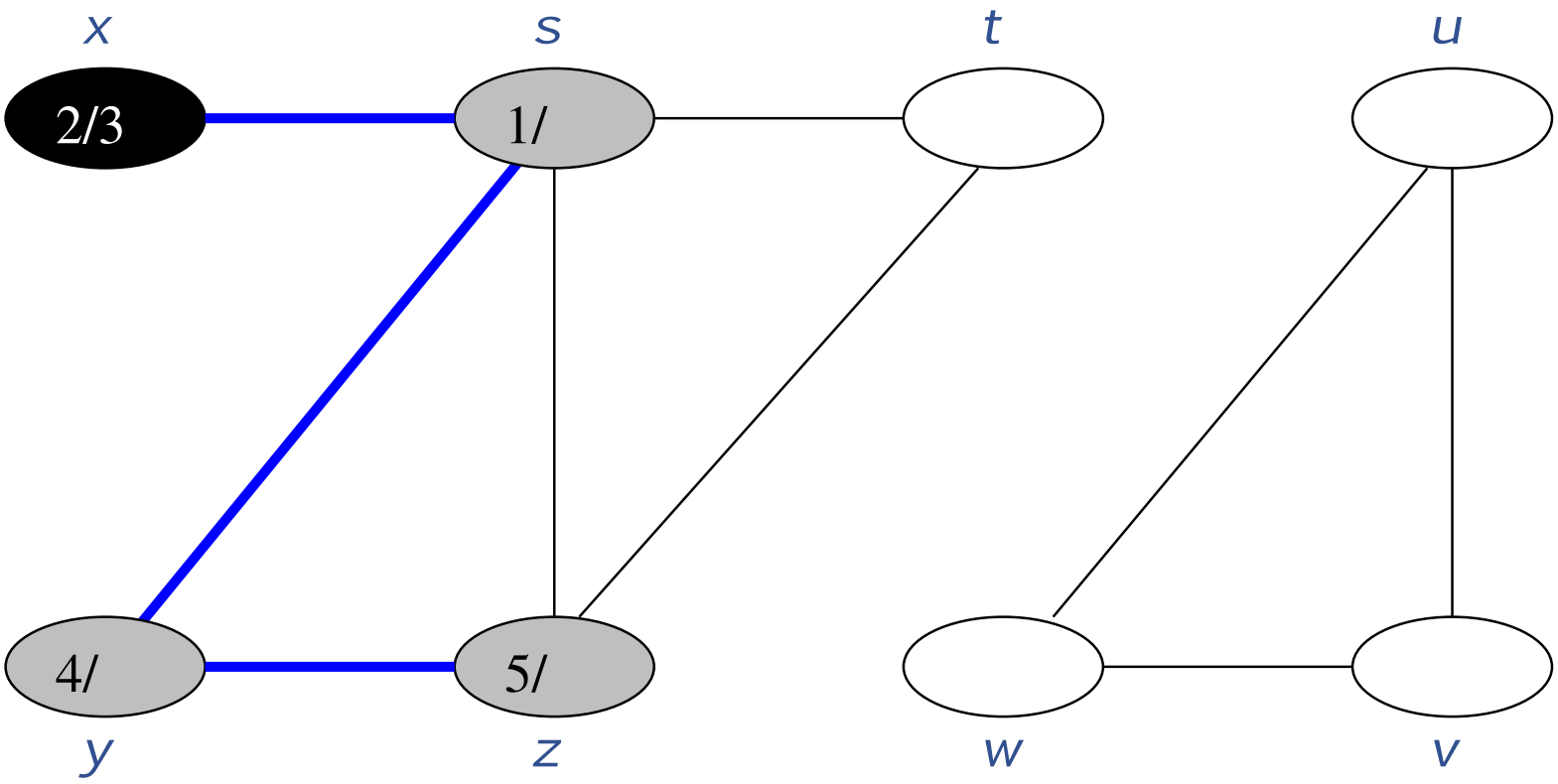
Aplicação: componentes conexos



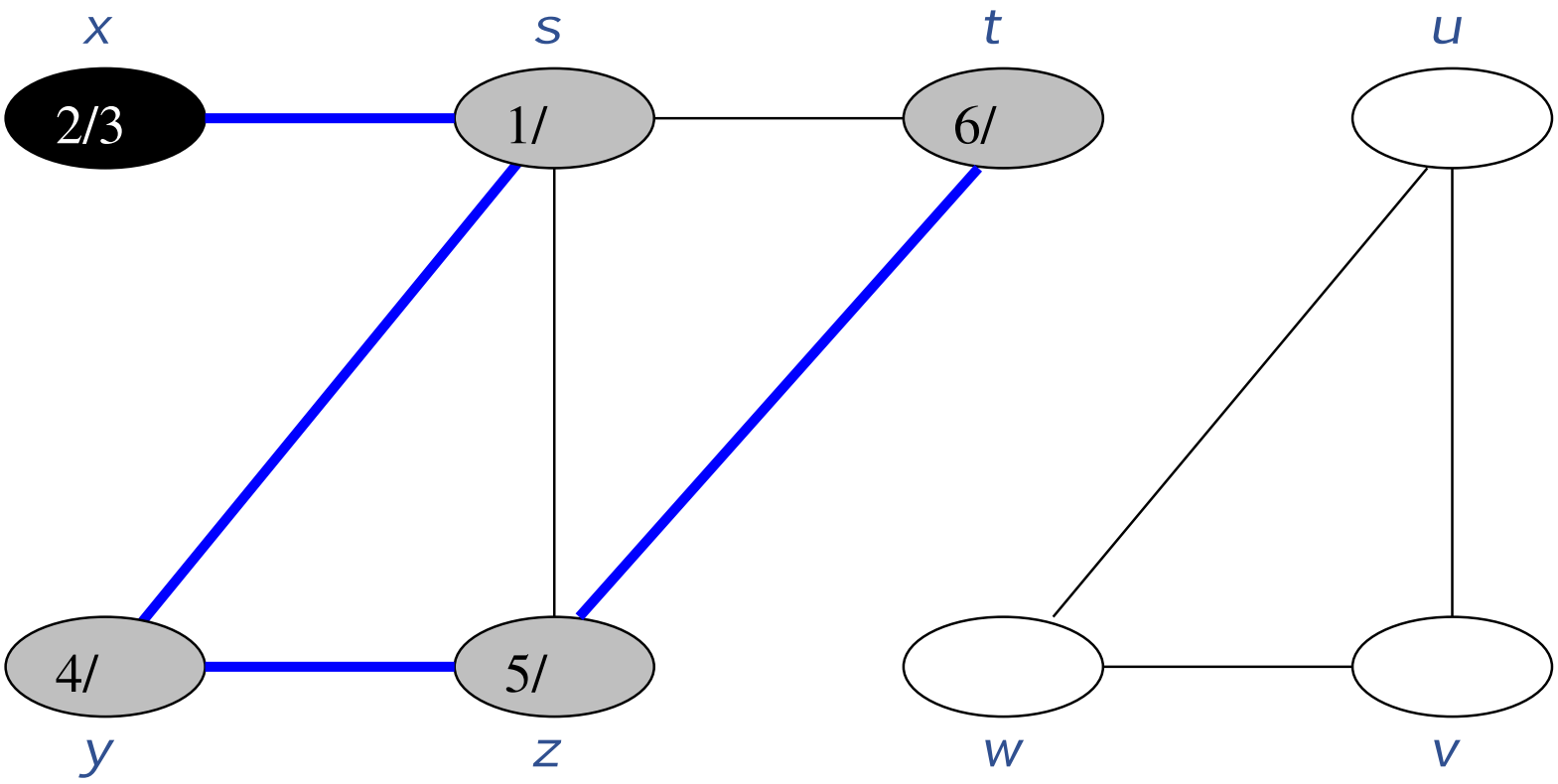
Aplicação: componentes conexos



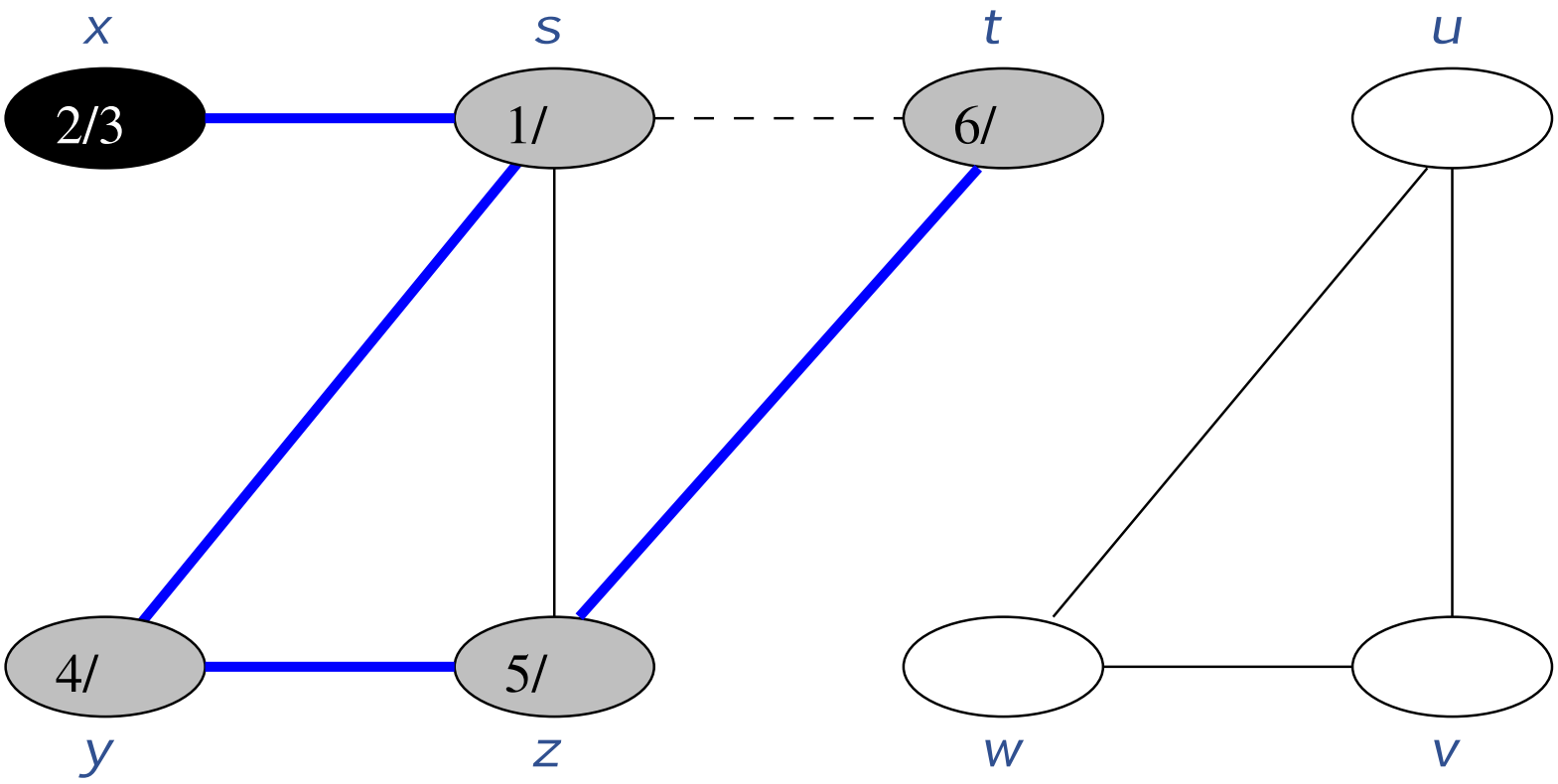
Aplicação: componentes conexos



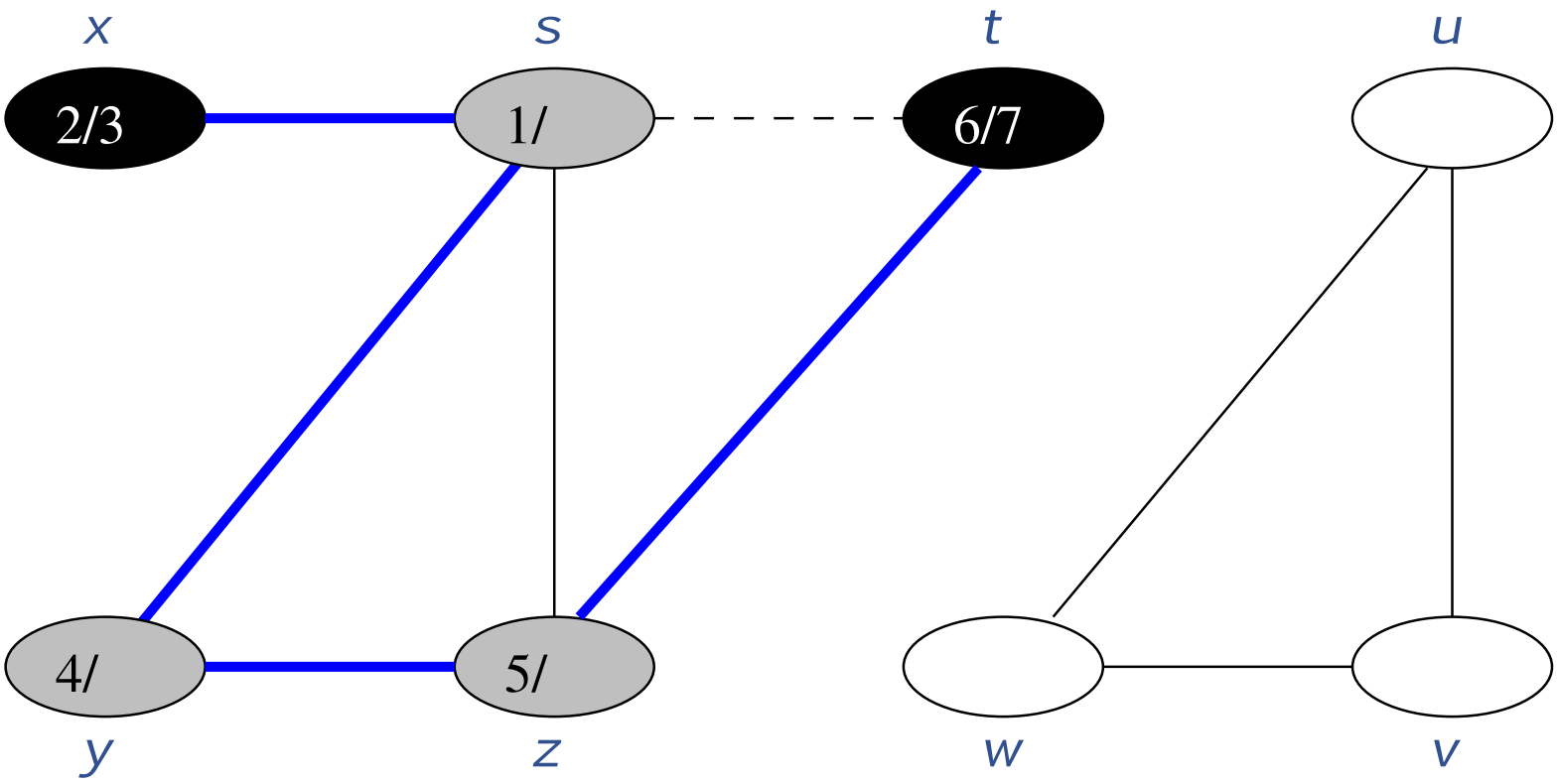
Aplicação: componentes conexos



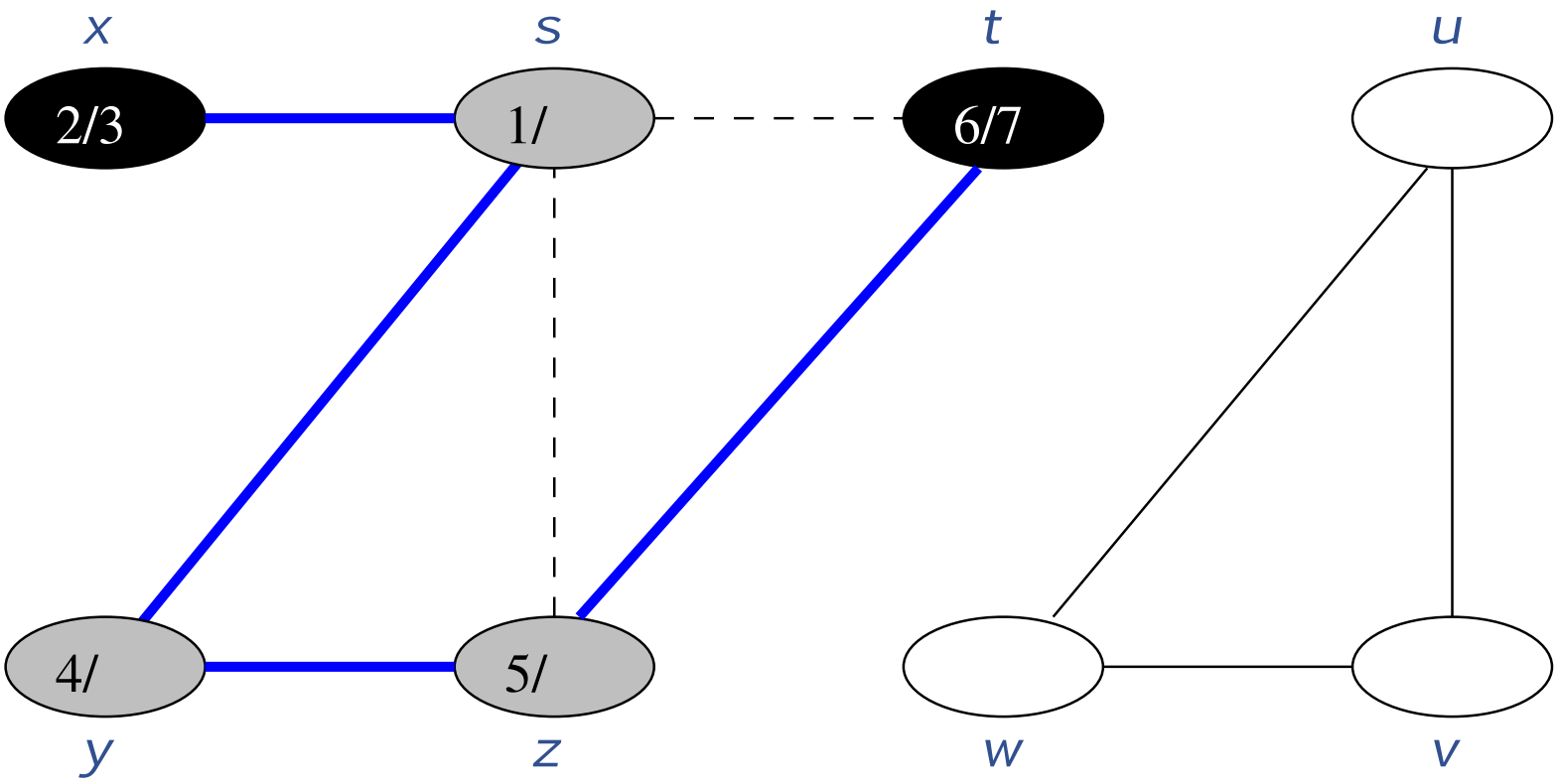
Aplicação: componentes conexos



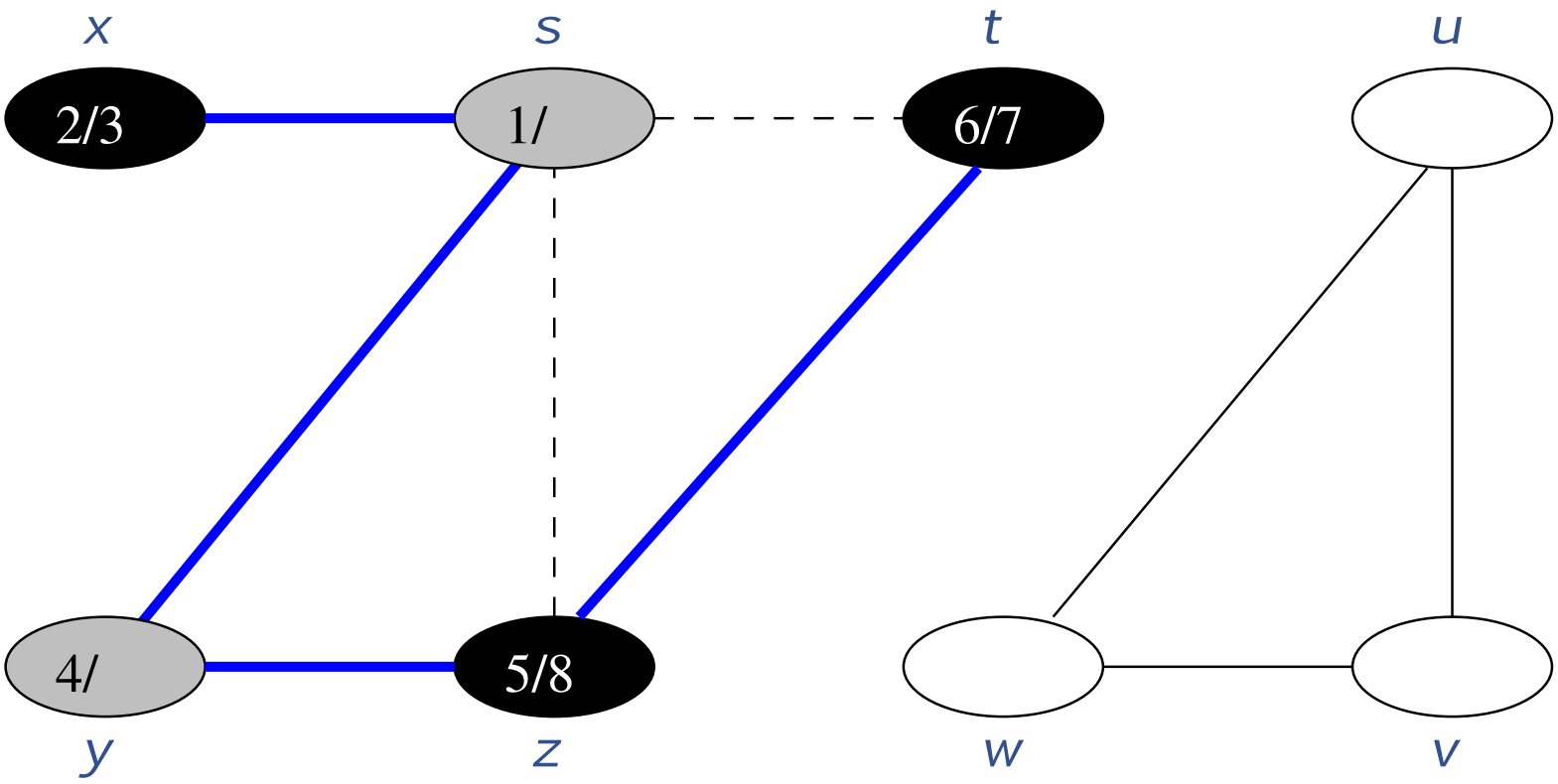
Aplicação: componentes conexos



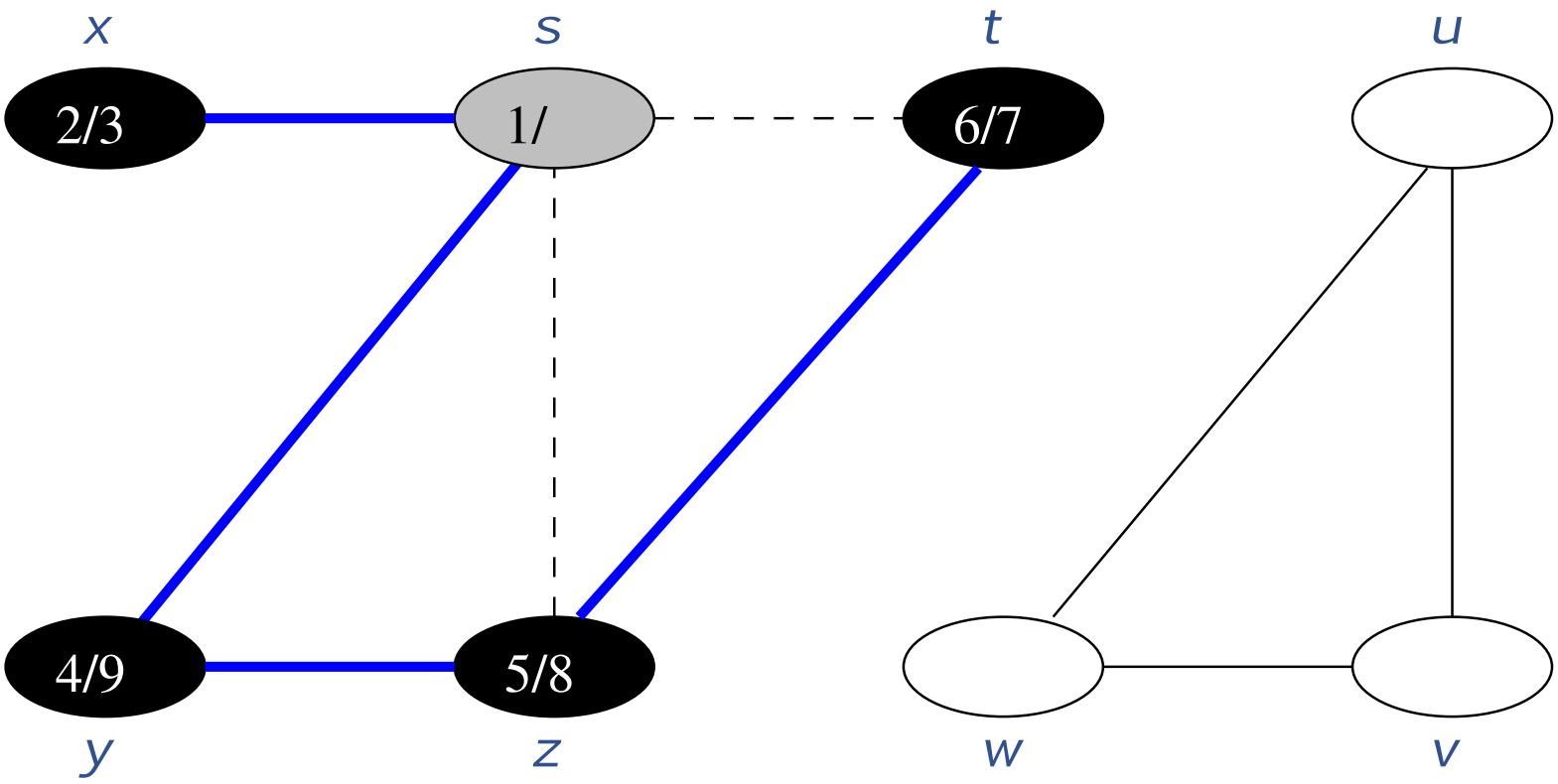
Aplicação: componentes conexos



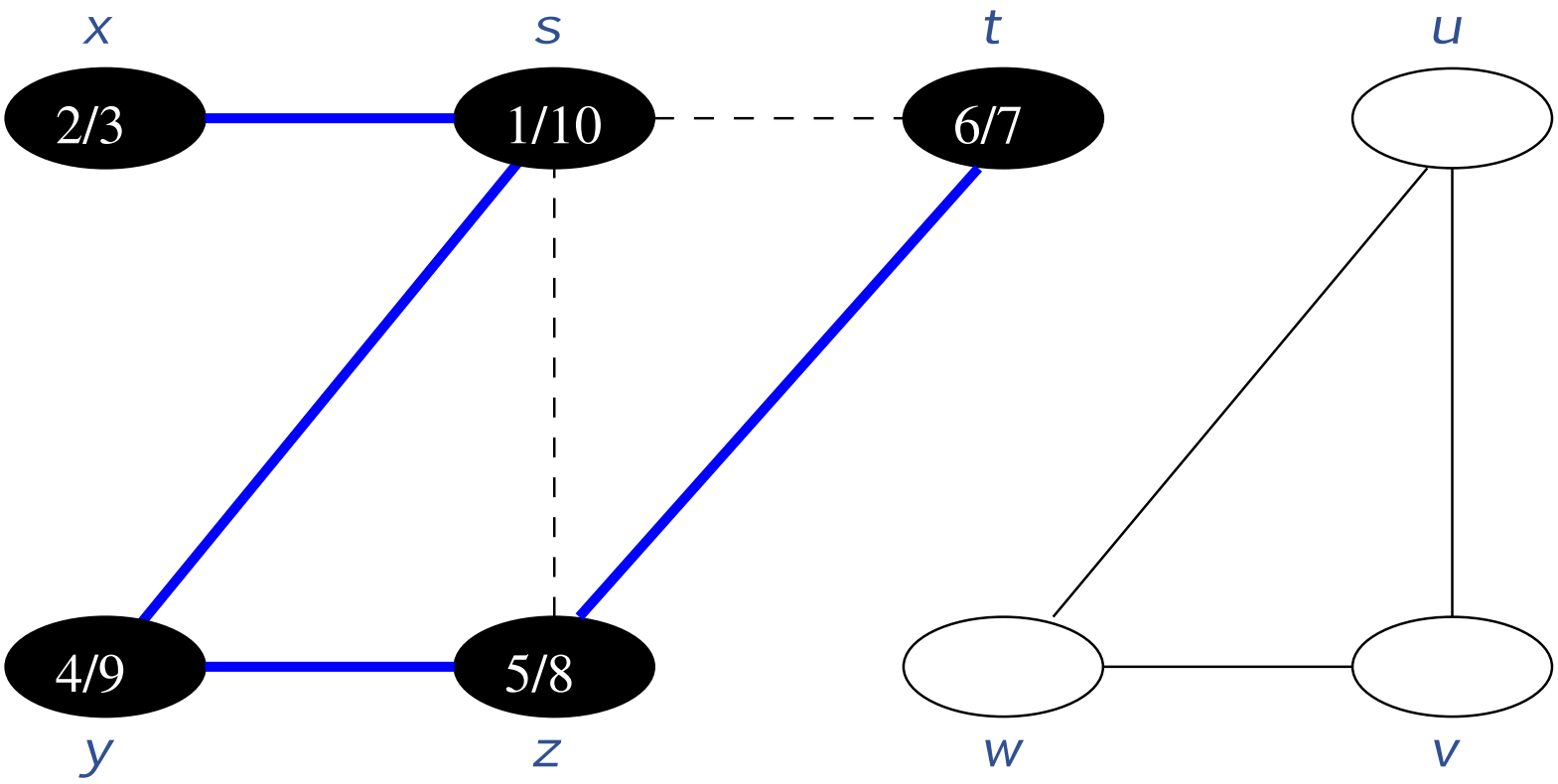
Aplicação: componentes conexos



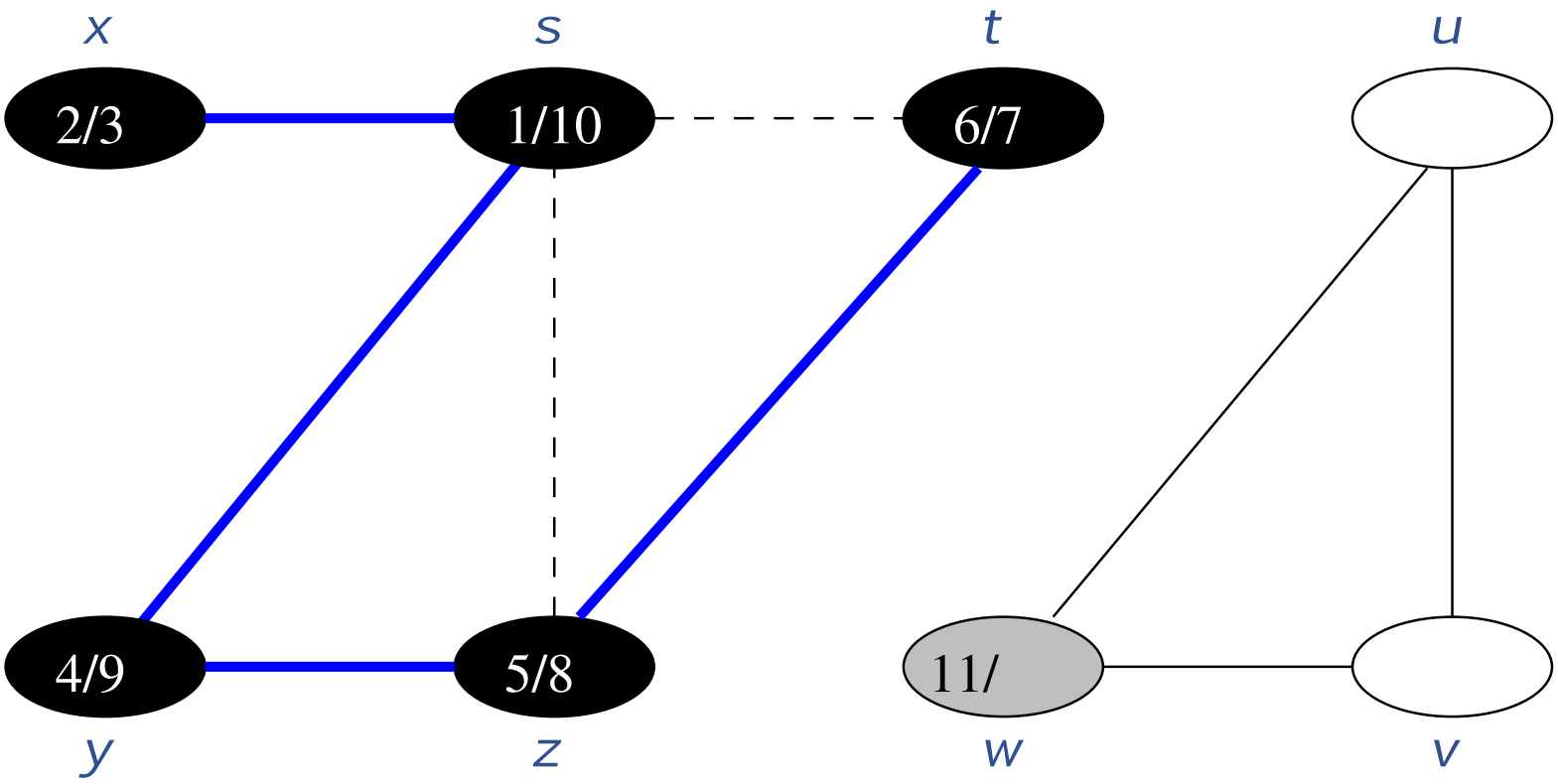
Aplicação: componentes conexos



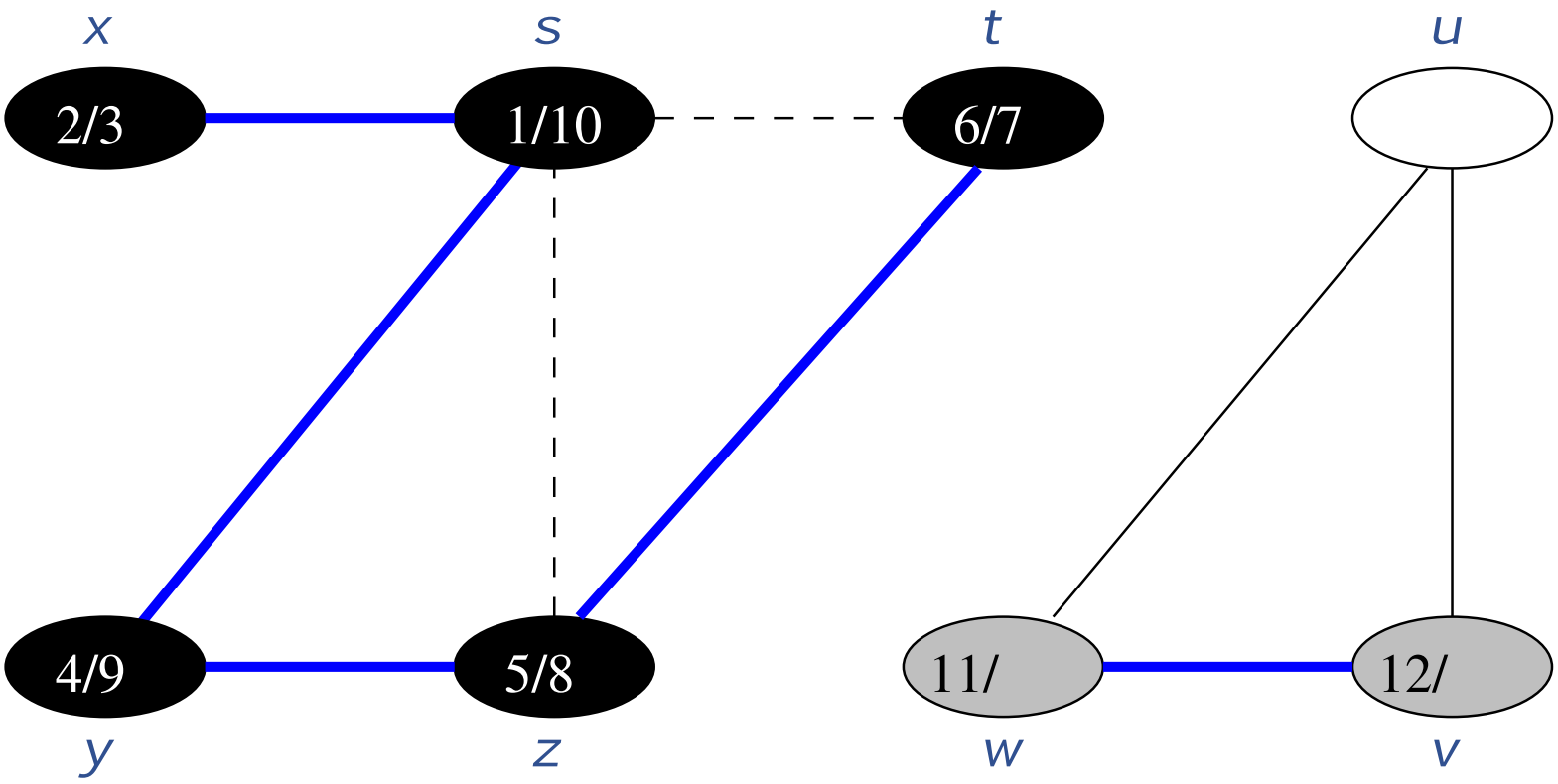
Aplicação: componentes conexos



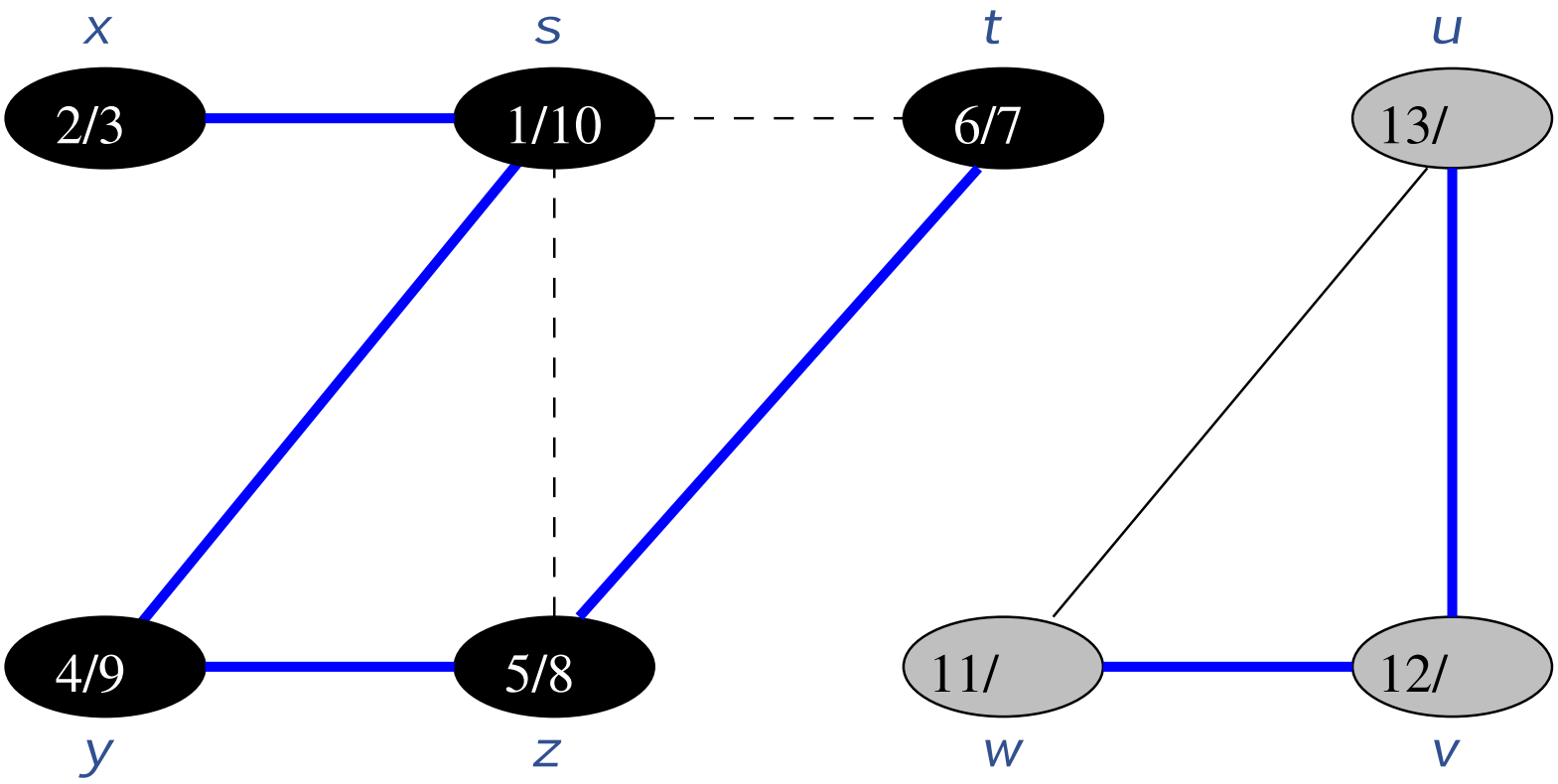
Aplicação: componentes conexos



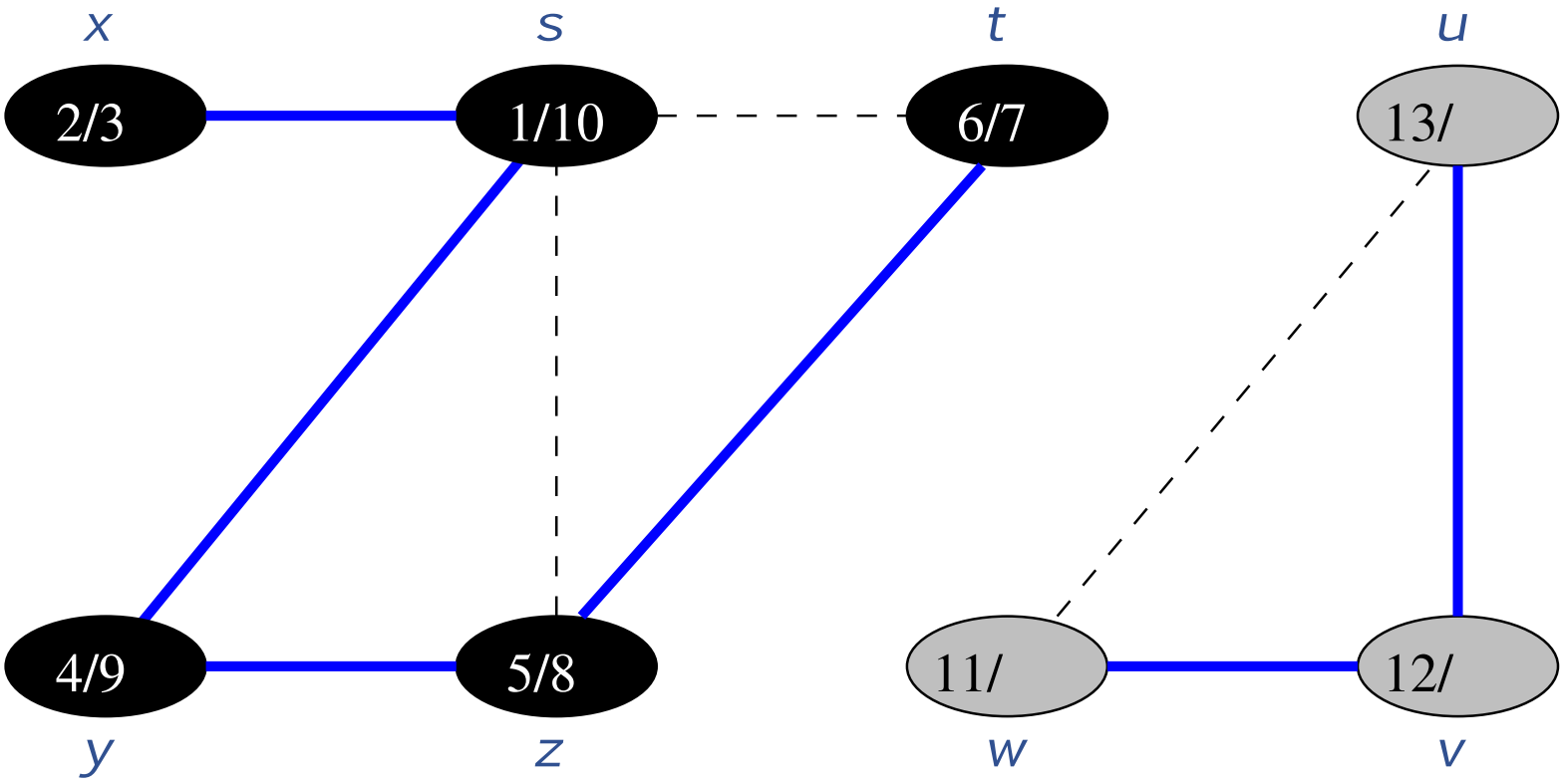
Aplicação: componentes conexos



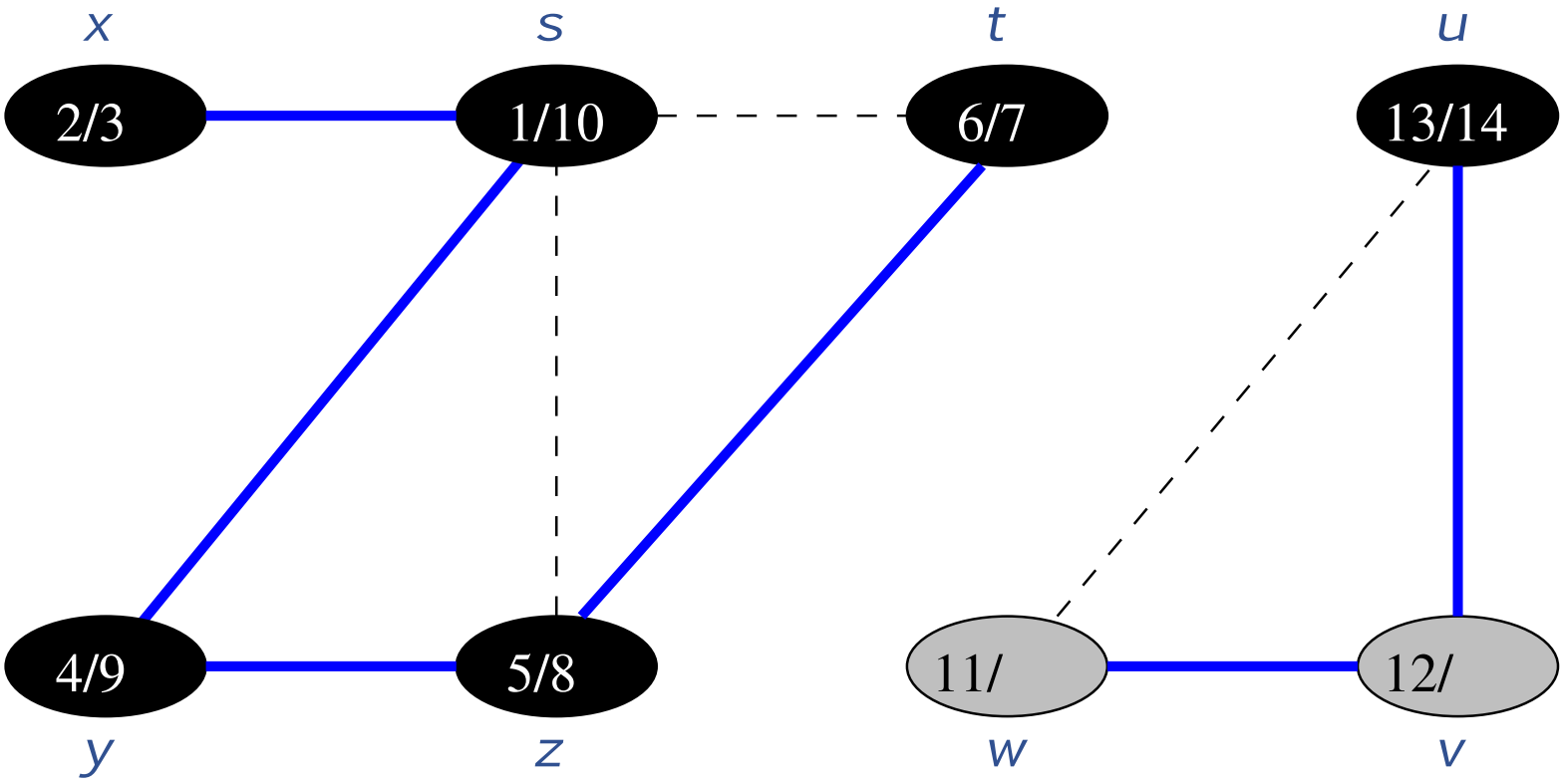
Aplicação: componentes conexos



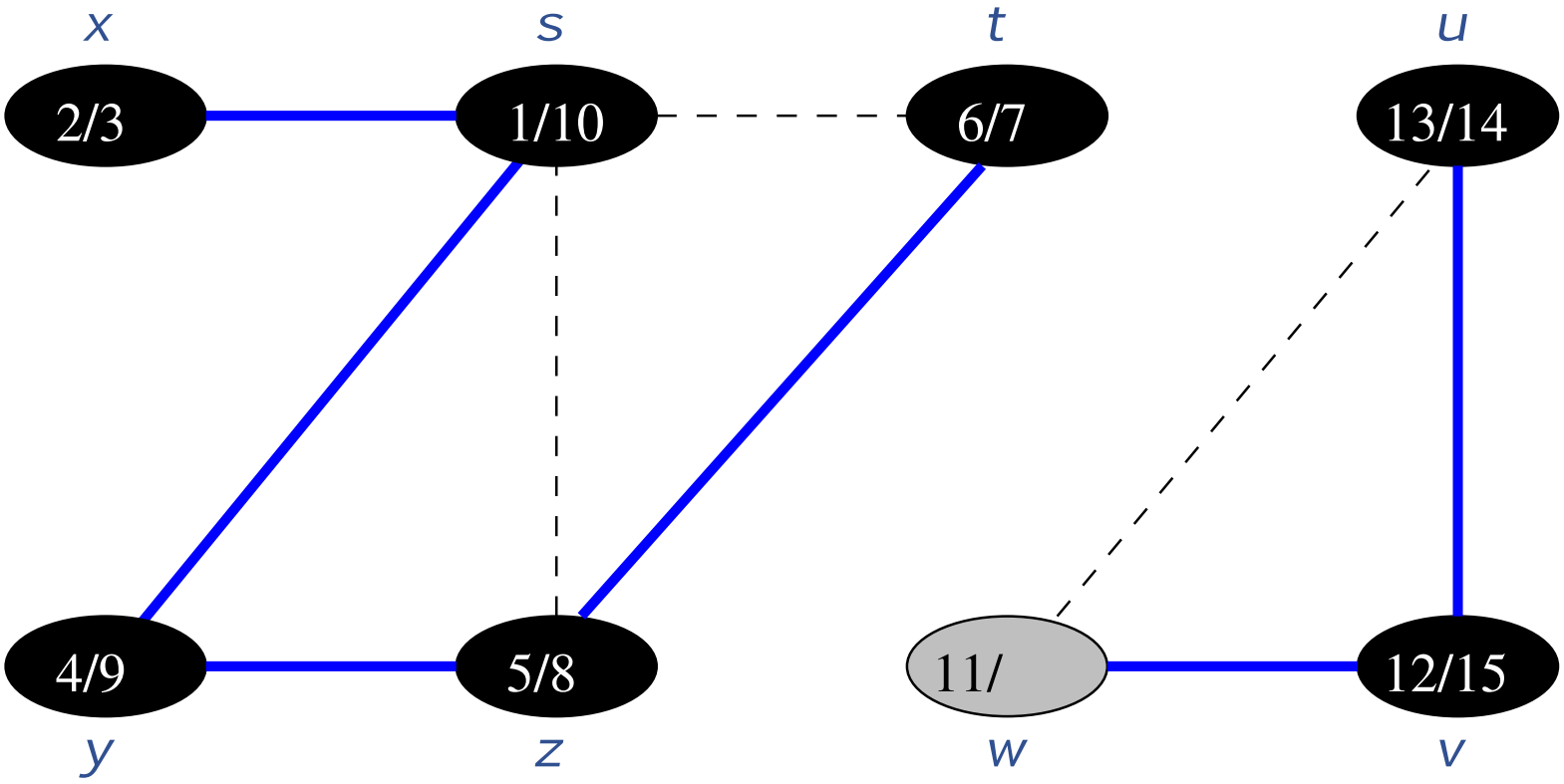
Aplicação: componentes conexos



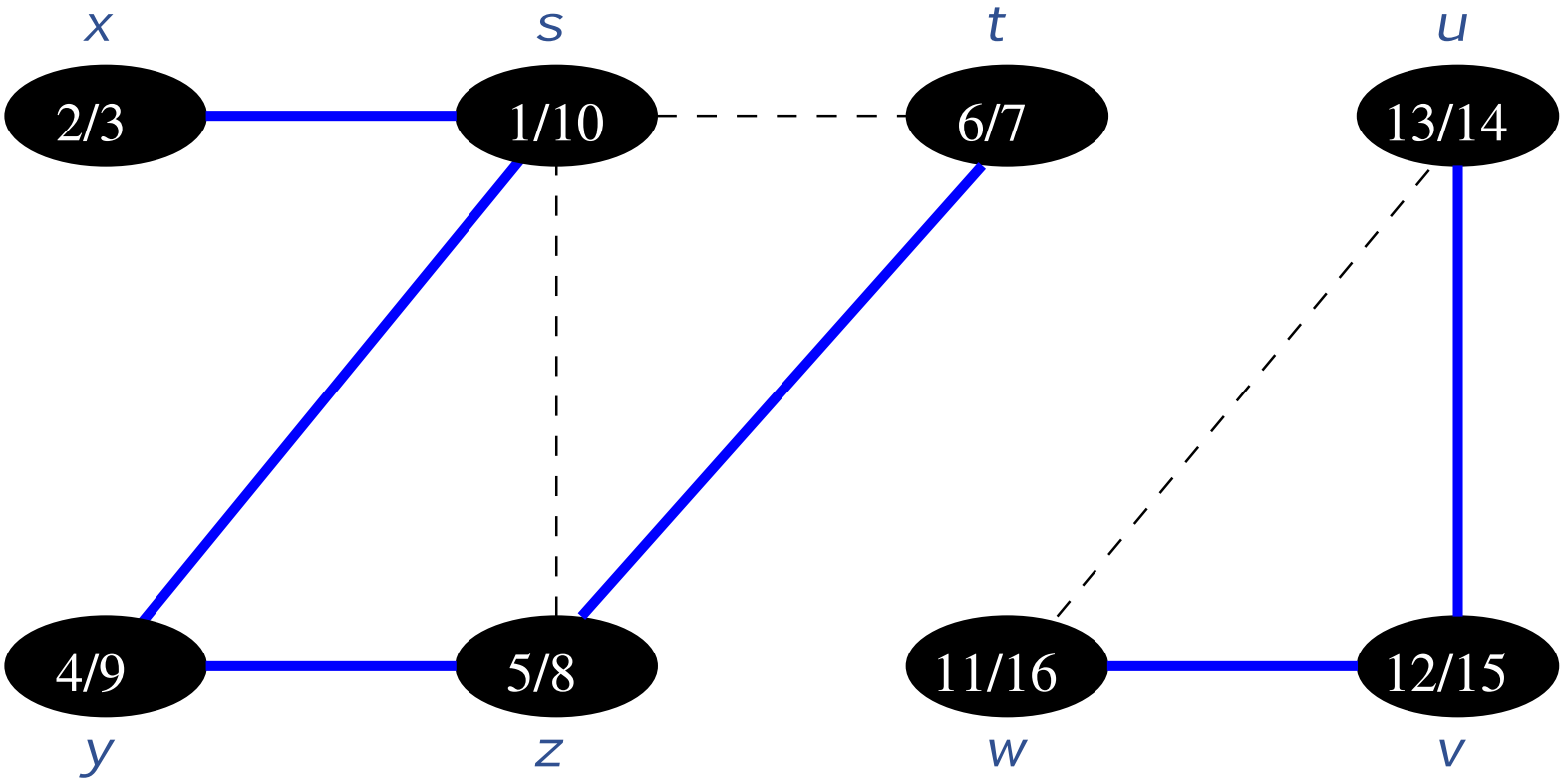
Aplicação: componentes conexos



Aplicação: componentes conexos



Aplicação: componentes conexos



Componentes conexos

O número de componentes é o **número de vezes** que DFS-VISIT(u) é chamado em 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)
```

Componentes conexos

Vamos modificar DFS de modo que:

- determine o número de componentes conexos (ncomps) de G e os componentes sejam enumerados por $1, 2, \ldots,$ ncomps, e
- para cada vértice v determinamos a qual componente ele pertence e guardamos em comp[v].

Componentes conexos

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

Componentes conexos

```
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   tempo ← tempo + 1
10   f[u] ← tempo
11   comp[u] ← ncomps;
```

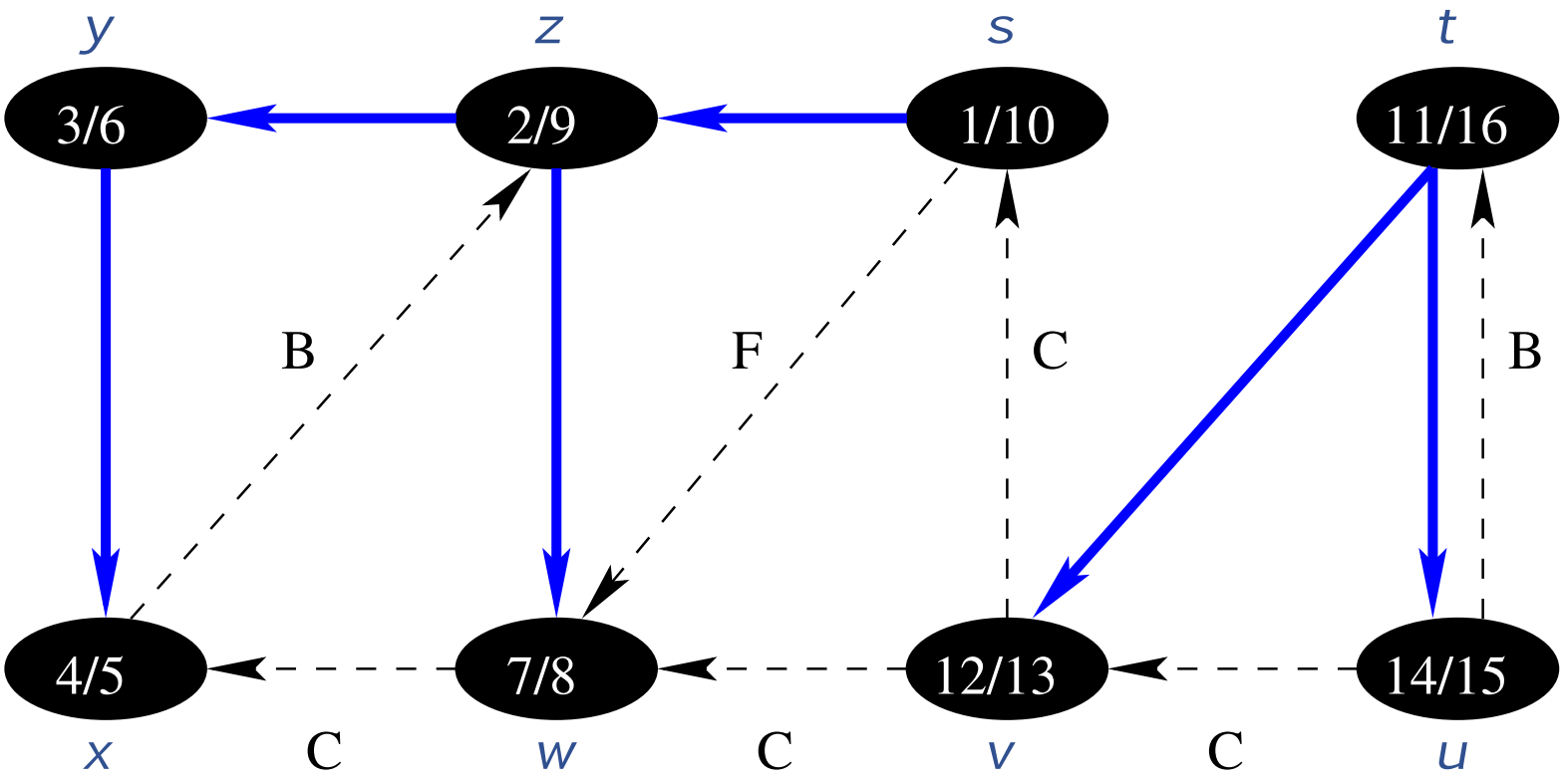
Classificação de arestas

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

Ela classifica as arestas em quatro tipos:

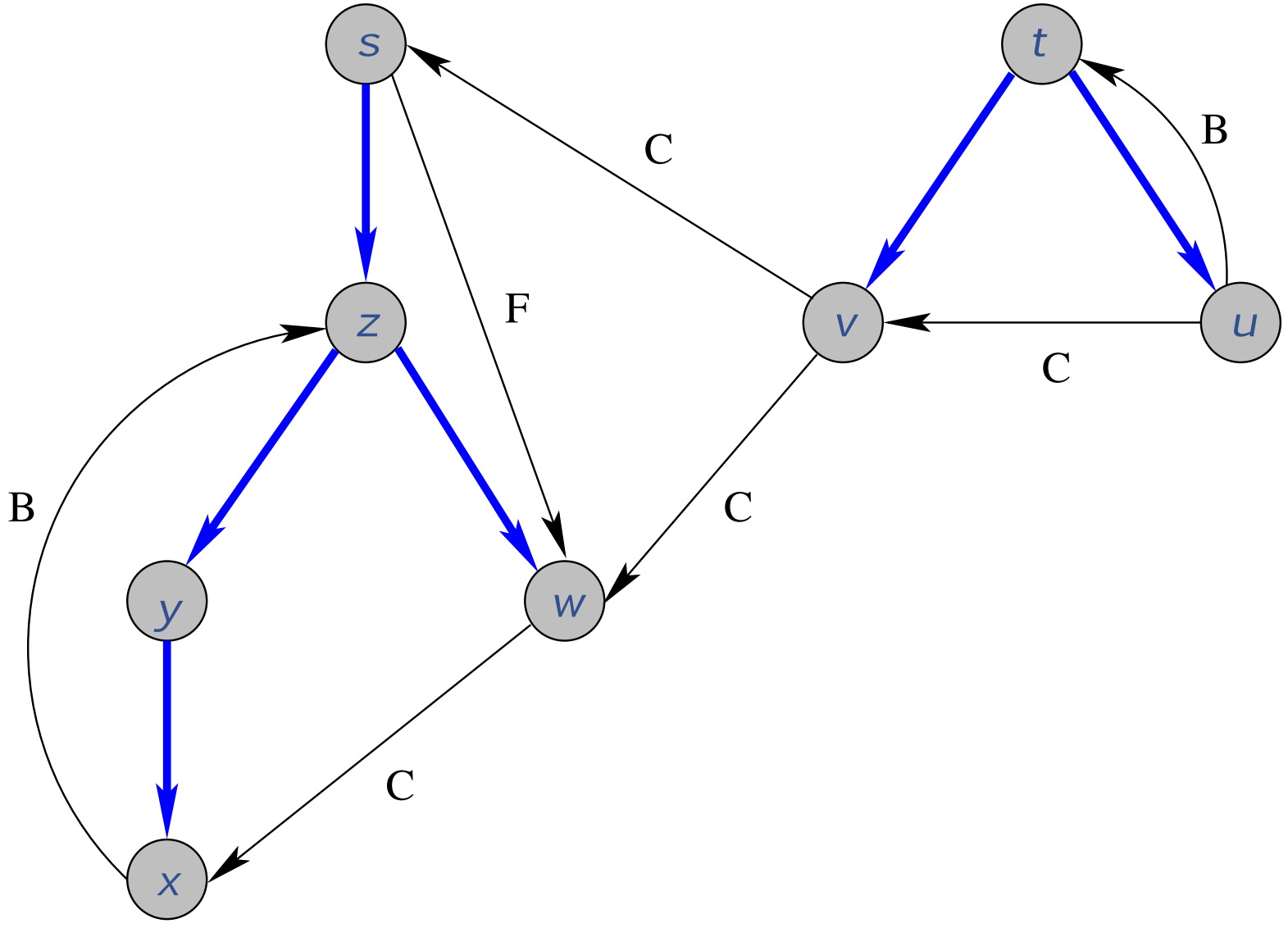
- Arestas da árvore (tree edges): arestas que pertencem à Floresta de BP.
- Arestas de retorno (backward edges): arestas (u, v) ligando um vértice u a um ancestral v na Árvore de BP.
- Arestas de avanço (forward edges): arestas (u, v) ligando um vértice u a um descendente próprio v na Árvore de BP.
-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



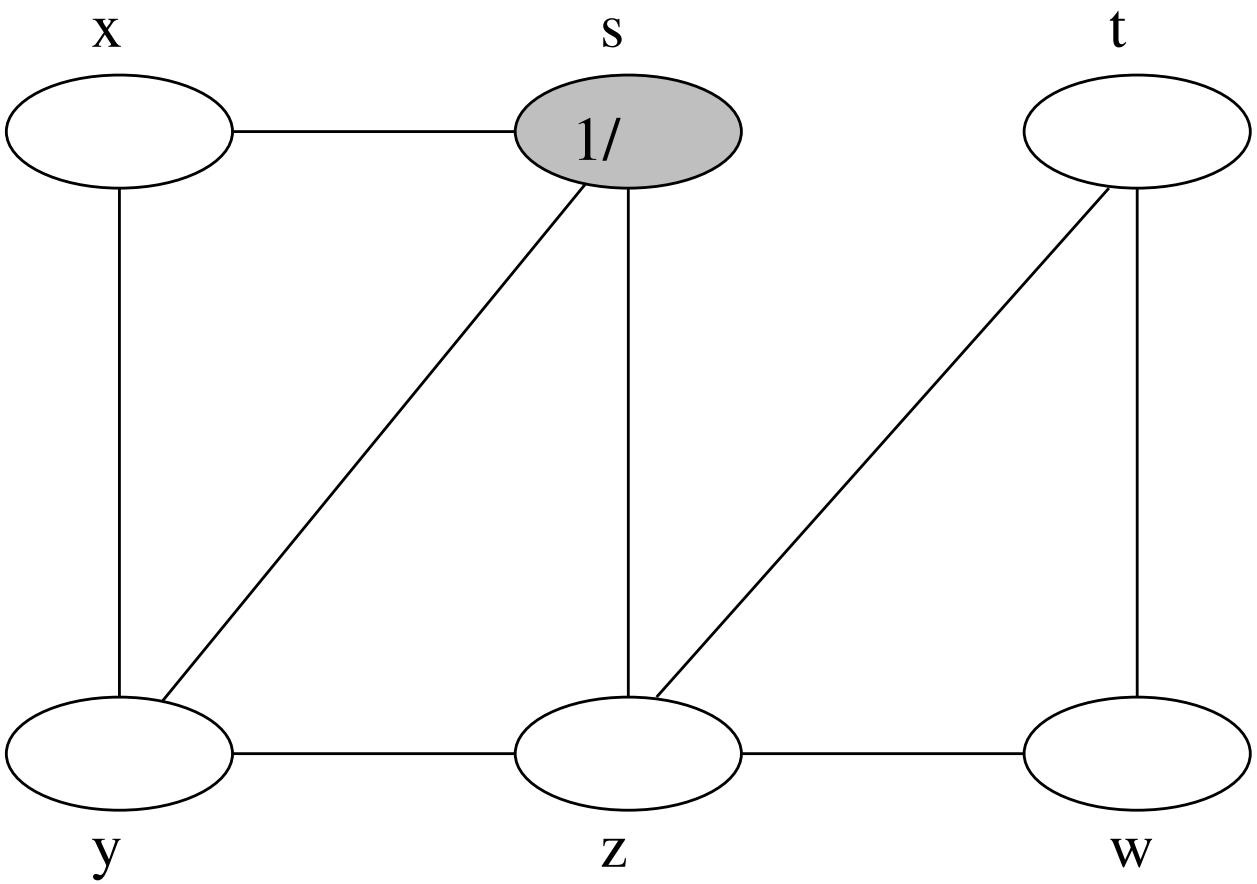
Grafos não direcionados

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

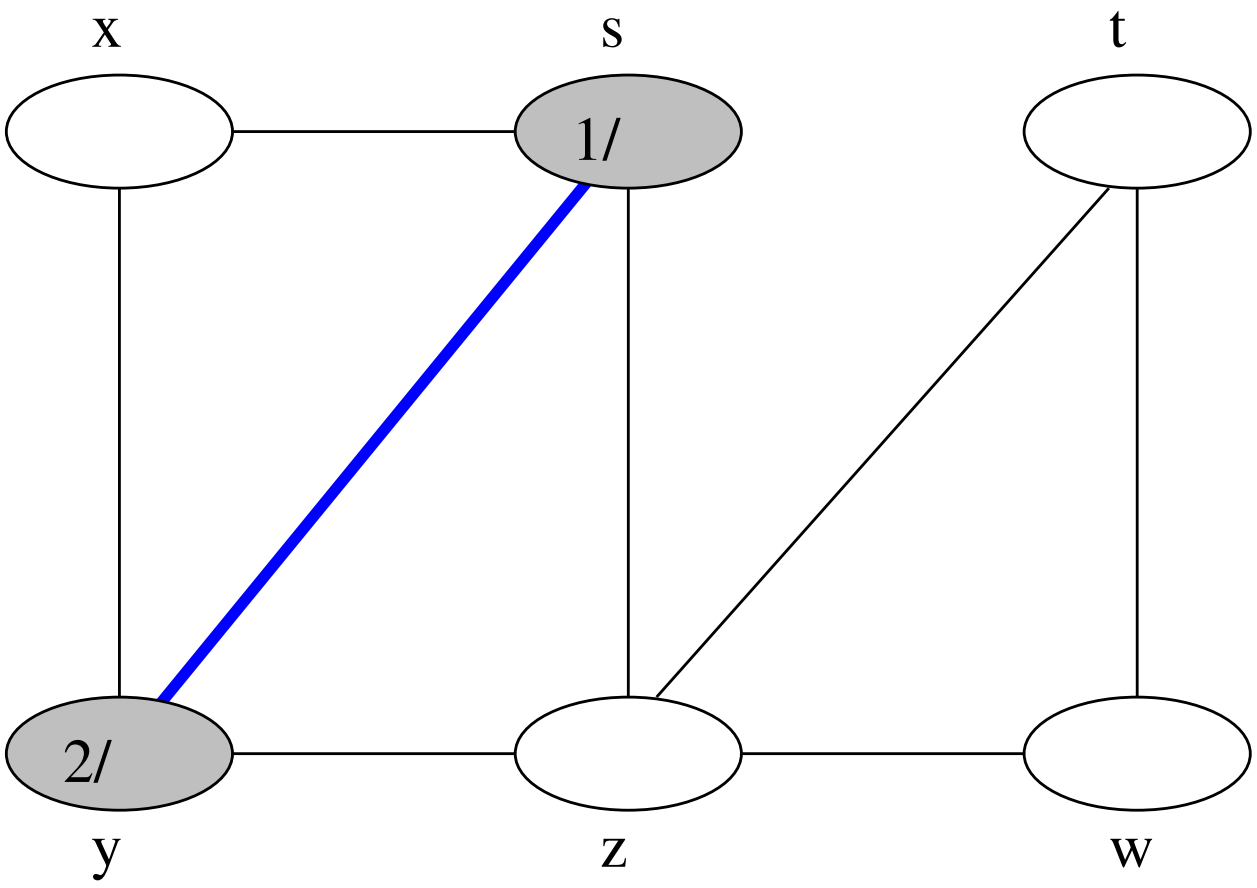
Para grafos não direcionados, existem apenas dois tipos de arestas.

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

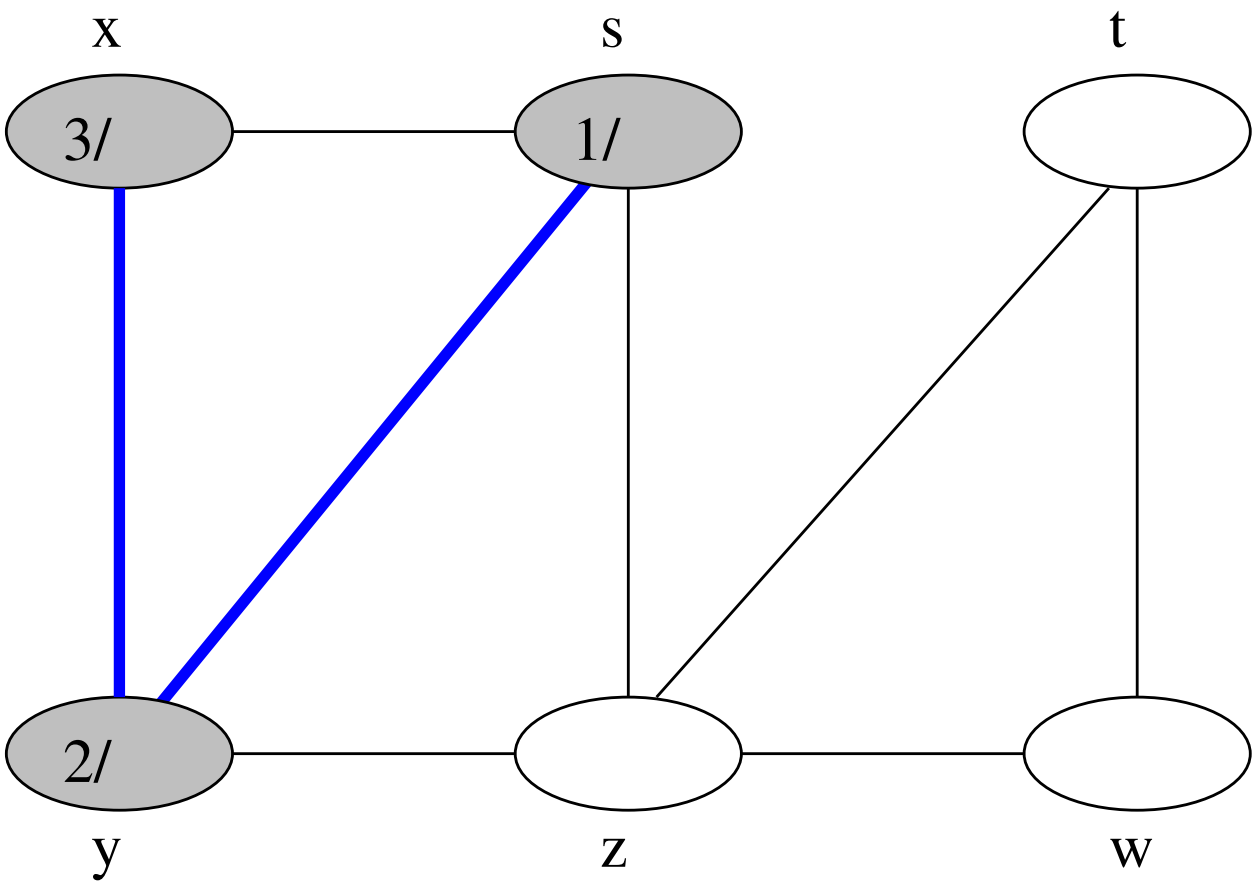
Exemplo



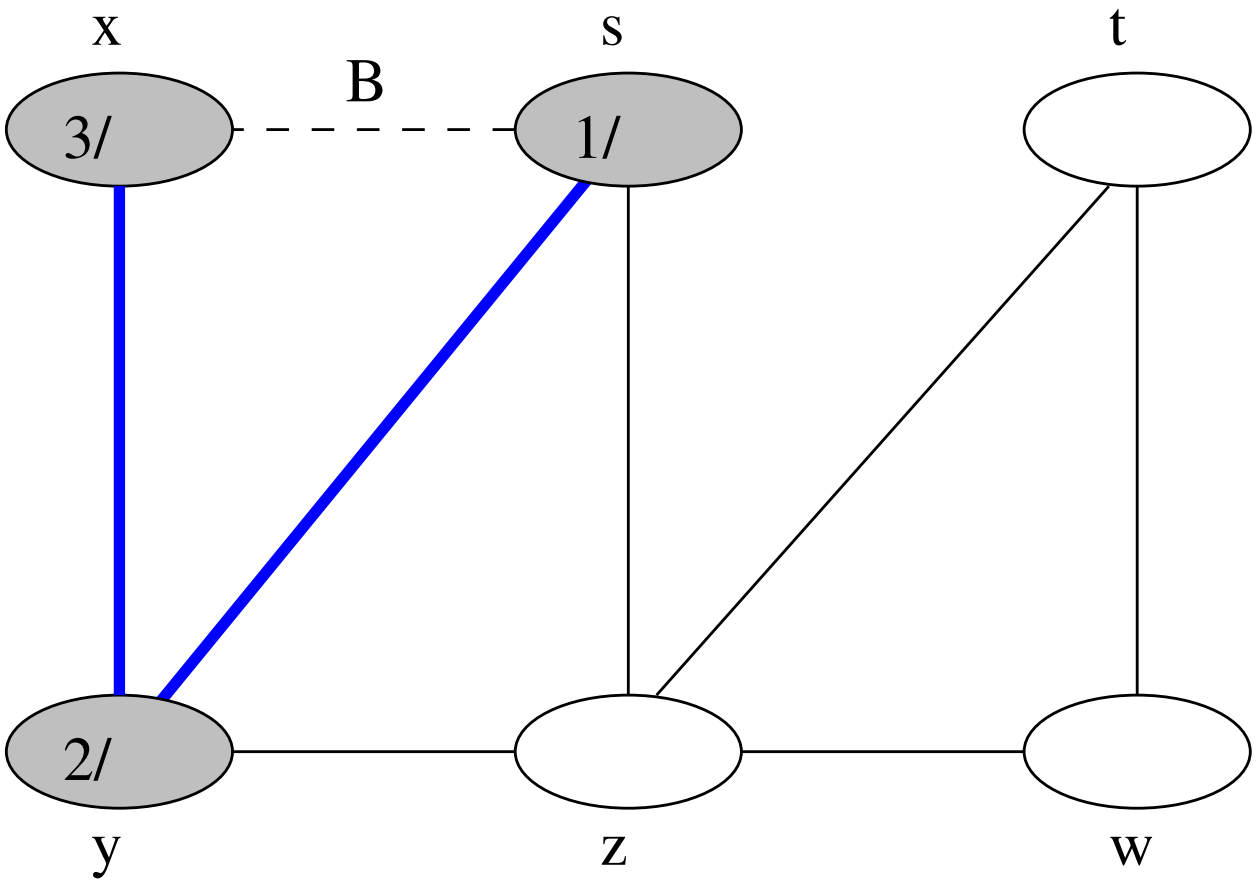
Exemplo



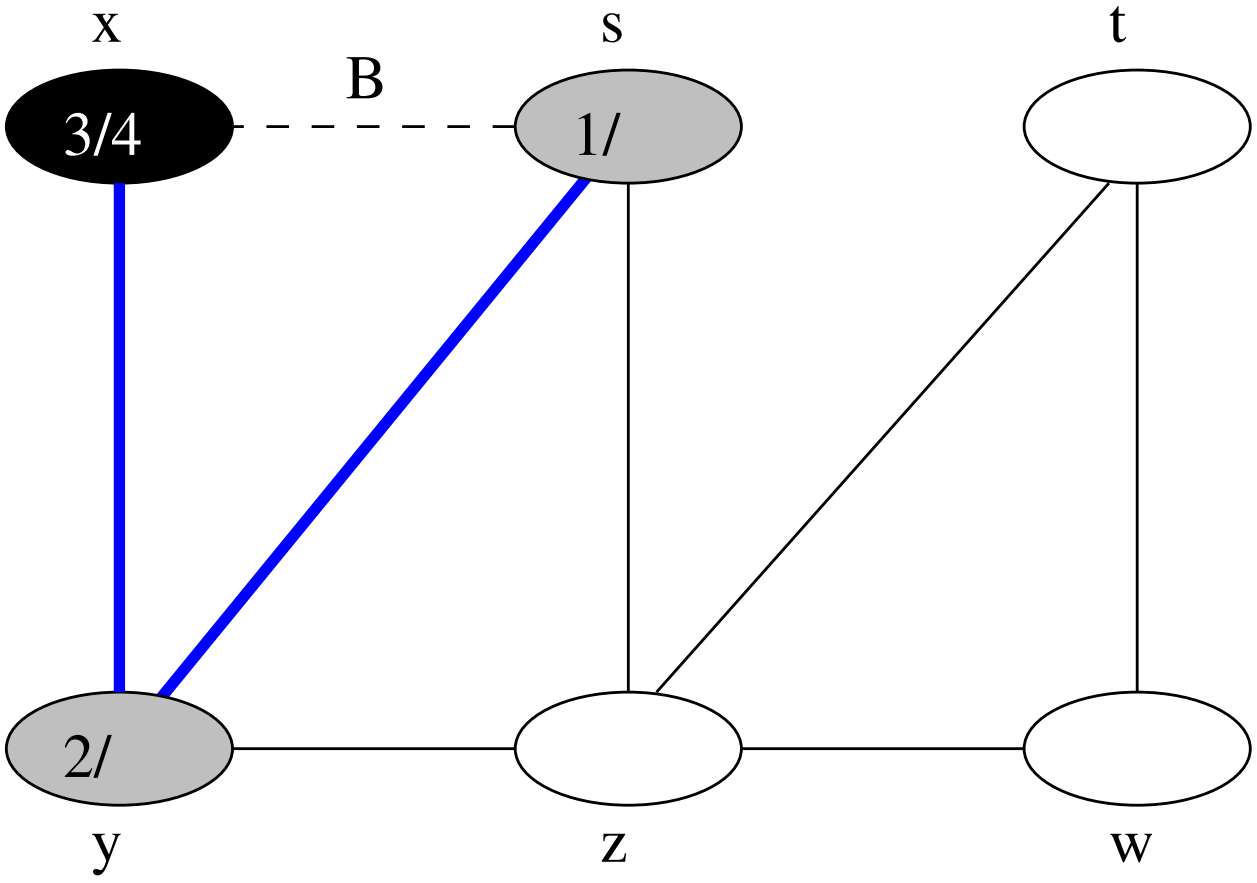
Exemplo



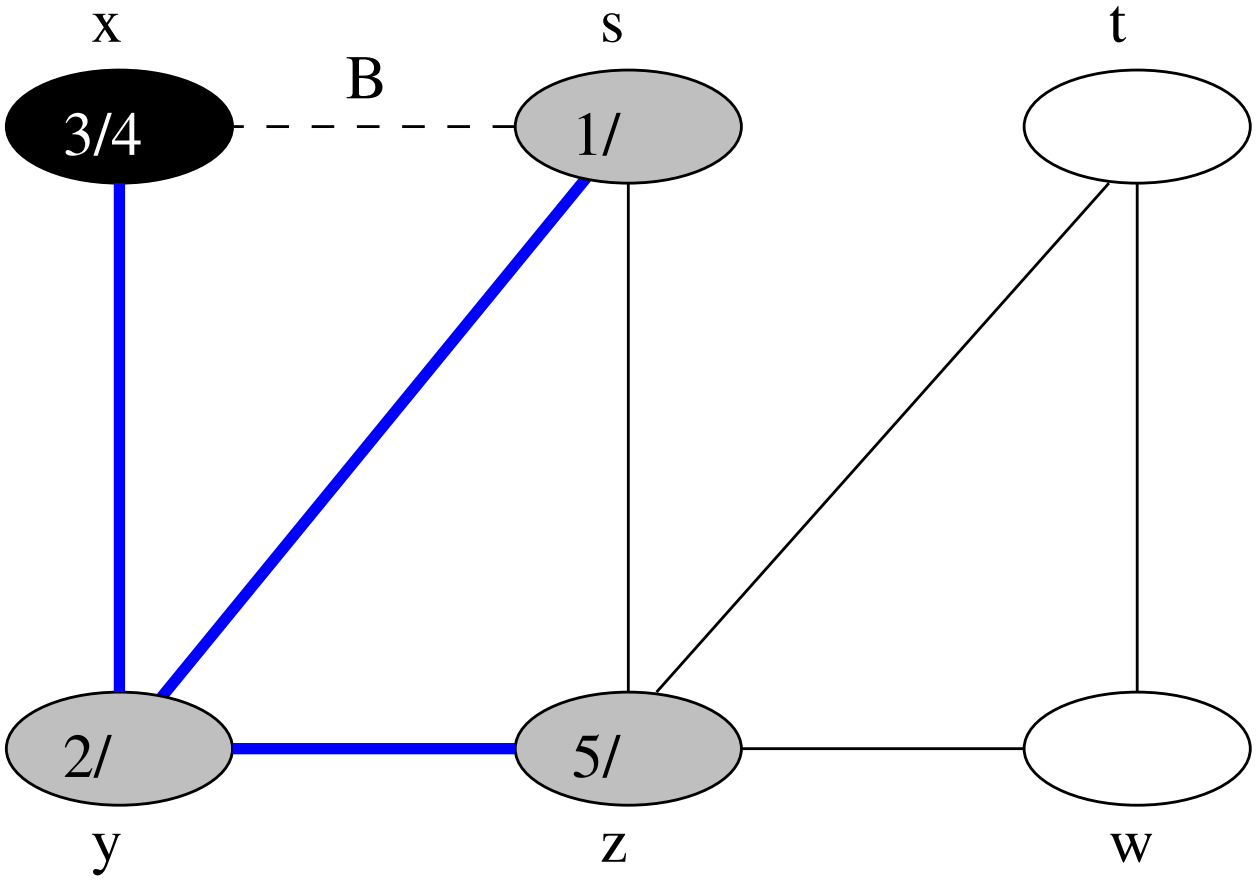
Exemplo



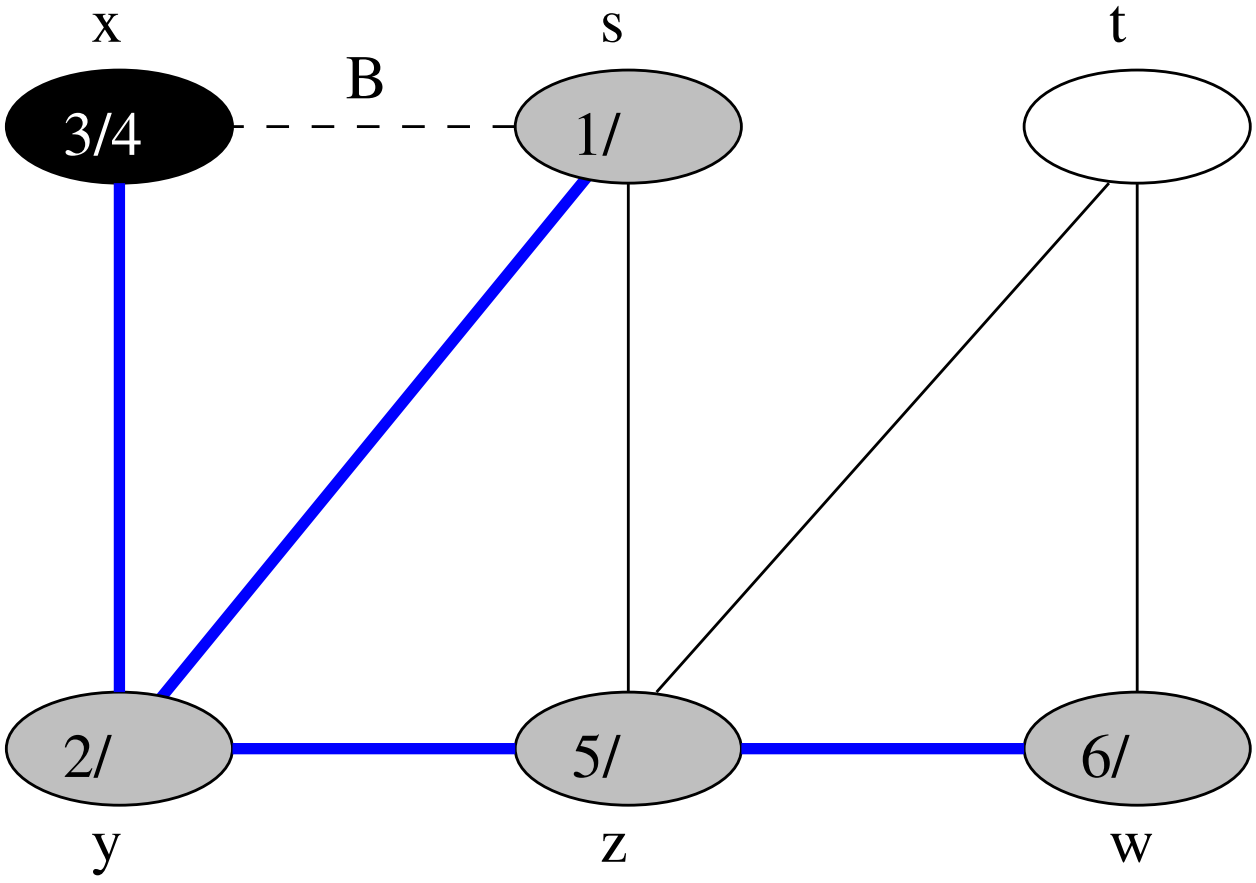
Exemplo



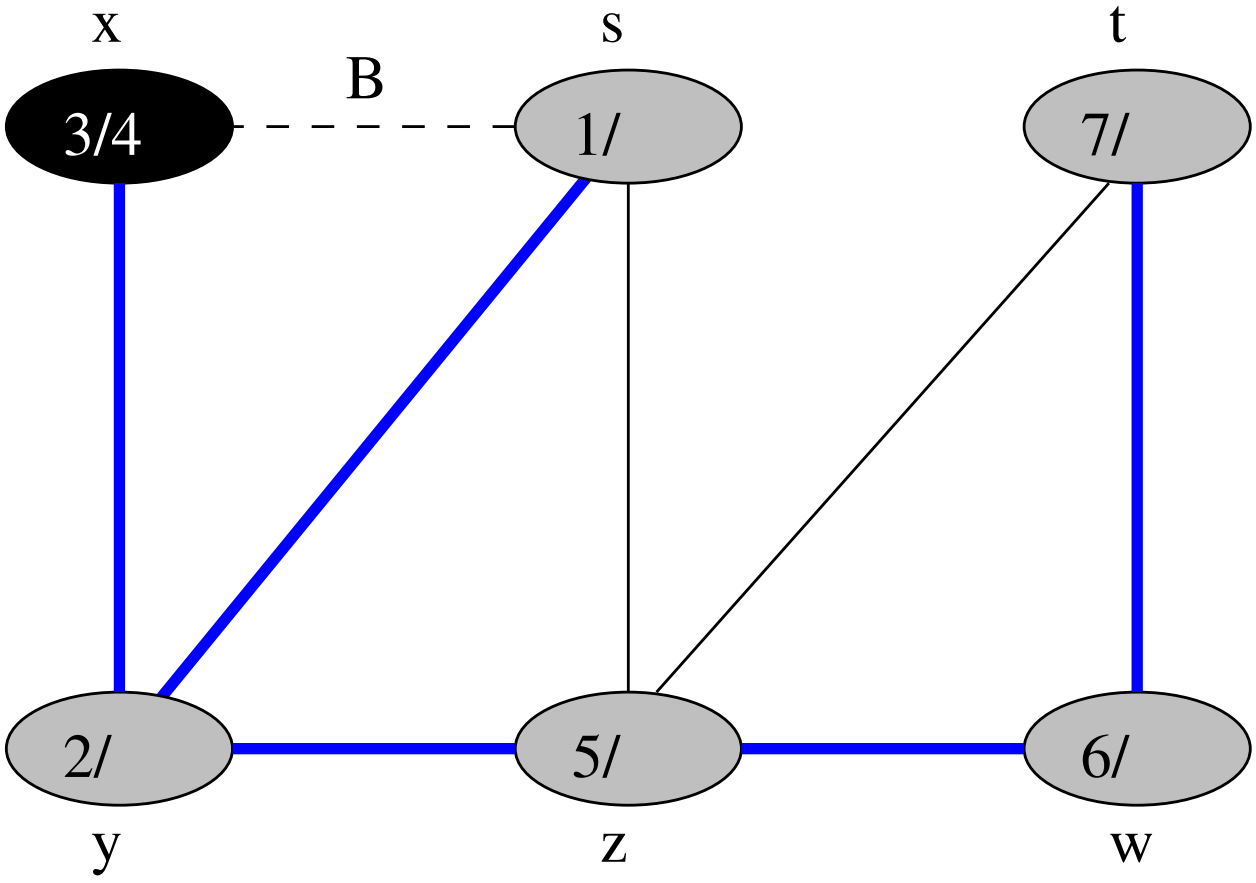
Exemplo



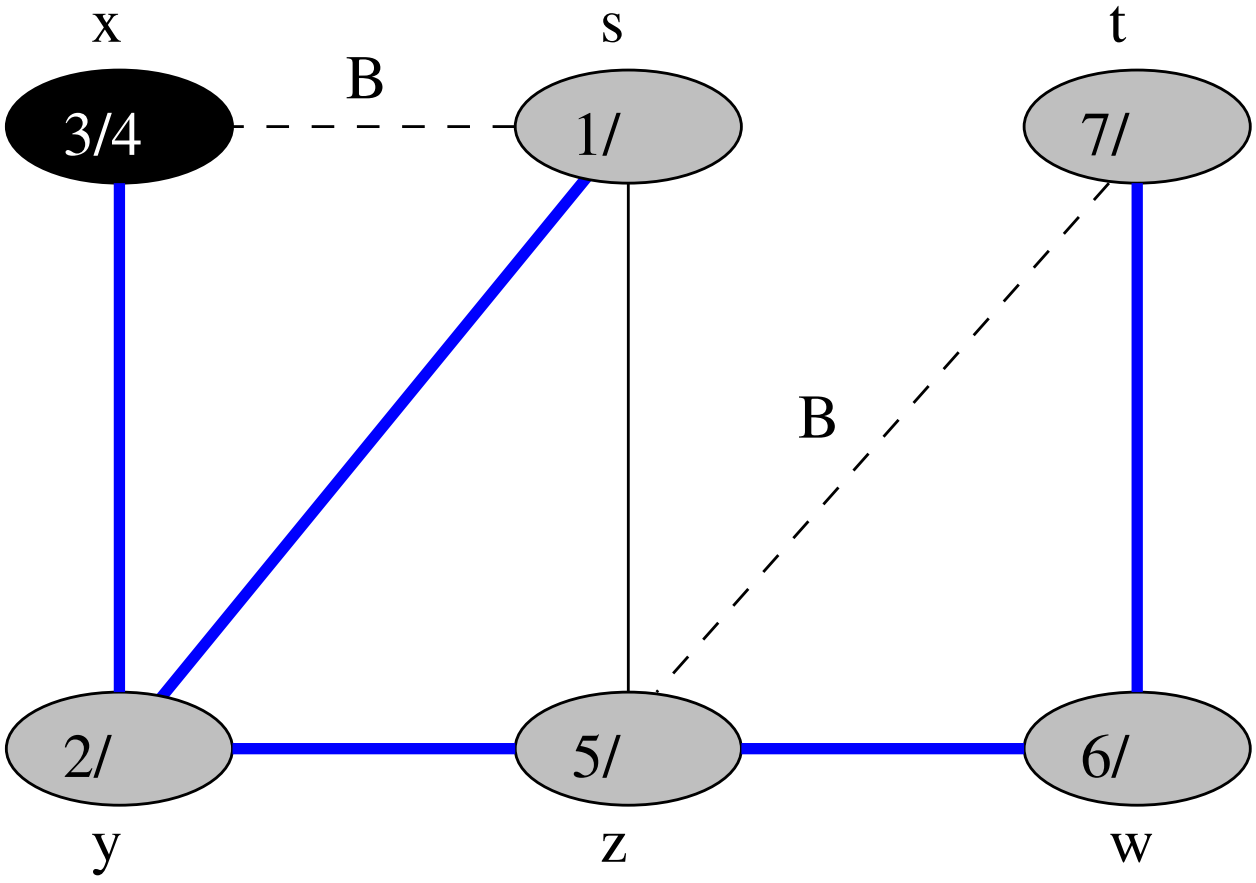
Exemplo



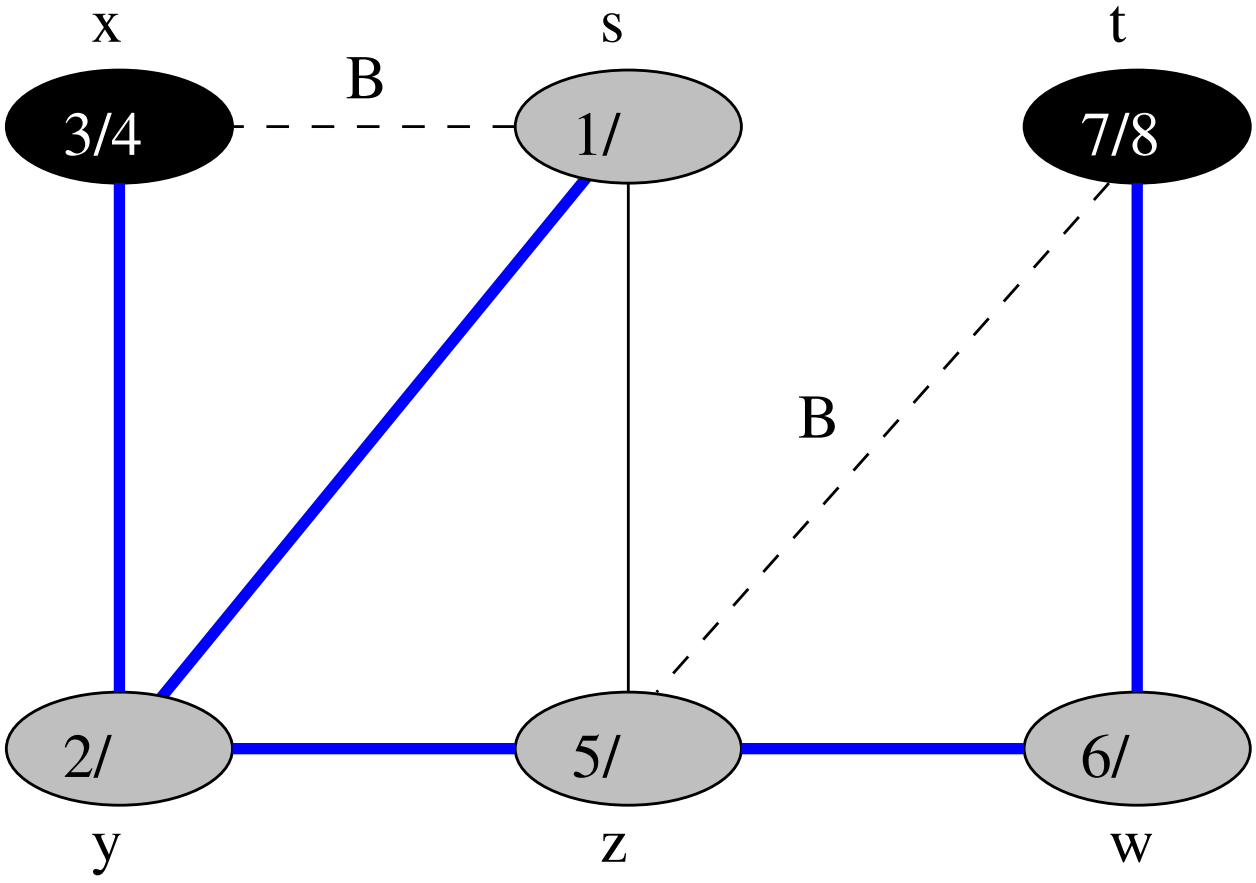
Exemplo



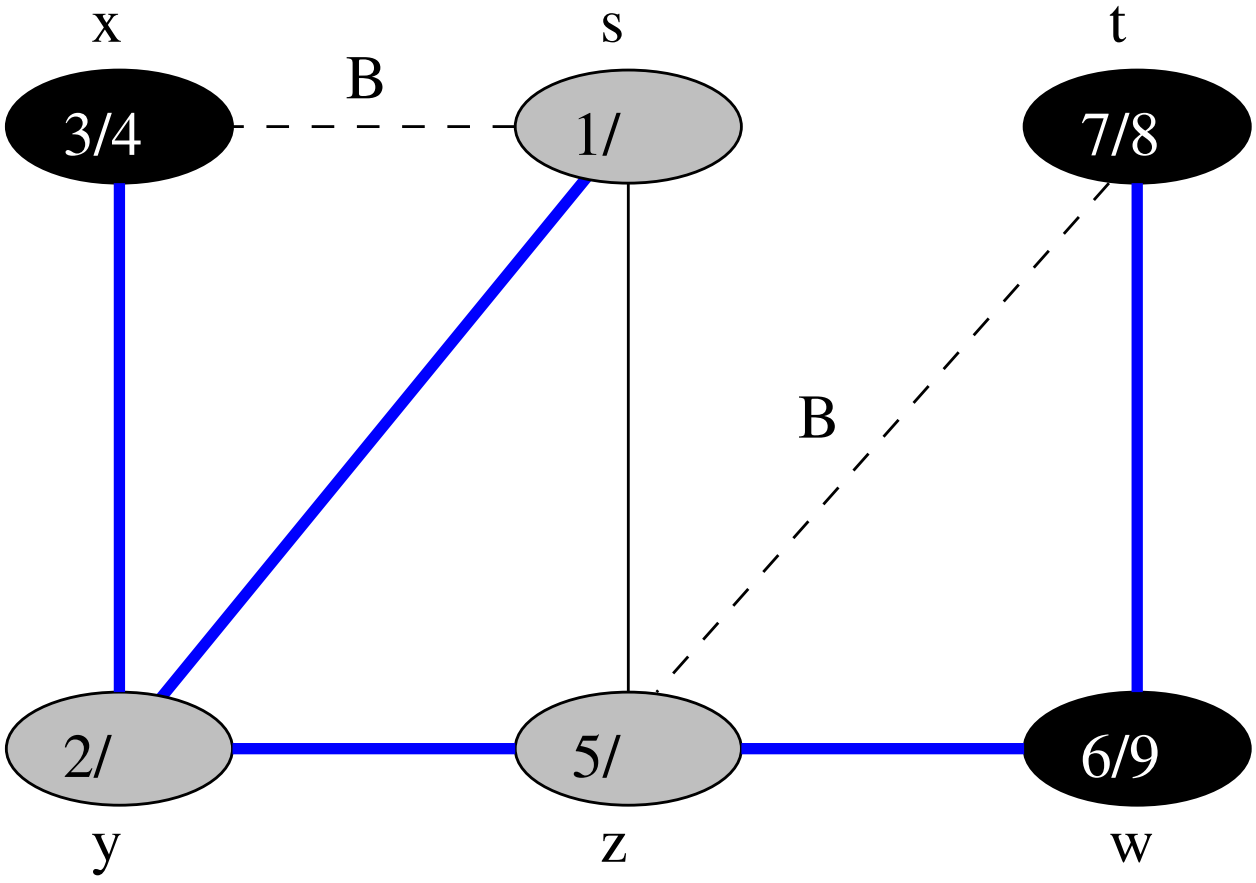
Exemplo



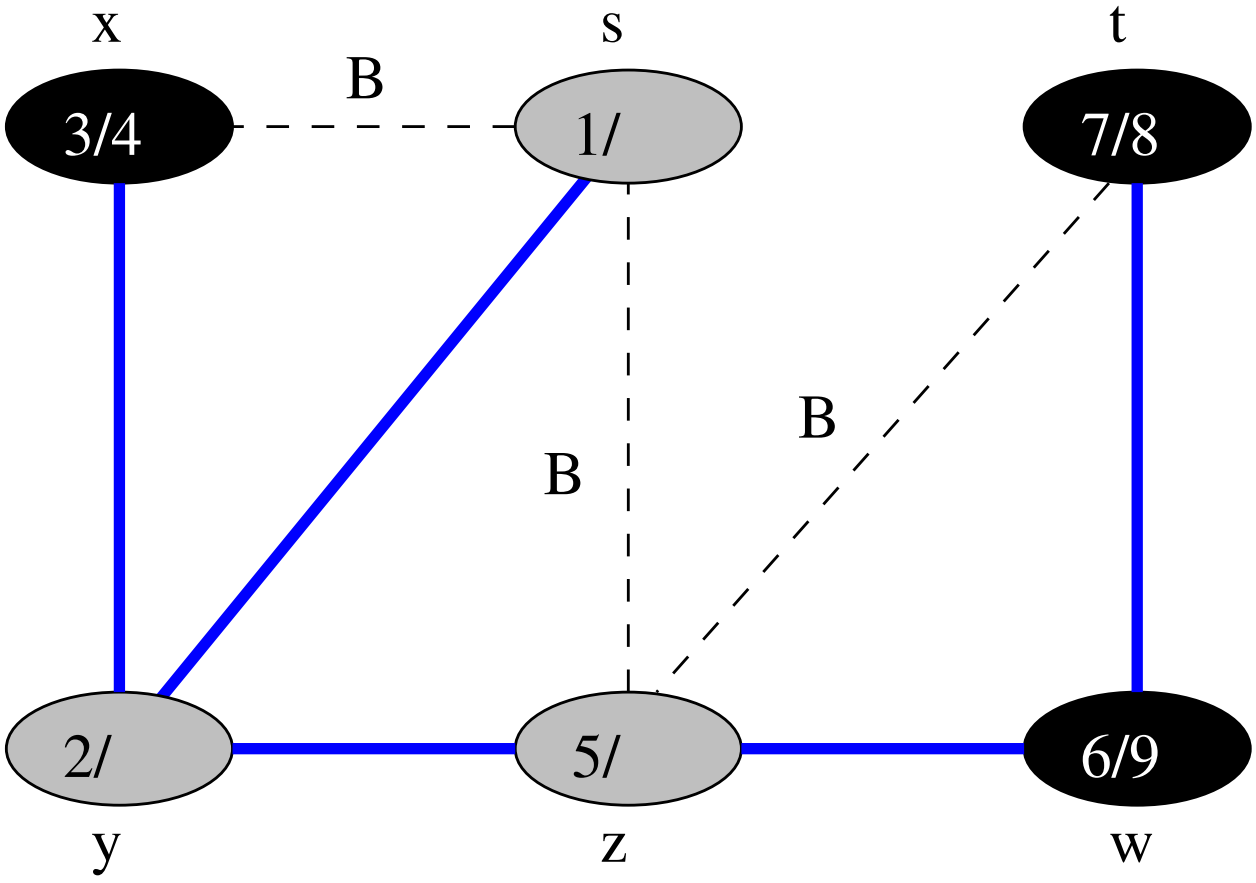
Exemplo



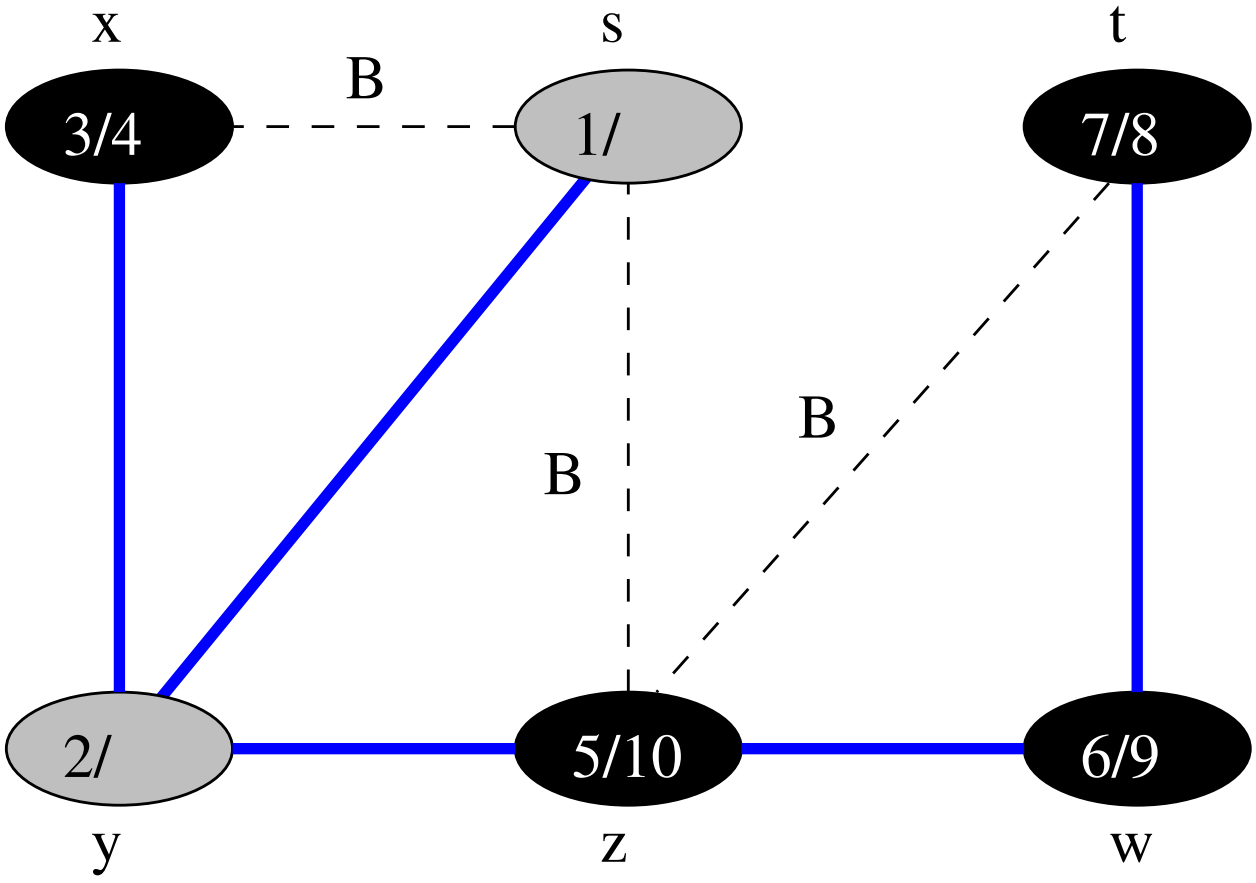
Exemplo



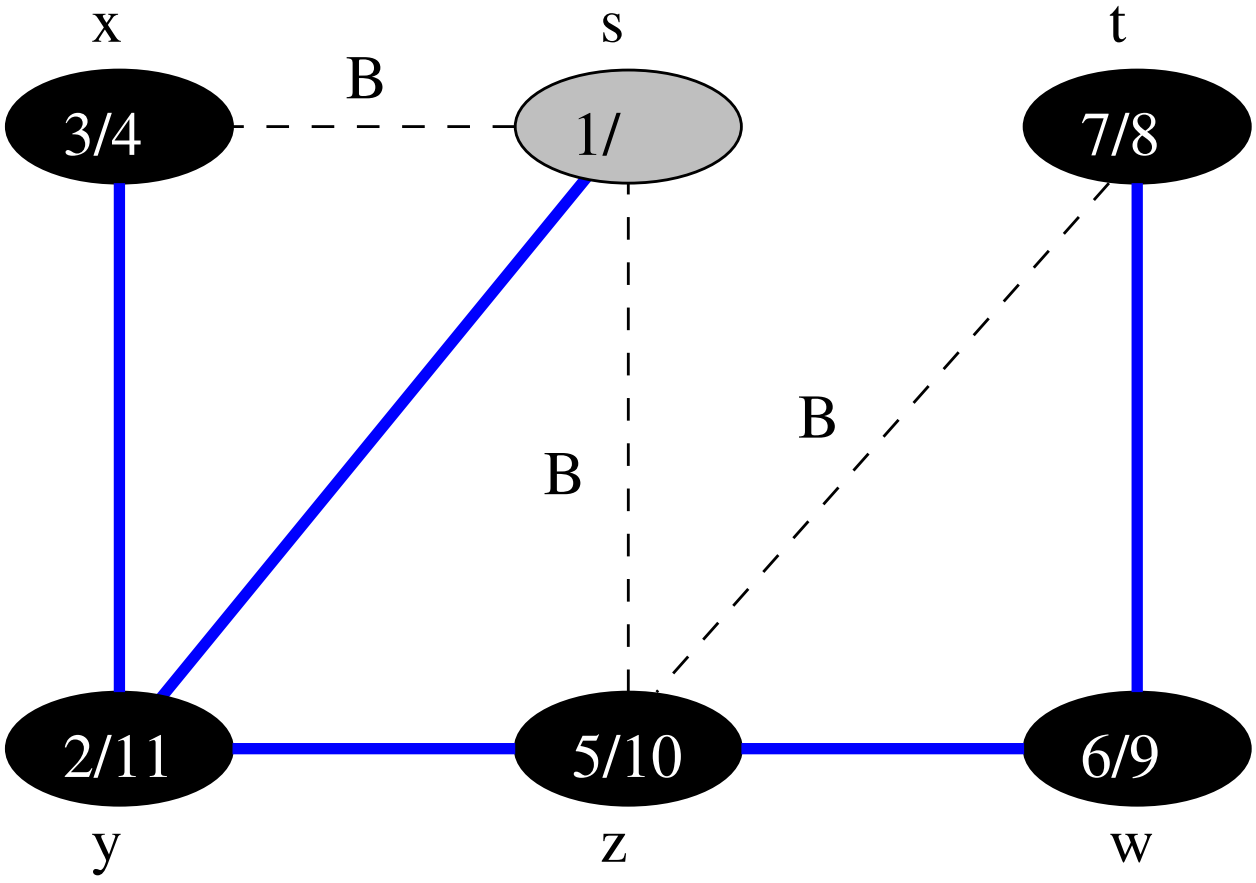
Exemplo



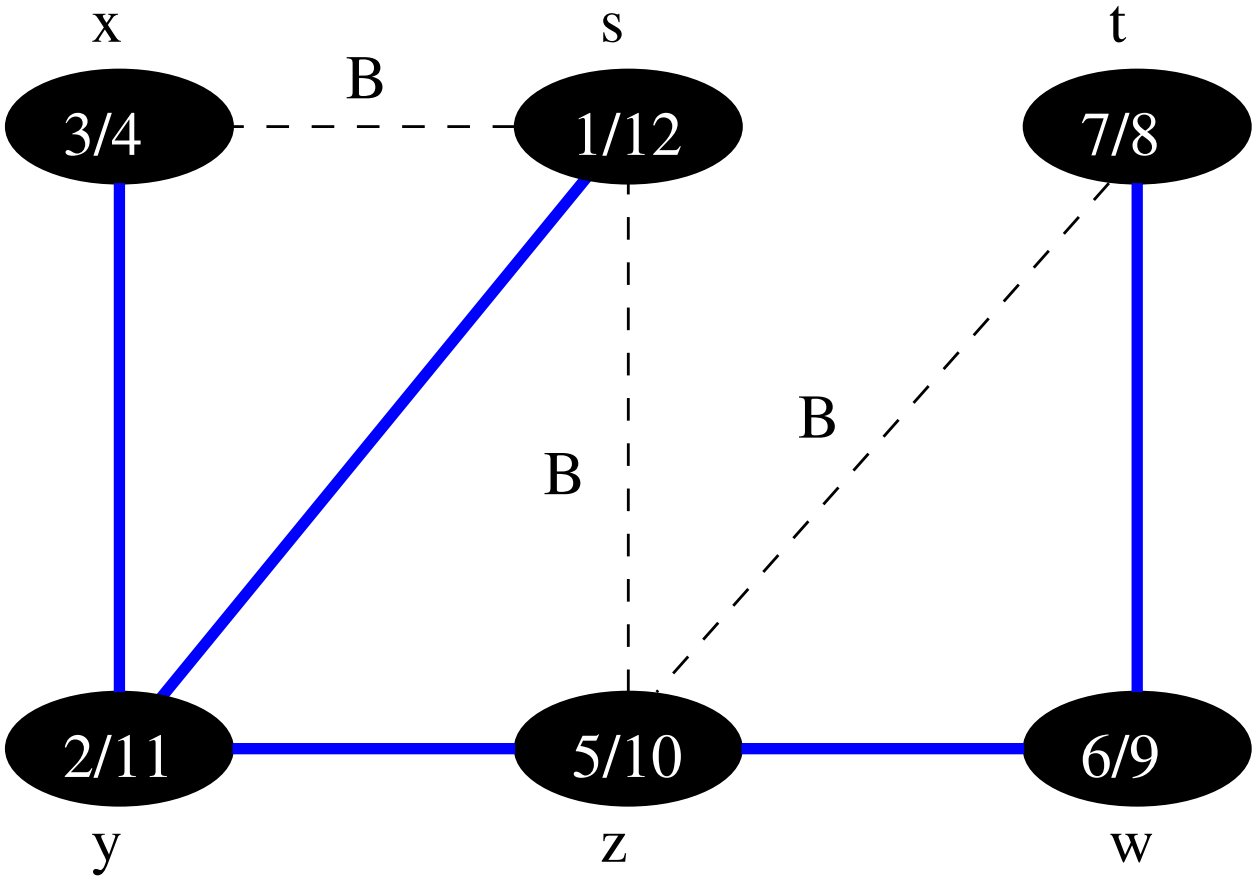
Exemplo



Exemplo



Exemplo

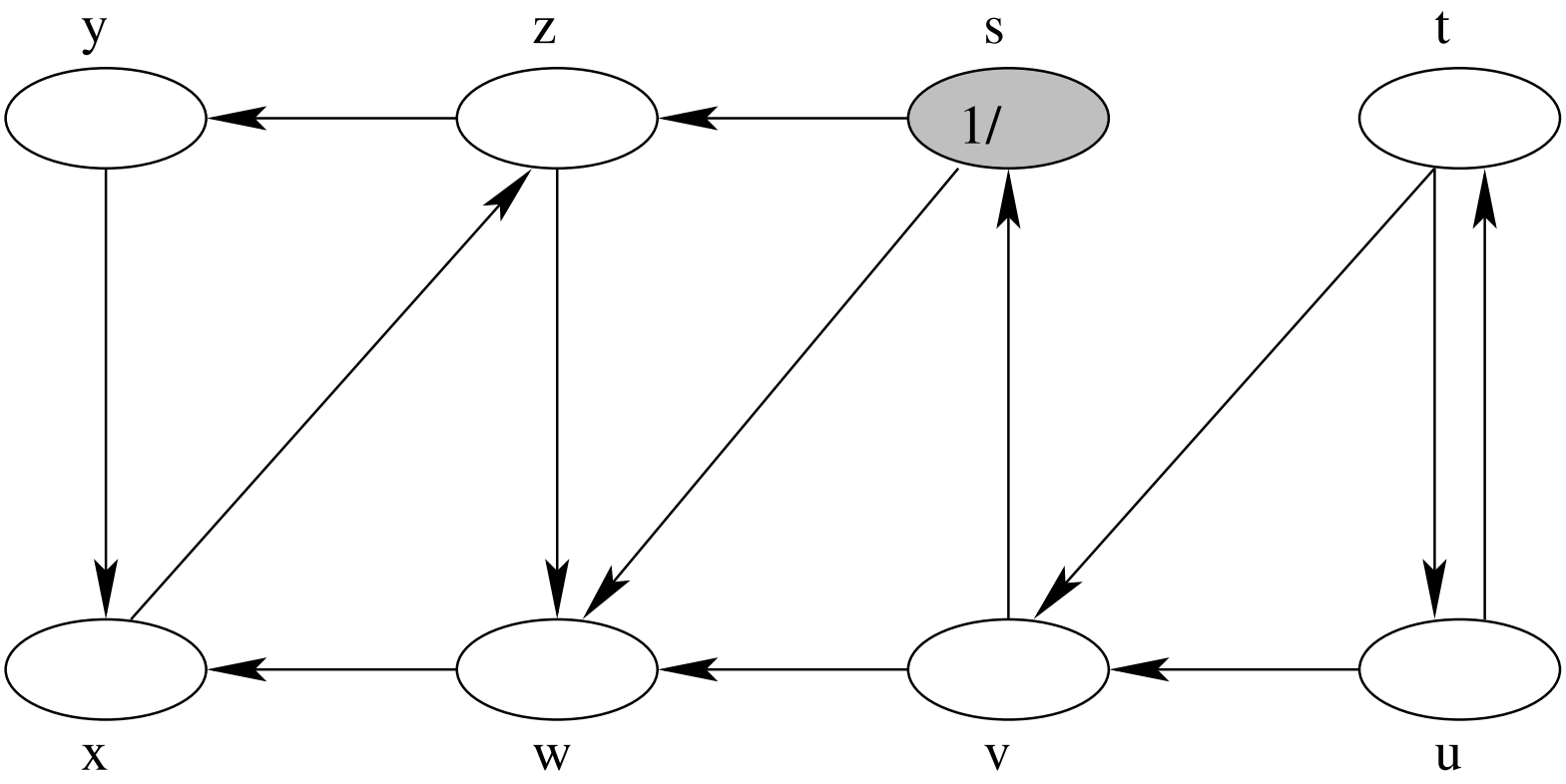


Ordenação Topológica

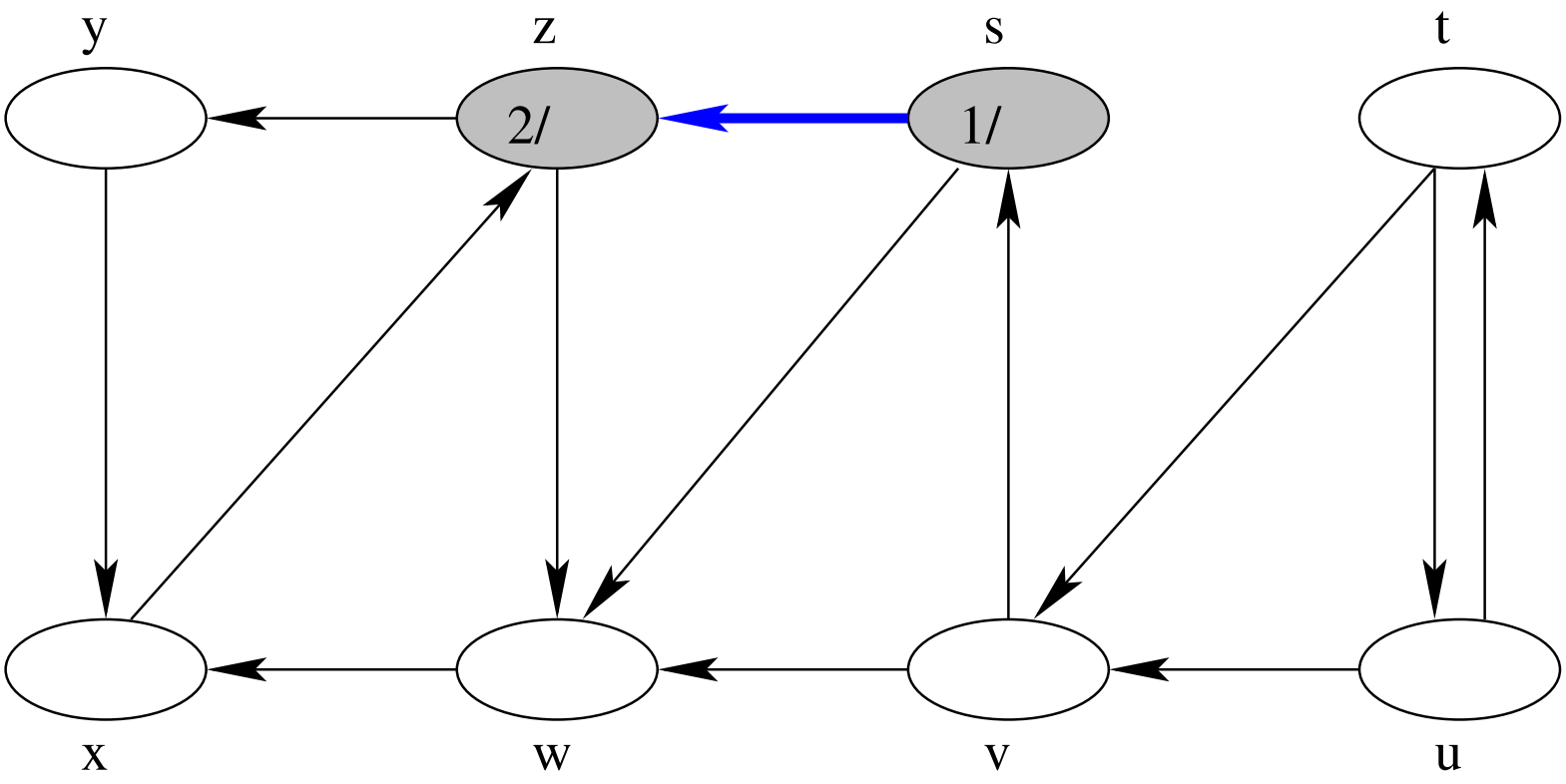
Antes de começar

Vamos rever o nosso exemplo de busca em profundidade em um grafo direcionado.

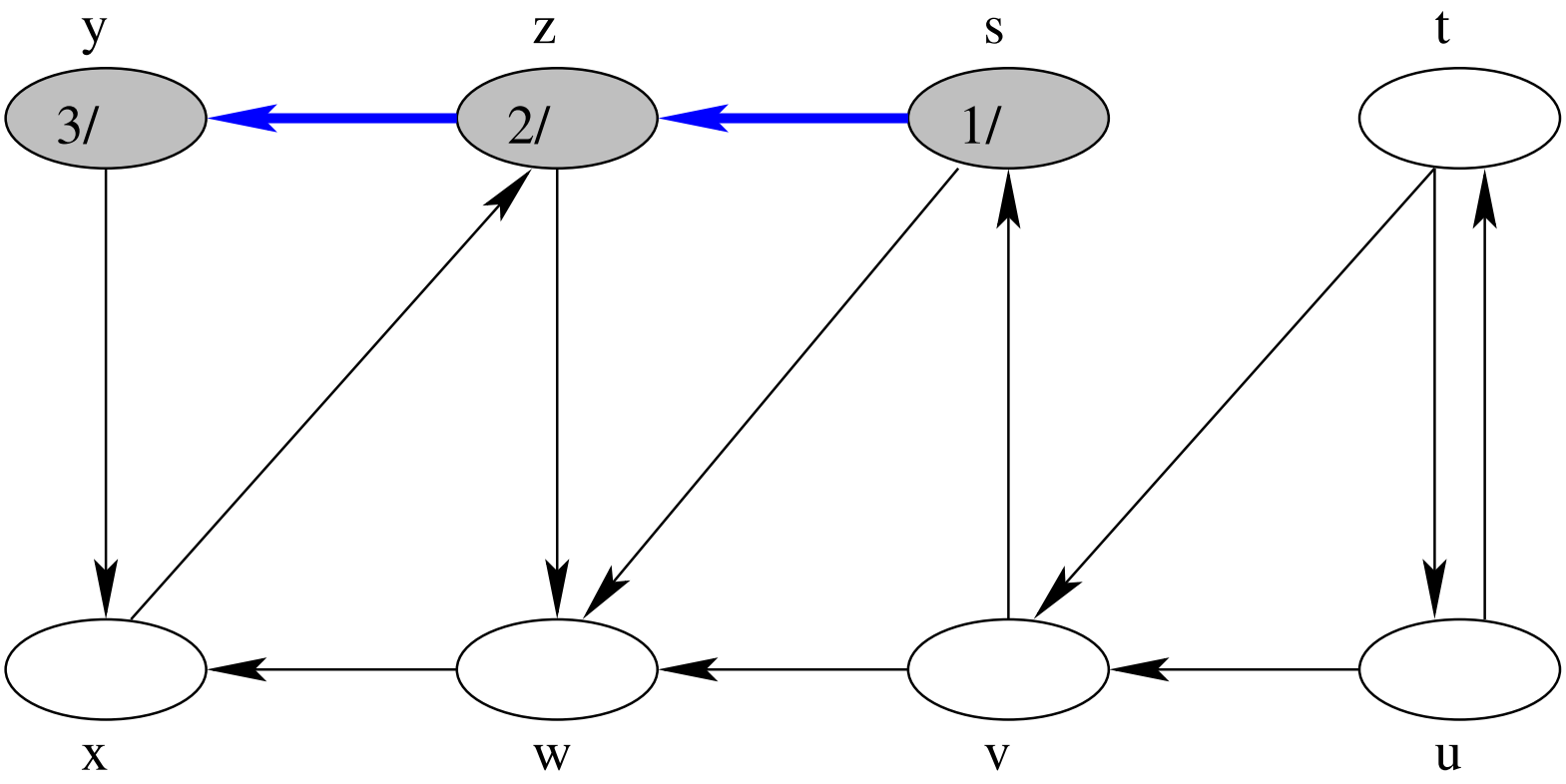
Exemplo DFS



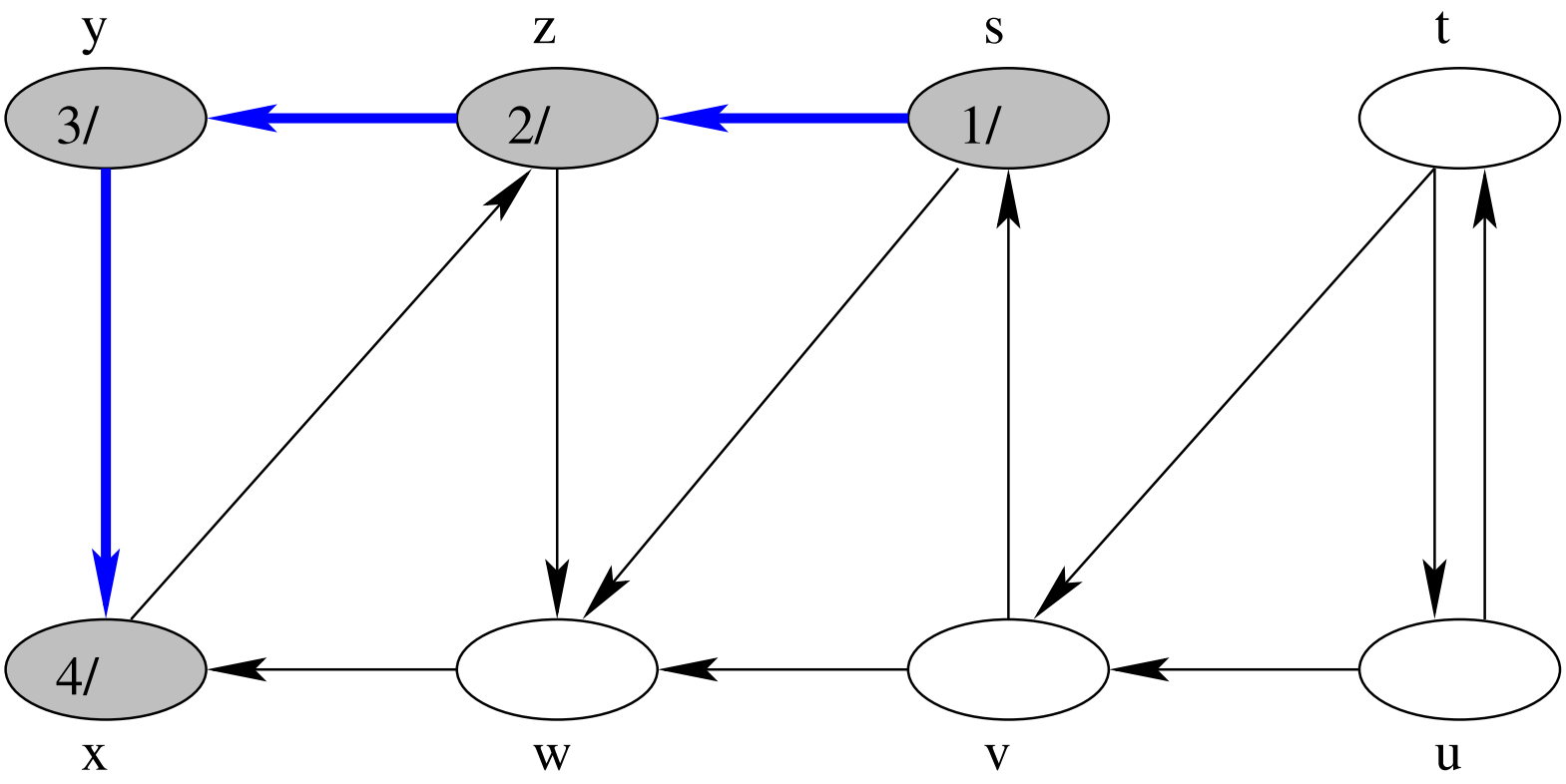
Exemplo DFS



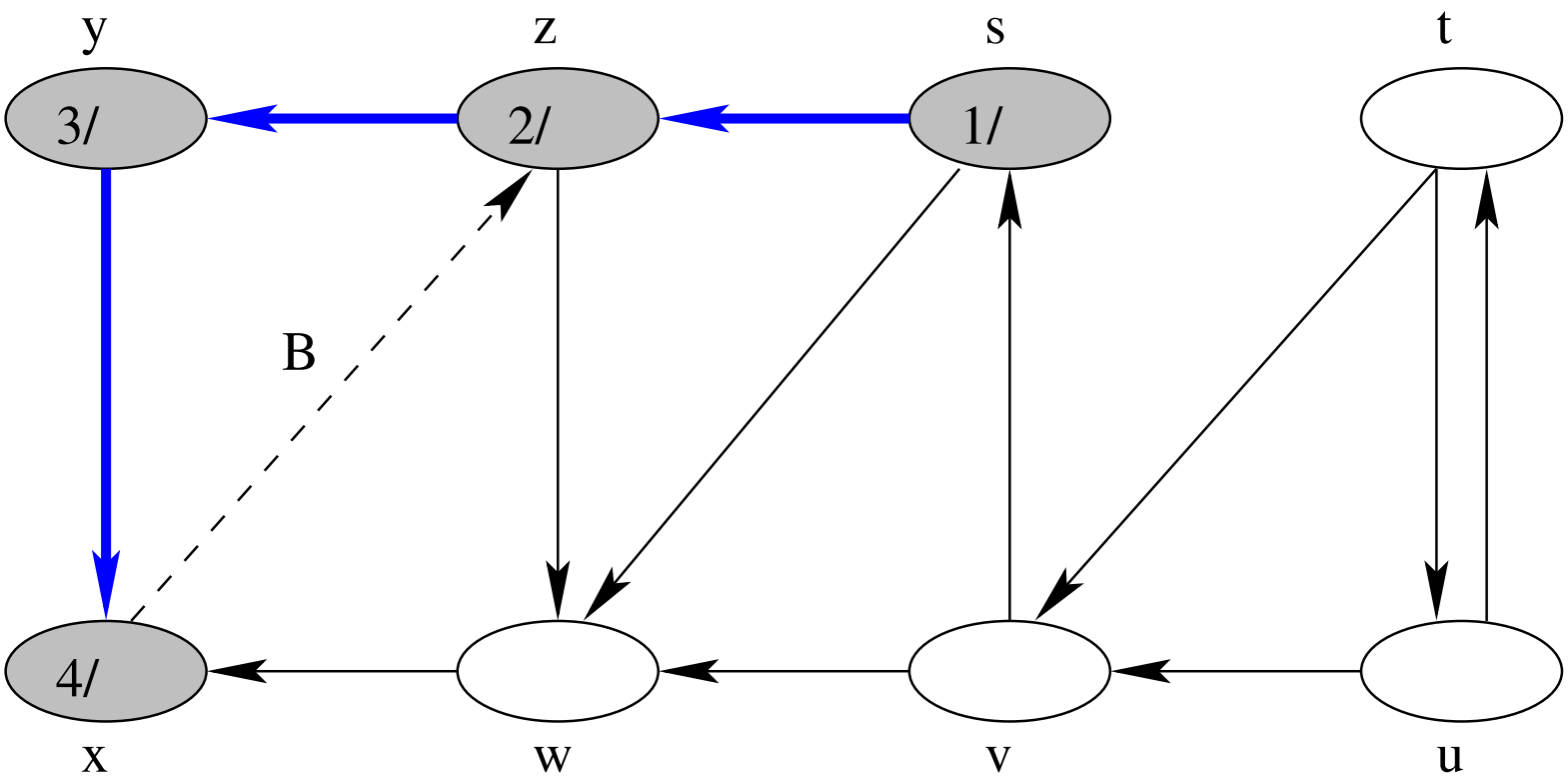
Exemplo DFS



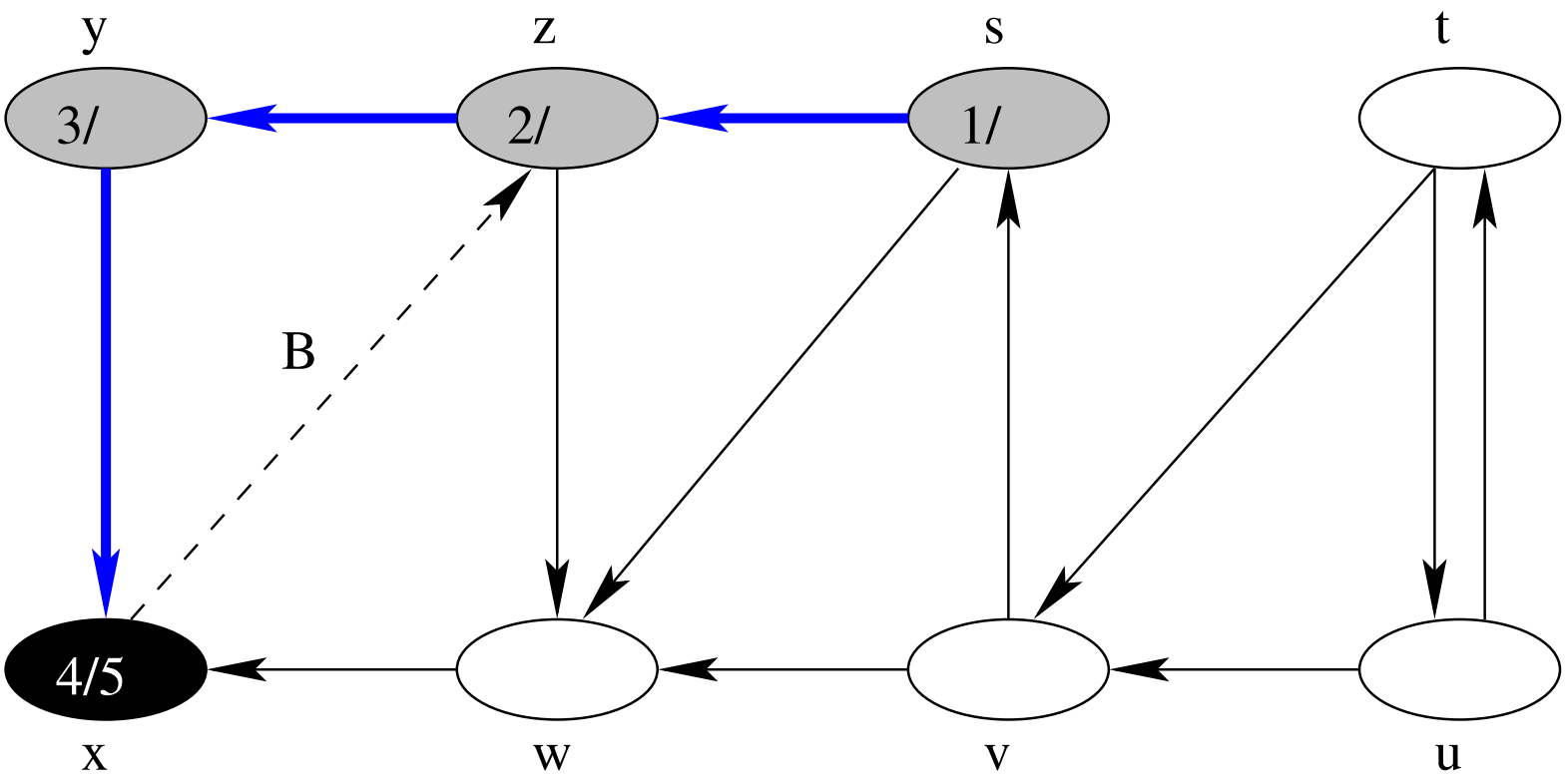
Exemplo DFS



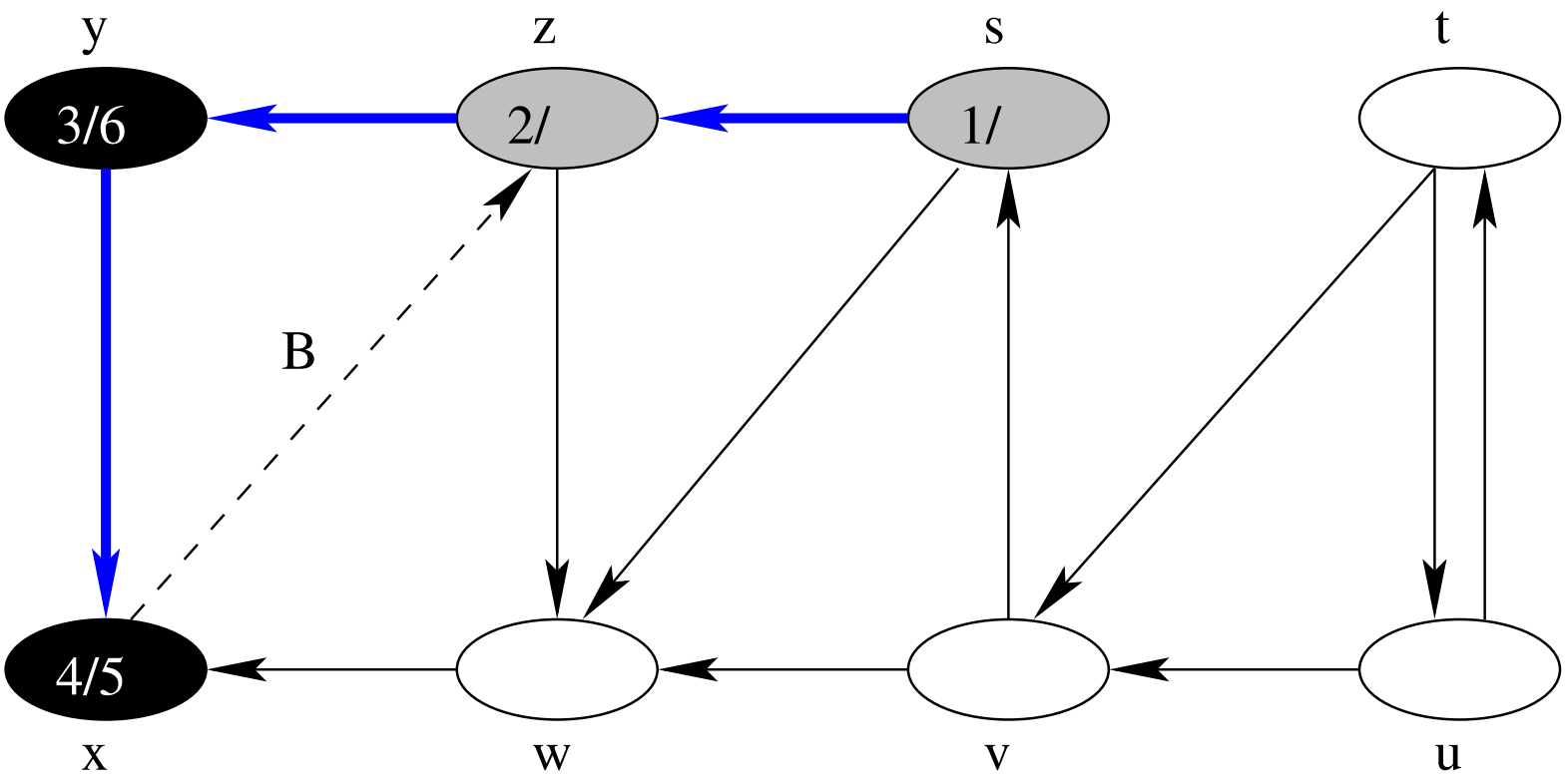
Exemplo DFS



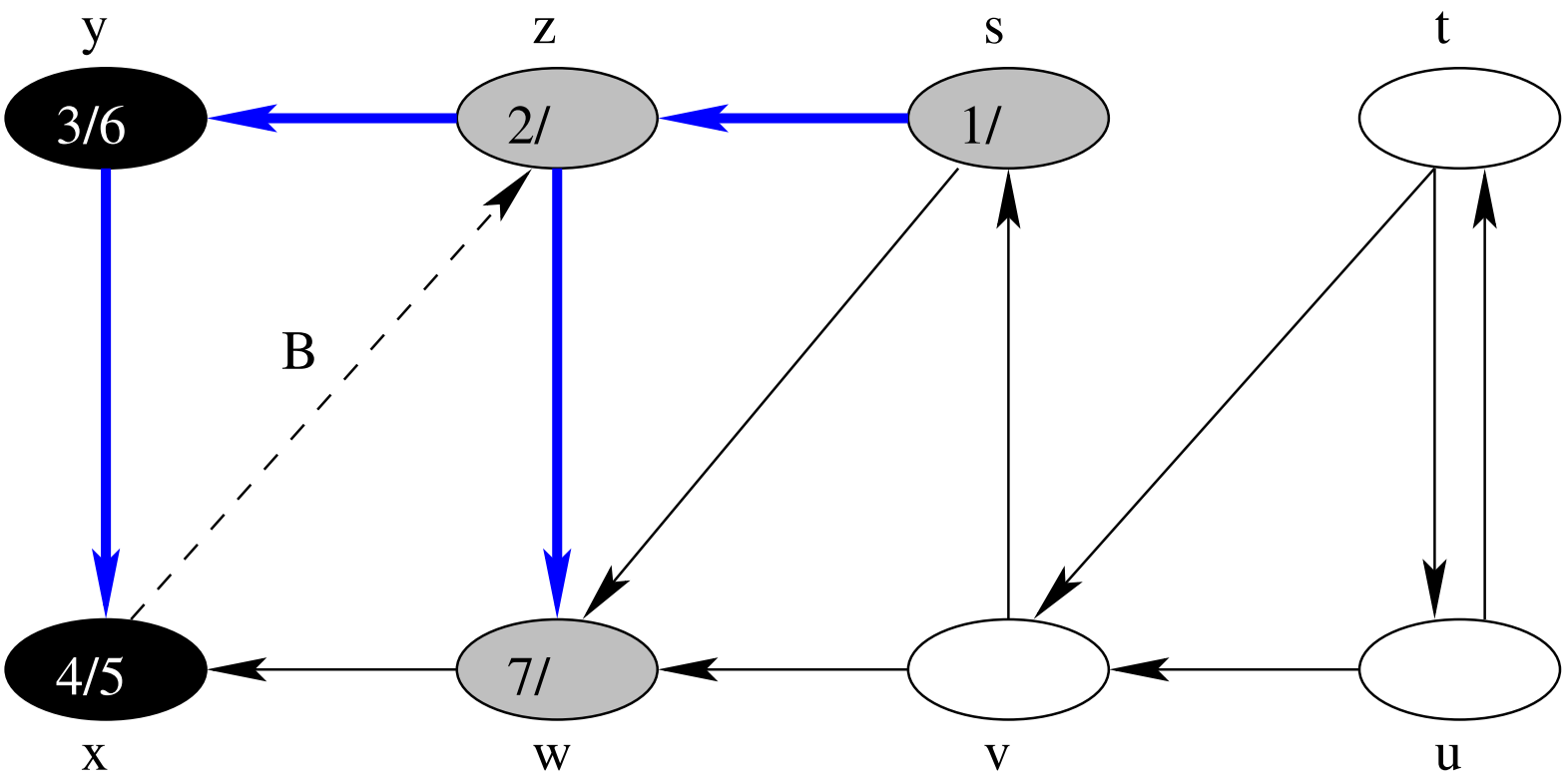
Exemplo DFS



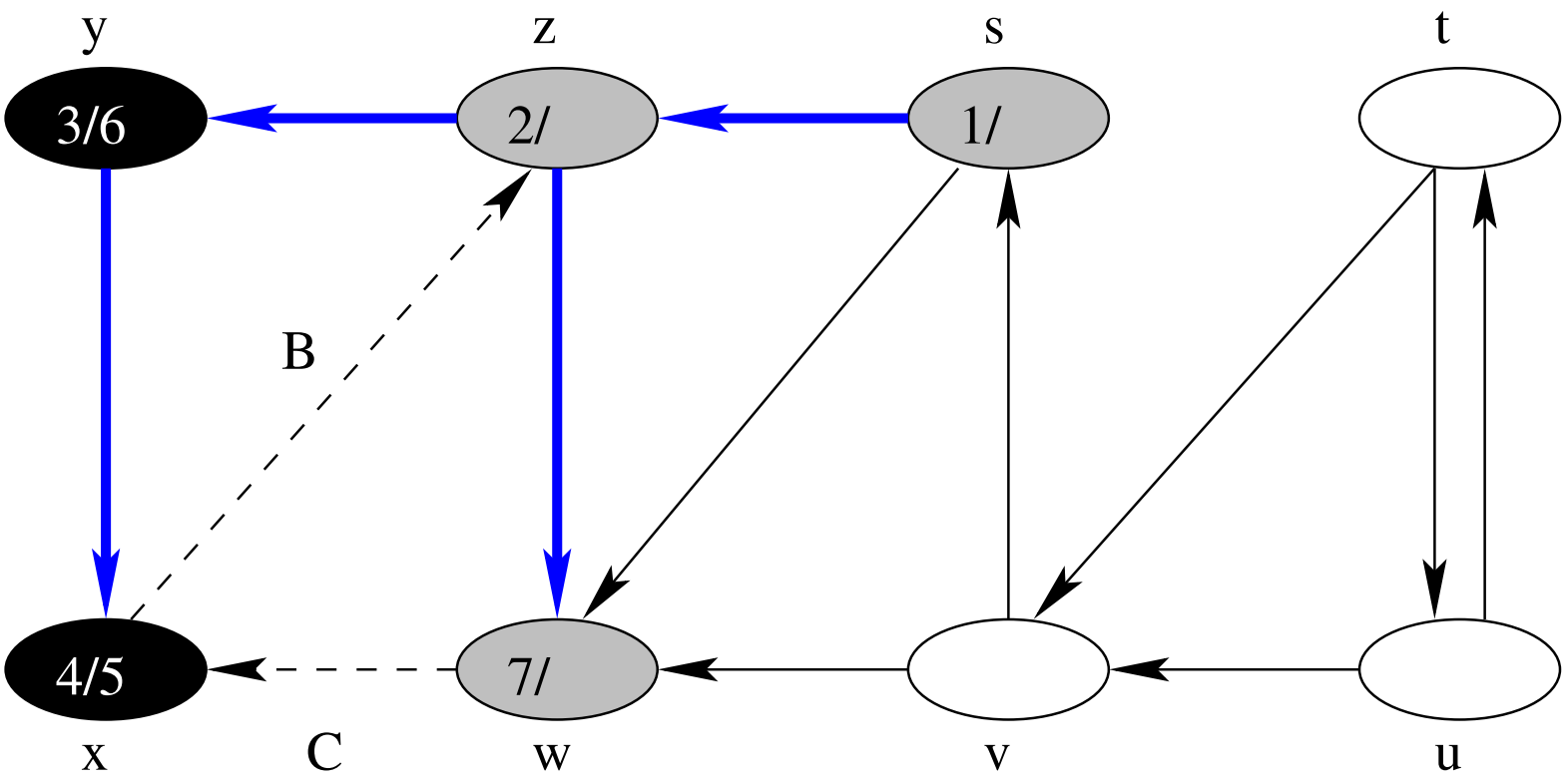
Exemplo DFS



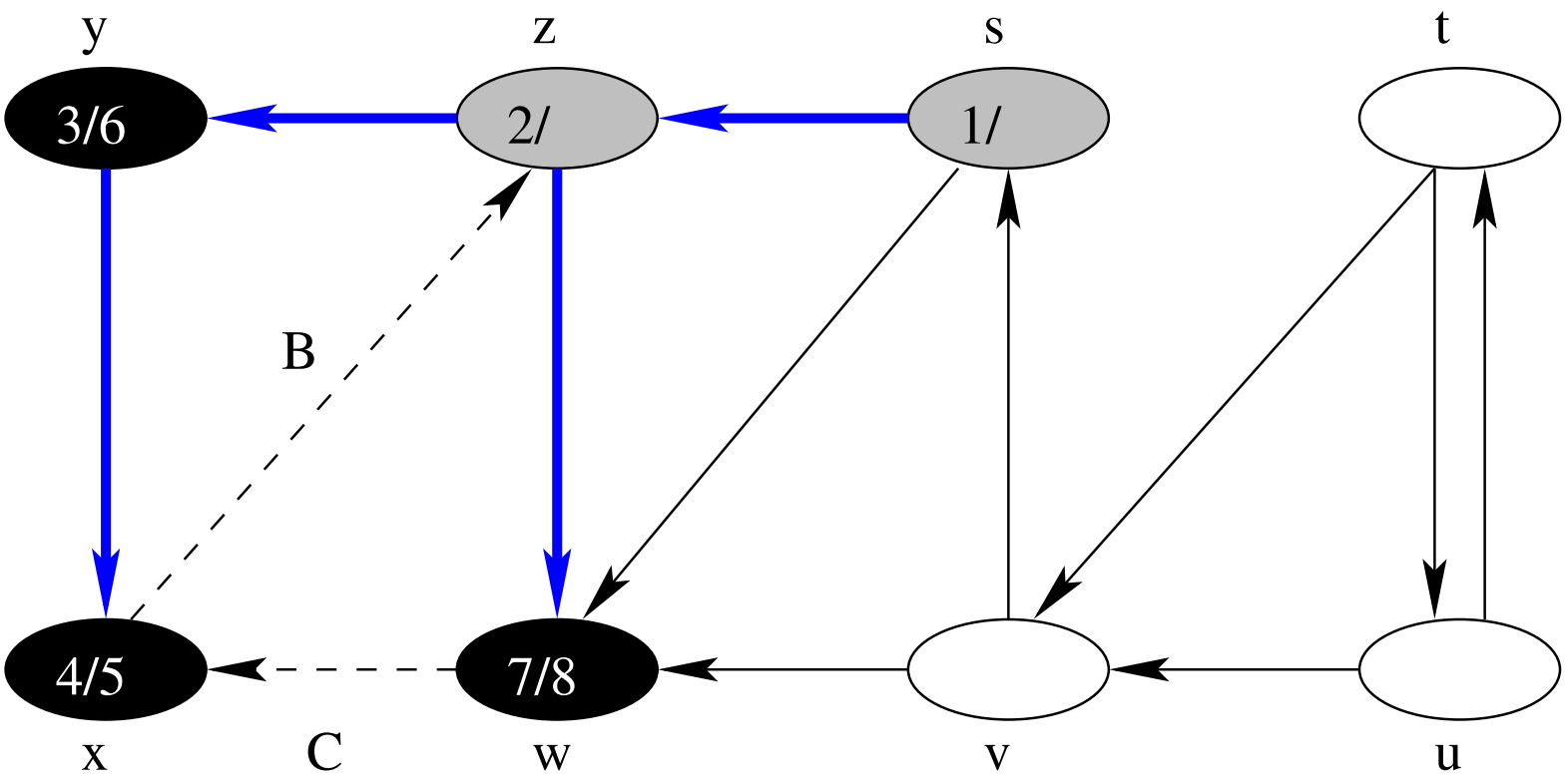
Exemplo DFS



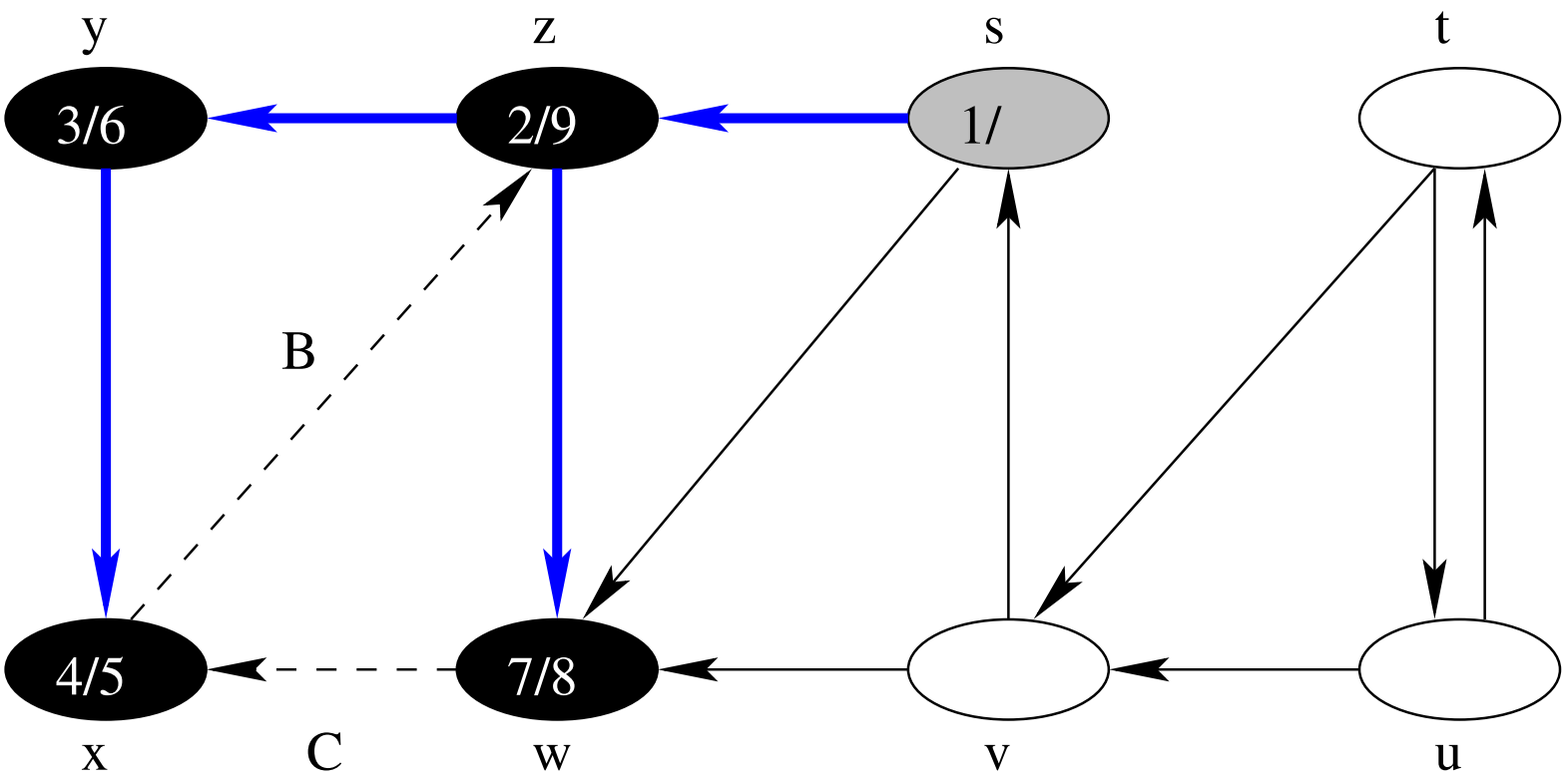
Exemplo DFS



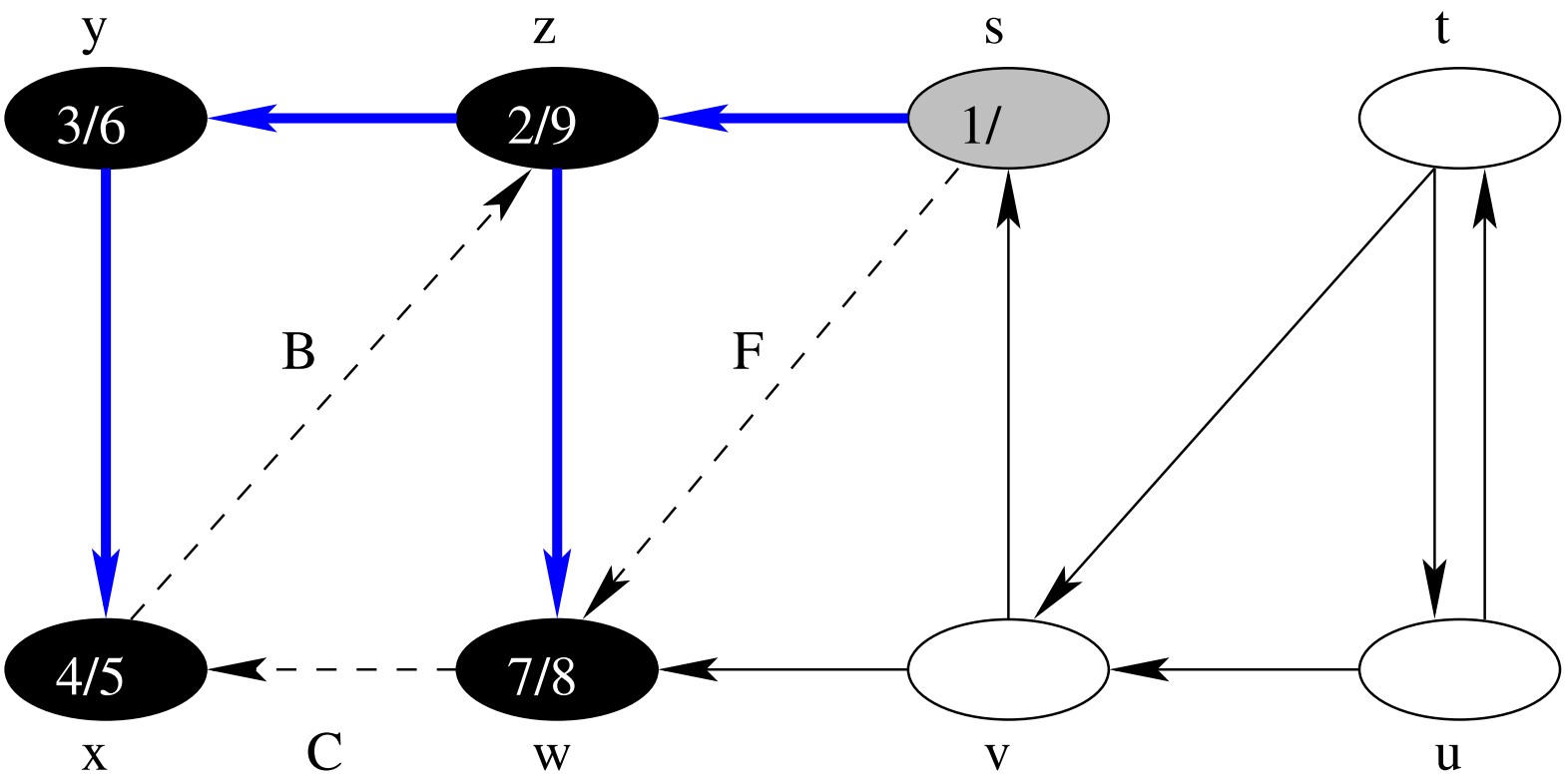
Exemplo DFS



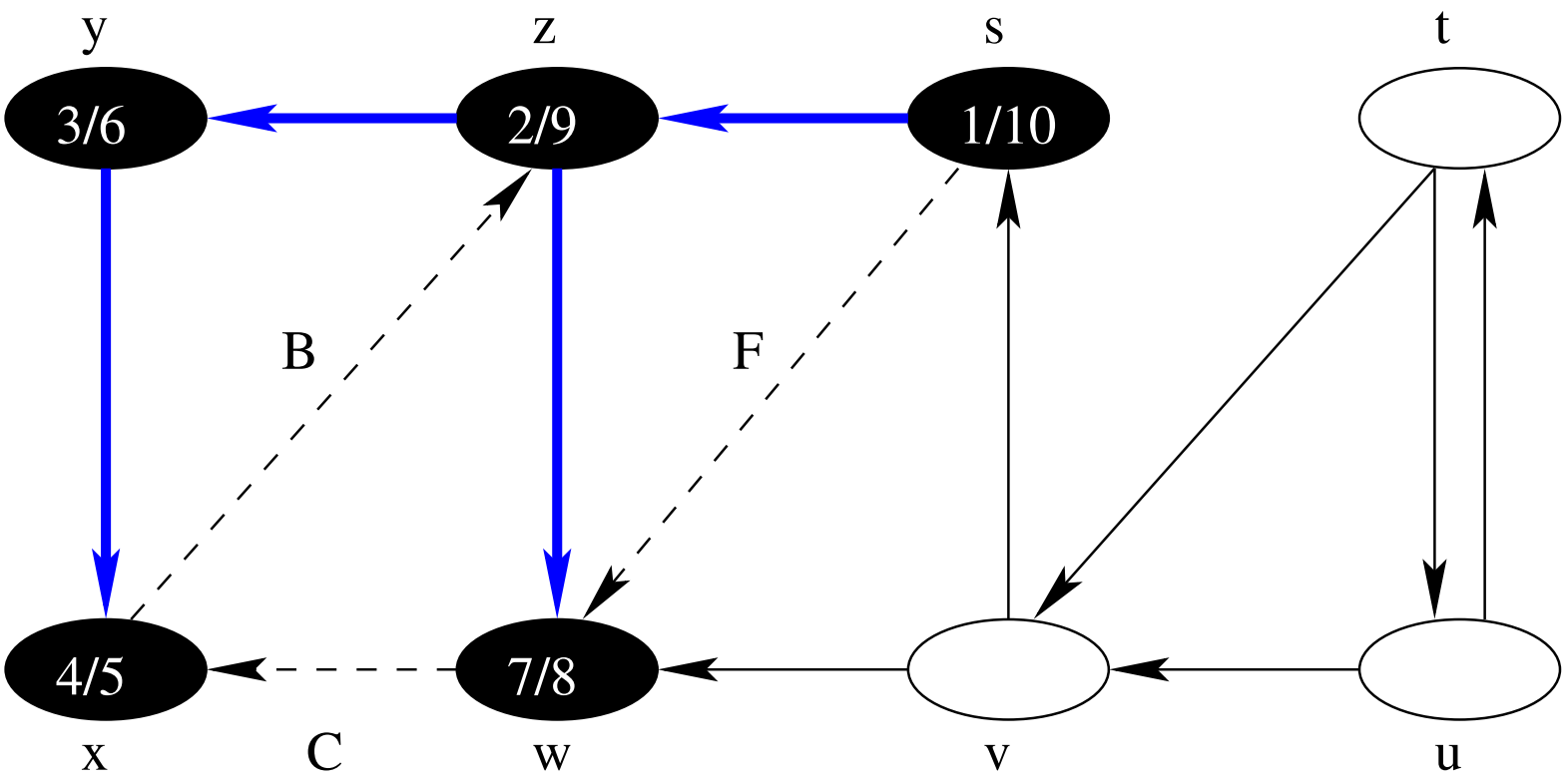
Exemplo DFS



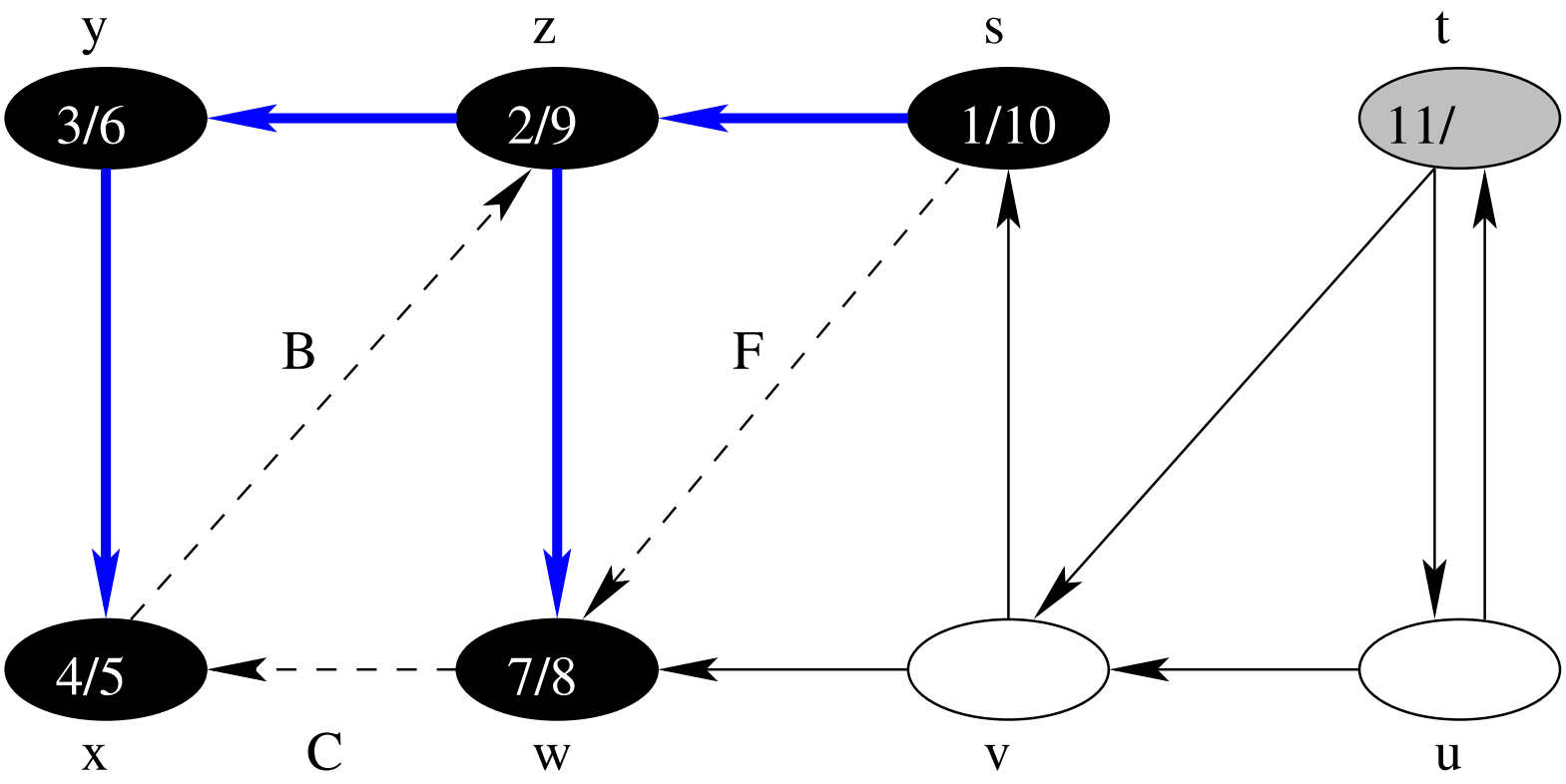
Exemplo DFS



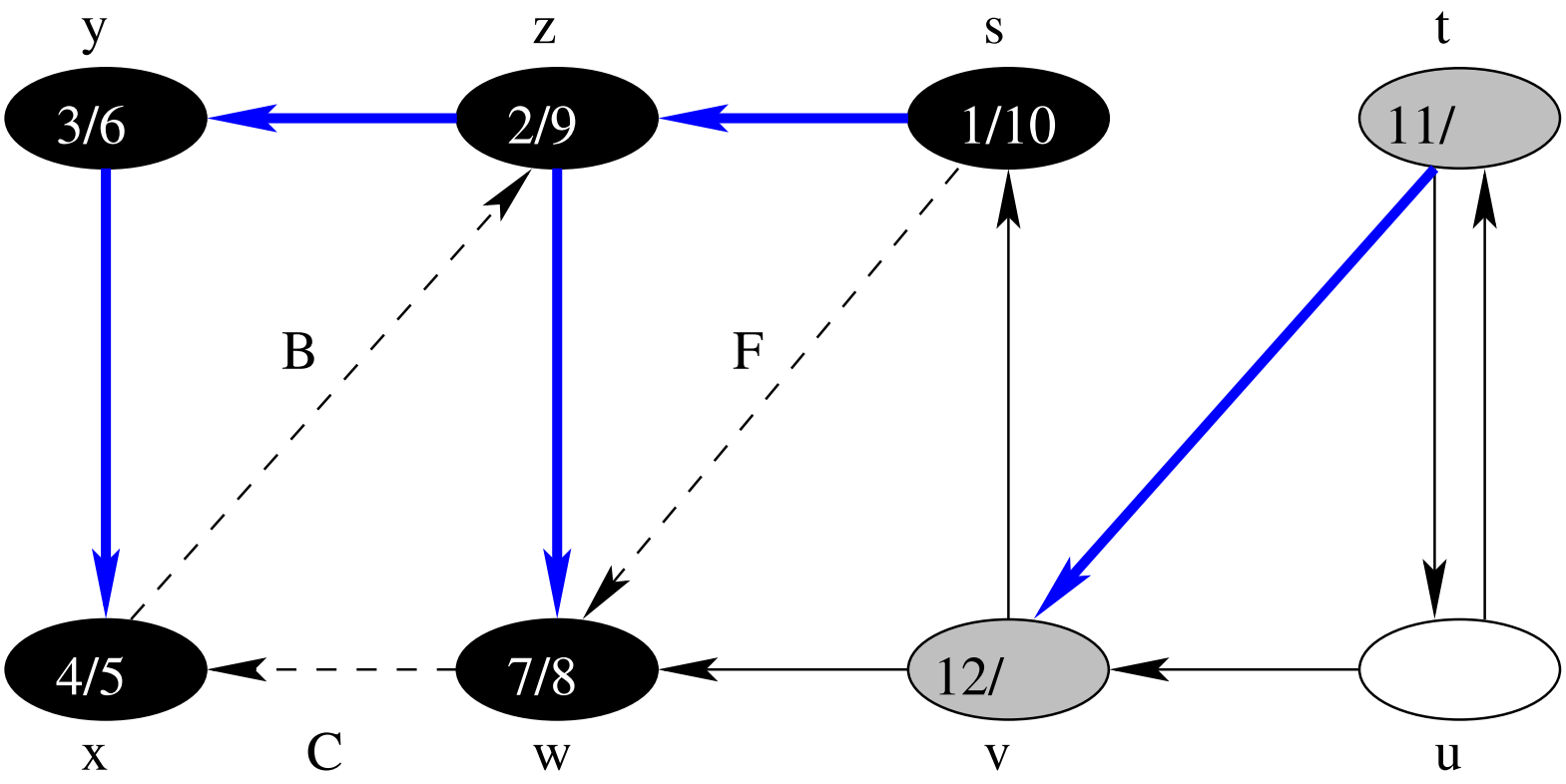
Exemplo DFS



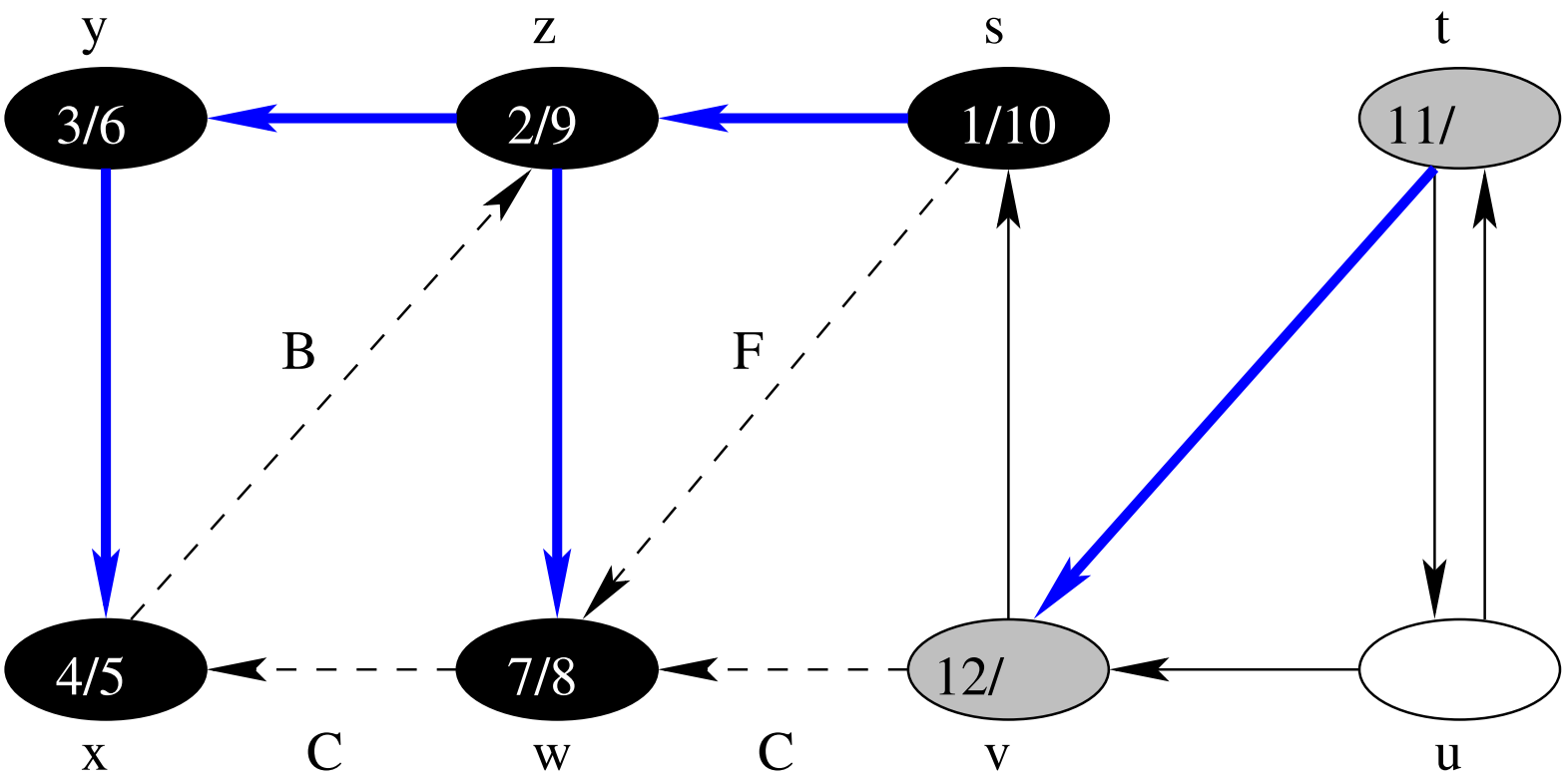
Exemplo DFS



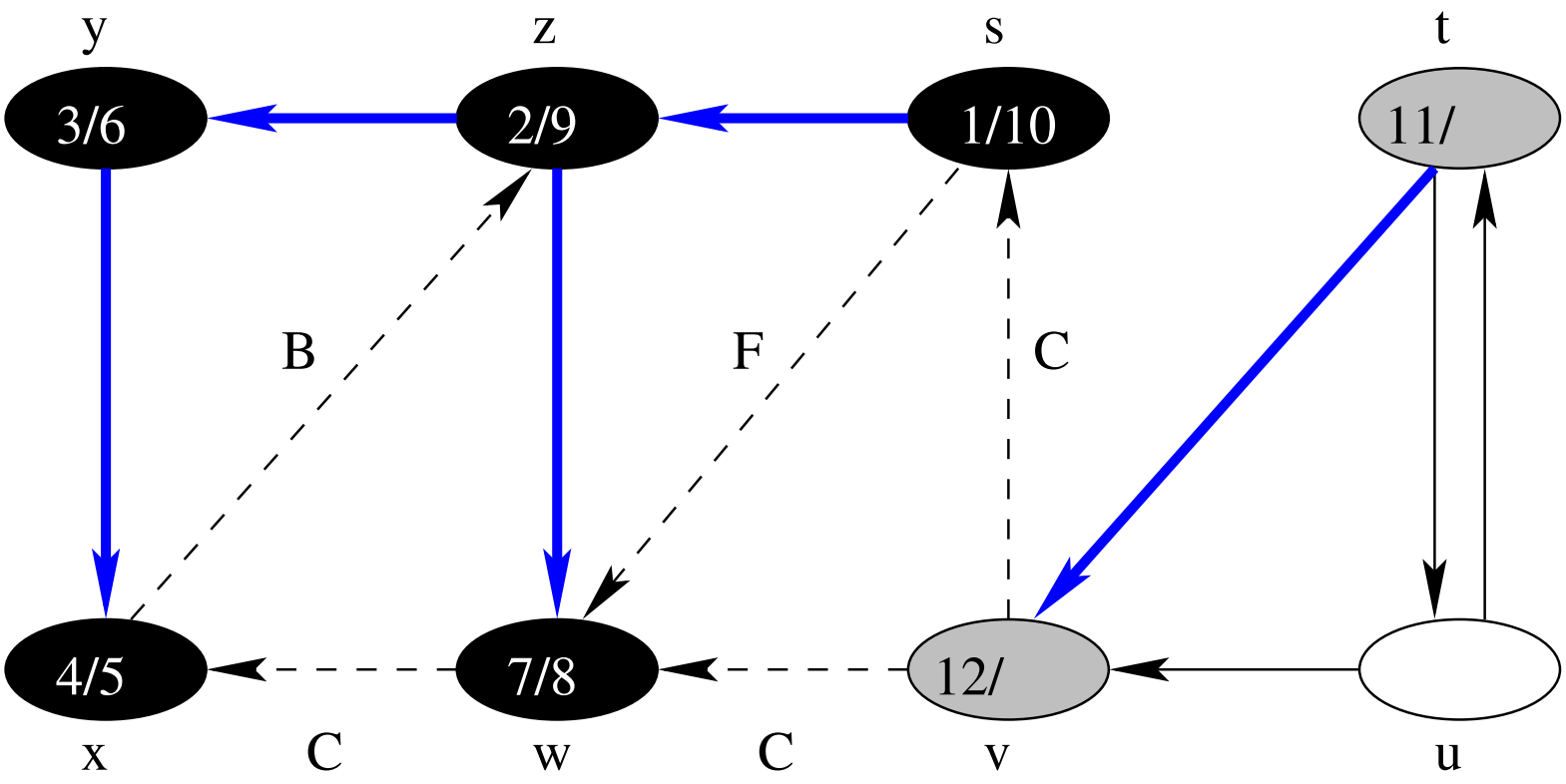
Exemplo DFS



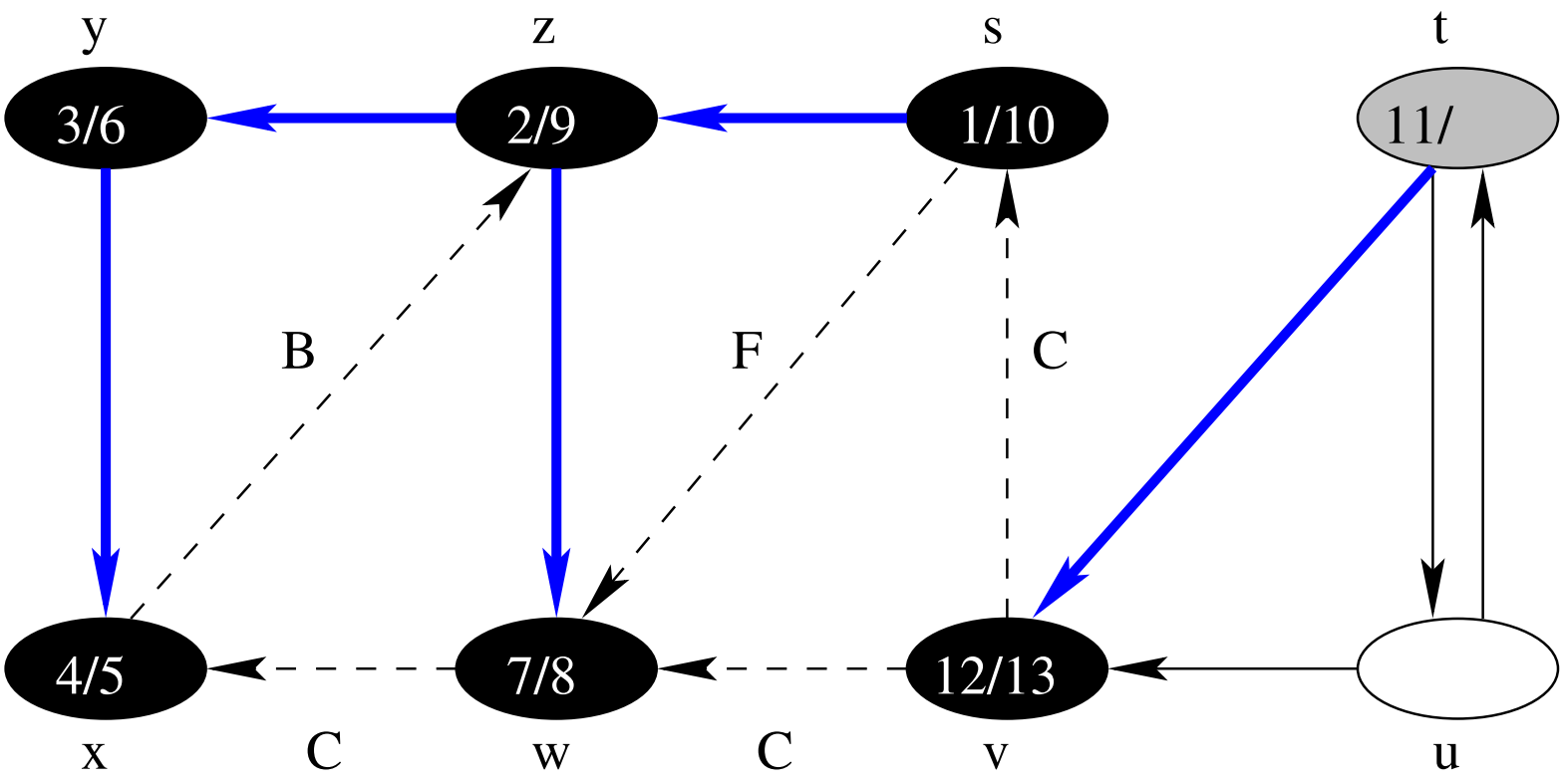
Exemplo DFS



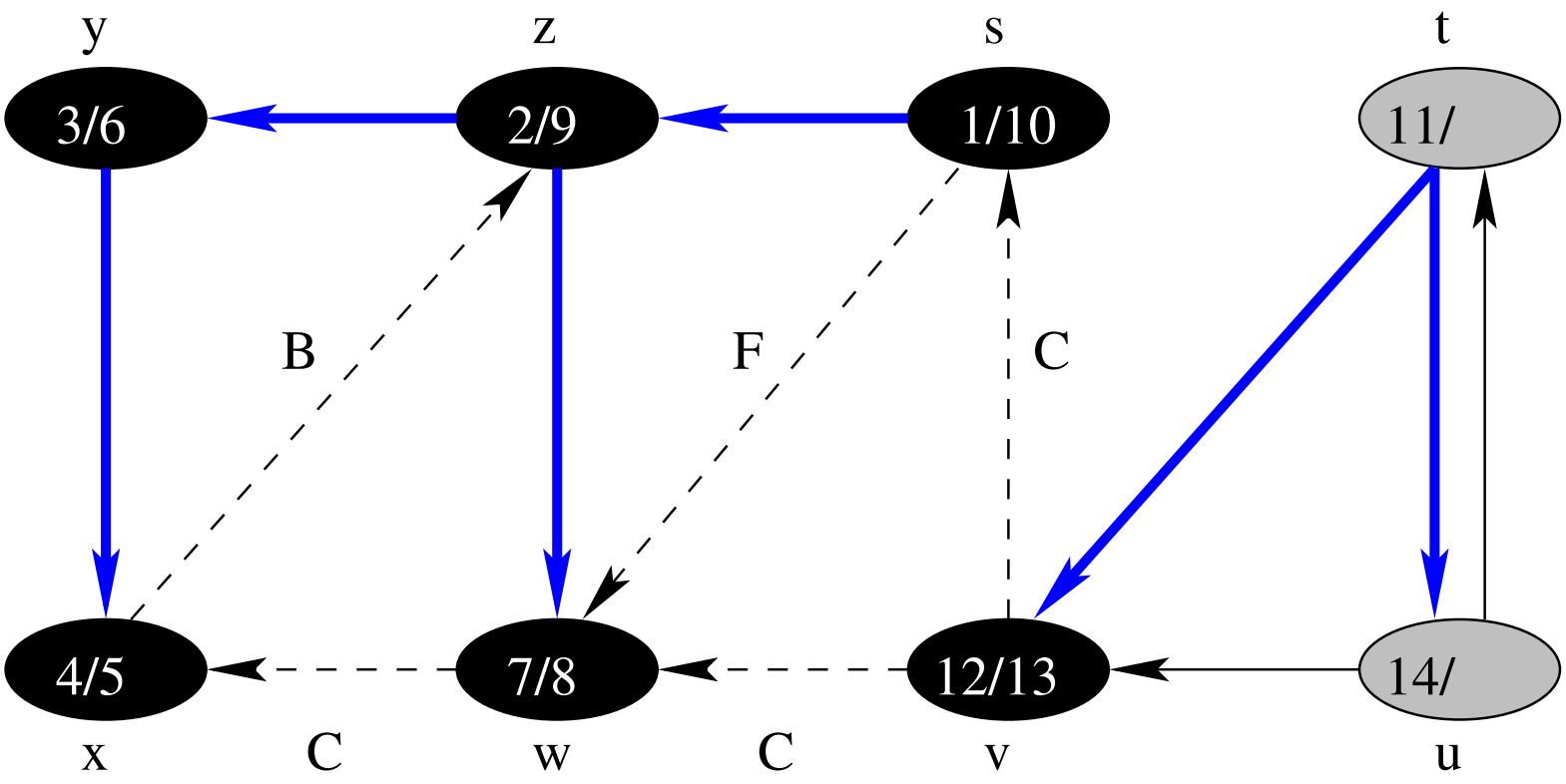
Exemplo DFS



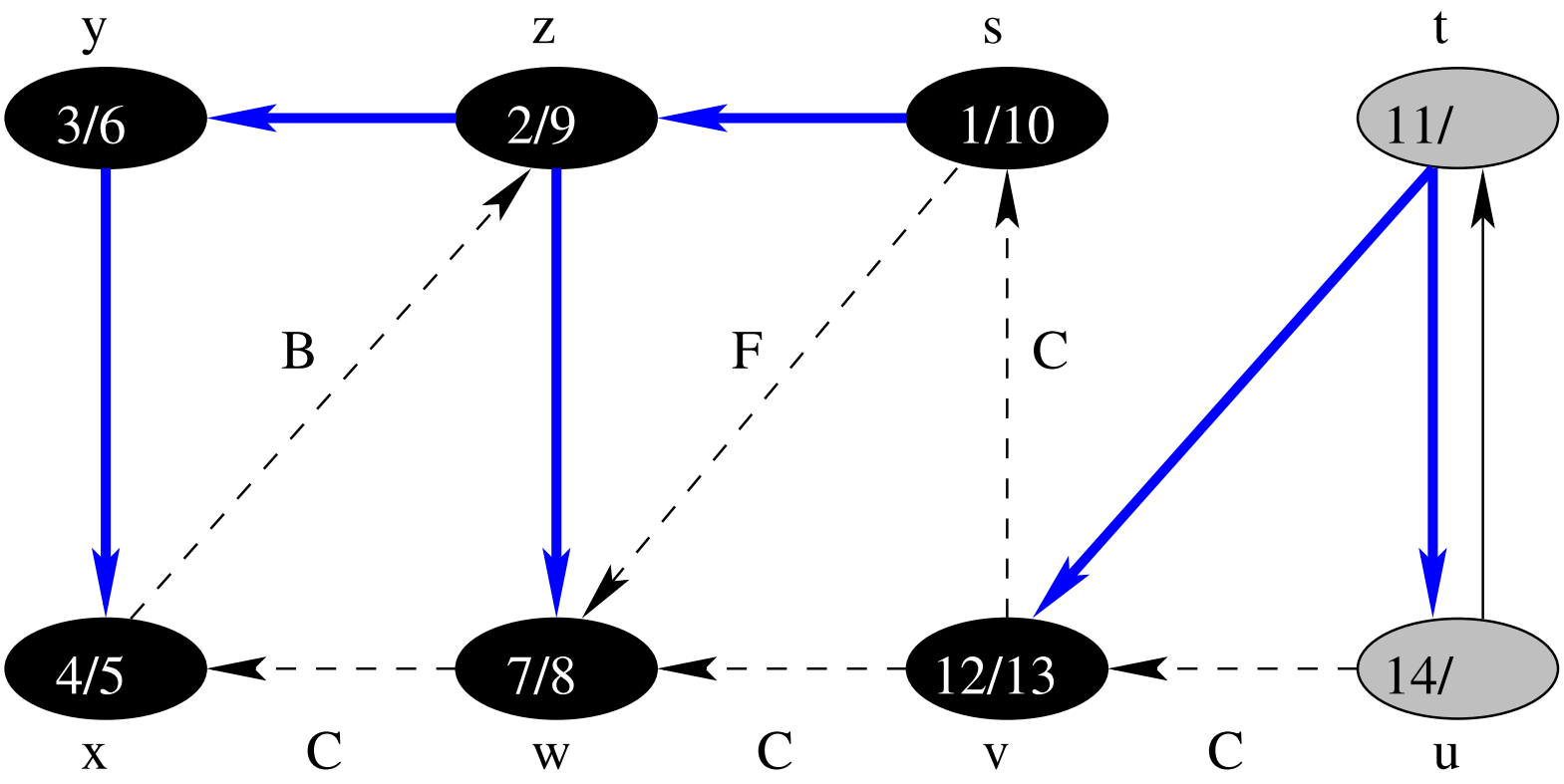
Exemplo DFS



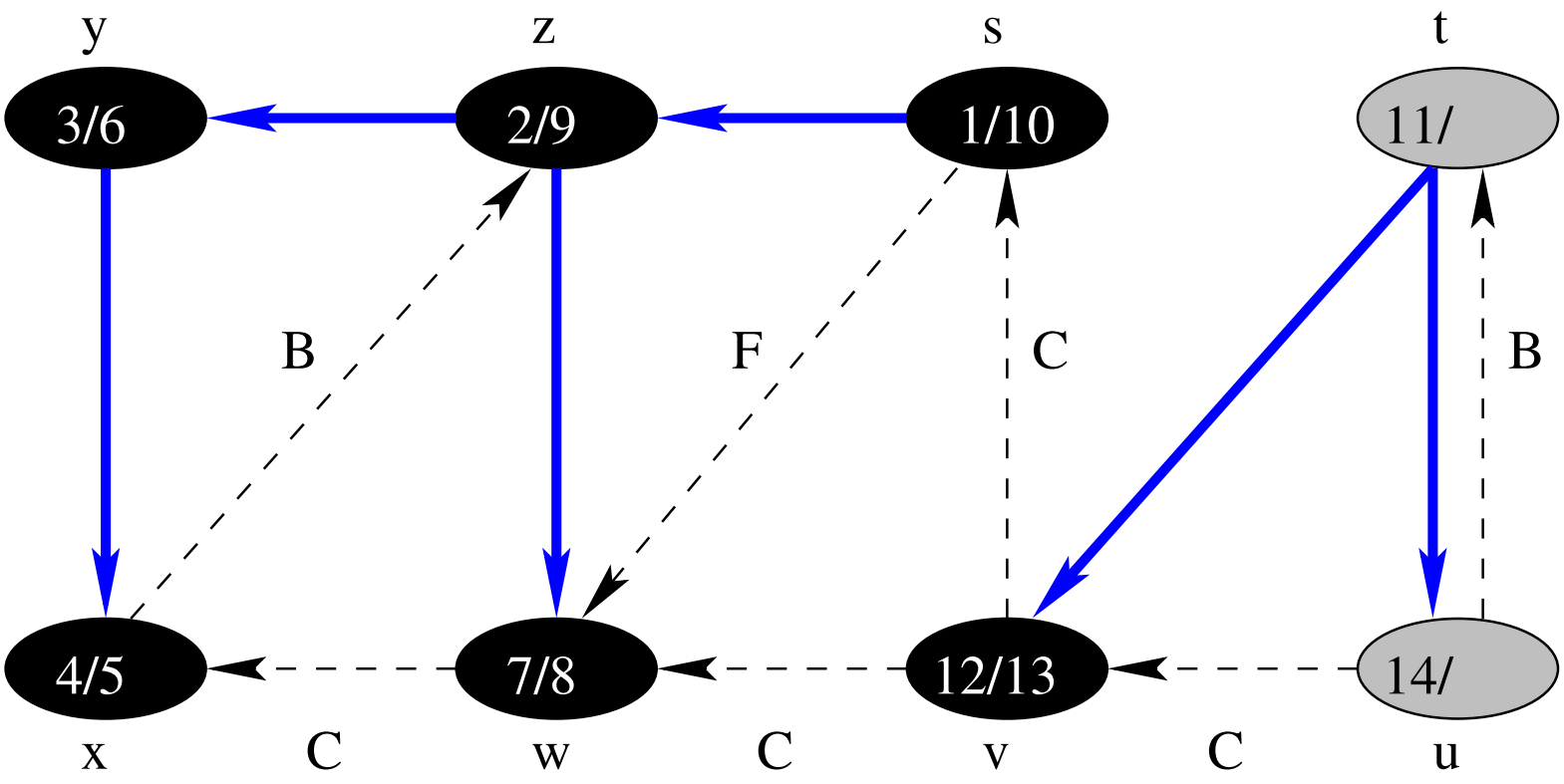
Exemplo DFS



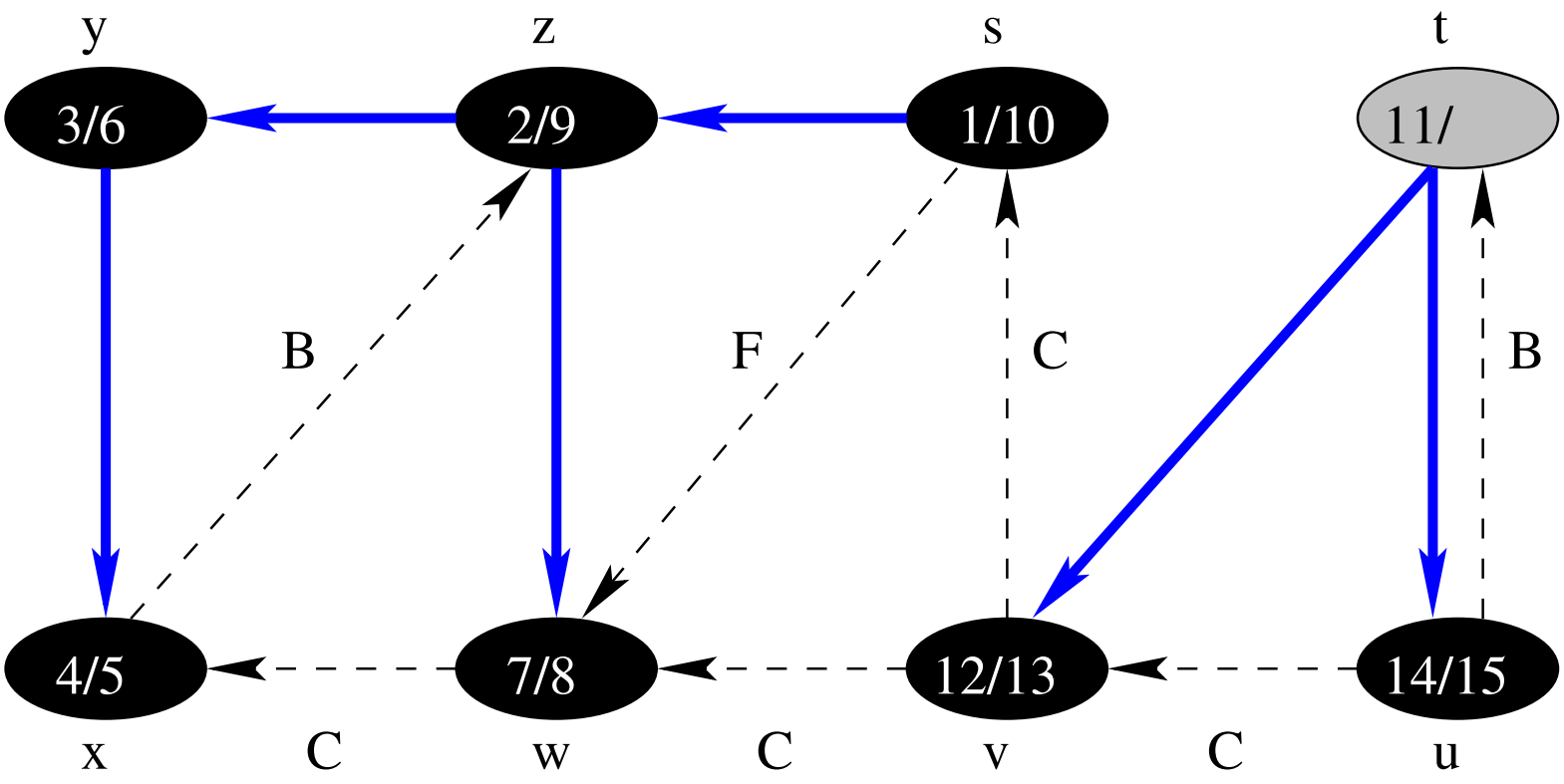
Exemplo DFS



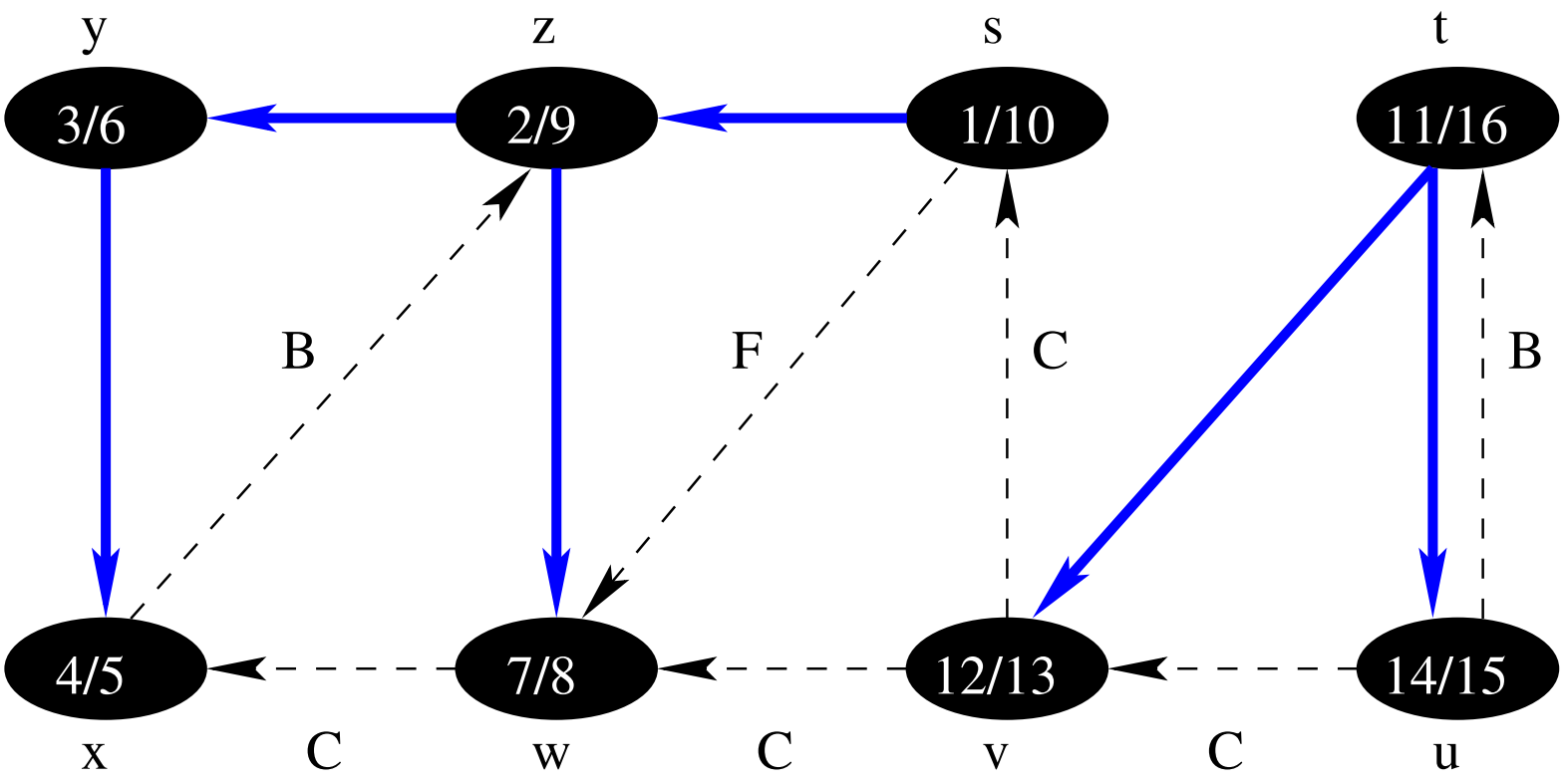
Exemplo DFS



Exemplo DFS



Exemplo DFS

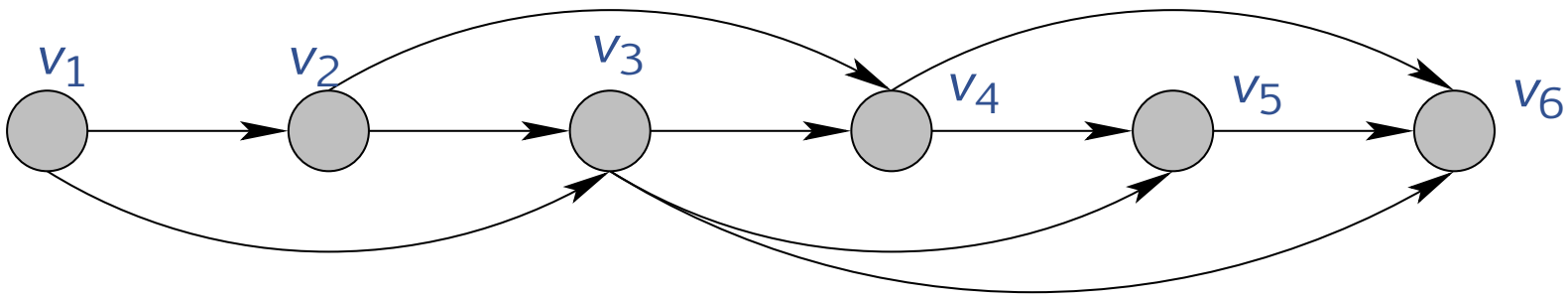


Ordenação Topológica

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

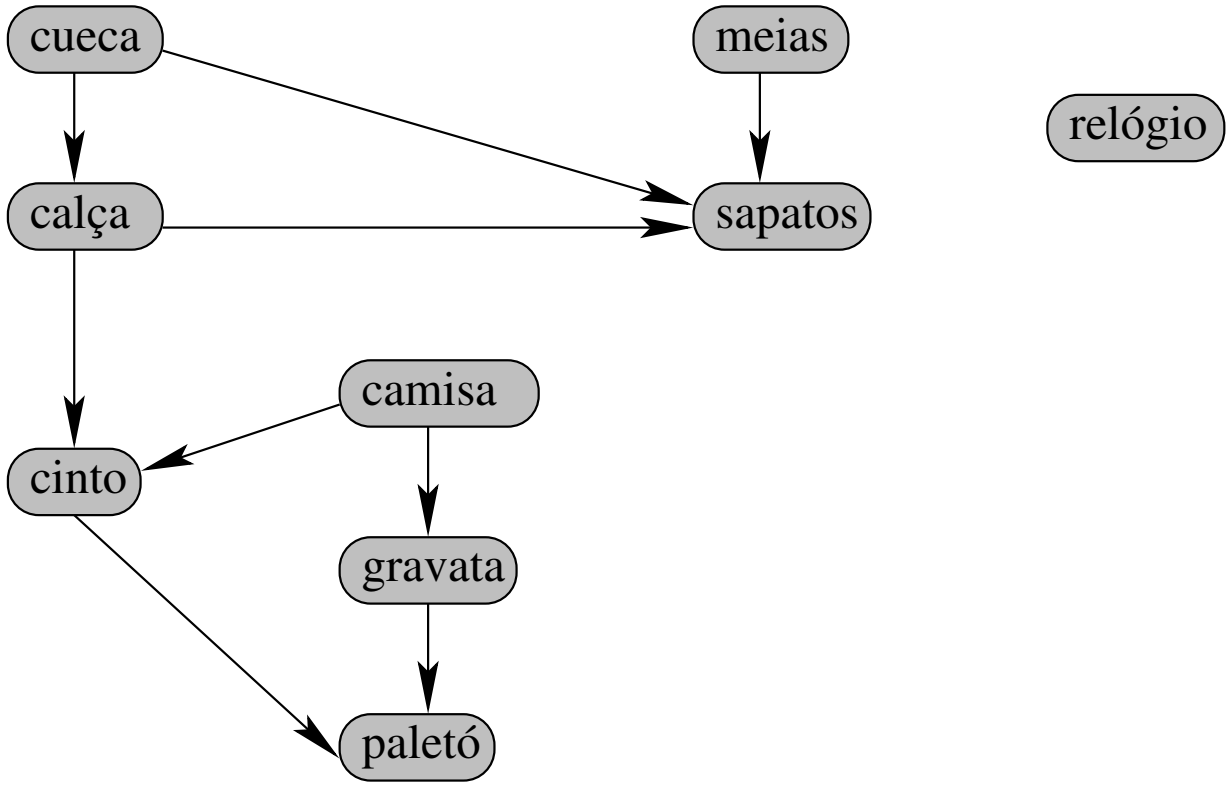
$$v_1, v_2, v_3, \ldots, v_{n-2}, v_{n-1}, v_n$$

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

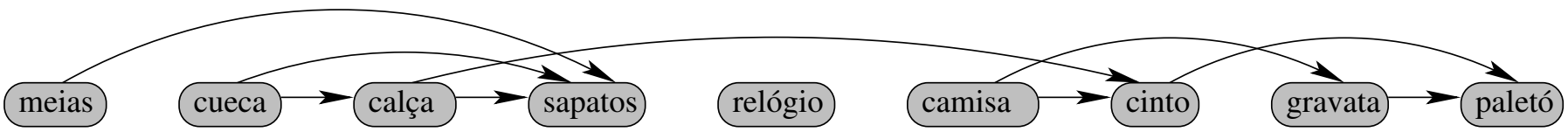


Ordenação Topológica

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

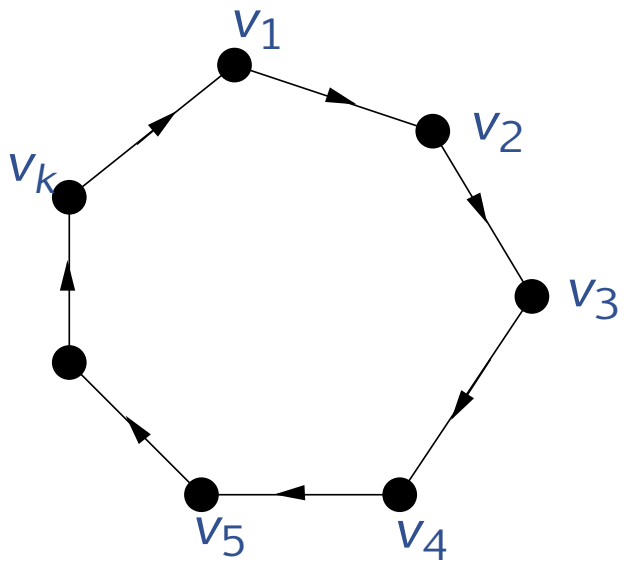


Ordenação Topológica



Ordenação Topológica

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



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

Grafo Direcionado Acíclico

Teorema. Um grafo direcionado 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 Direcionado Acíclico

Lema. Todo grafo direcionado acíclico G possui uma fonte e um sorvedouro.

Prova. Seja P um caminho mais longo em G e sejam s o início e t o término de P. Claramente, s é uma fonte e t é um sorvedouro. □

Agora para terminar a prova do teorema, usamos indução em $n := |V|$ para mostrar que todo grafo acíclico $G = (V, E)$ possui uma ordenação topológica.

Grafo Direcionado Acíclico

Base: se $n = 1$ então a sequência v_1 é uma ordenação topológica de G onde $V = \{v_1\}$.

Hipótese de indução: suponha que todo grafo acíclico com menos de n vértices possui uma ordenação topológica.

Passo de indução: pelo Lema acima, G possui um sorvedouro v_n. O grafo $G - v_n$ é acíclico e pela HI possui uma ordenação topológica $v_1, \ldots, v_{n-1}$.

Logo,

$v_1, v_2, \ldots, v_n$

é uma ordenação topológica de G.

Grafo Direcionado Acíclico

Baseado na prova anterior, pode-se projetar por indução um algoritmo para obter uma ordenação topológica de um grafo direcionado acíclico G.

- Encontre um sorvedouro v_n de G.
- Recursivamente encontre uma ordenação topológica $v_1, \ldots, v_{n-1}$ de $G - v_n$.
- Devolva $v_1, v_2, \ldots, v_n$.

Complexidade: $O(V^2)$ (análise grosseira)
Em cada nível da recursão gasta-se $O(V)$ para encontrar um sorvedouro. Como há $|V|$ níveis, a complexidade do algoritmo é $O(V^2)$.

Pode-se fazer melhor: O(V+E) (CLRS 22.4-5)

Algoritmo baseado em DFS

Vamos projetar um algoritmo linear (i.e. $O(V+E)$) para encontrar uma ordenação topológica de um grafo acíclico $G=(V,E)$.

Ideia: considere o primeiro vértice u que foi **finalizado** (i.e. recebeu cor preta) em uma busca em profundidade de G. Suponha que existe alguma aresta (u,v) saindo de u (ou seja, $v \in \mathrm{Adj}[u]$).

Como u é o primeiro vértice a ser finalizado, então v já foi visitado (tem cor cinza). Pela propriedade da busca em profundidade, todos os vértices cinzas estão numa árvore e portanto v é um ancestral de u. Mas isto significa que existe um ciclo direcionado formado pelo caminho na árvore de v a u e pela aresta (u,v), o que é uma contradição. Logo, v é um sorvedouro.

Algoritmo baseado em DFS

Continuando a busca, toda vez que vértice é finalizado, este só pode ser adjacente a vértices **finalizados** (cor preta) anteriormente.

Logo, se colocarmos os vértices em ordem decrescente de tempo de finalização, obtemos uma ordenação topológica de G.

Algoritmo baseado em DFS

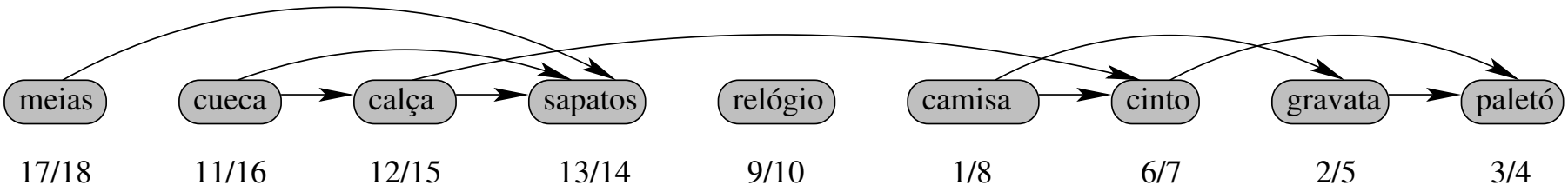
Recebe um grafo direcionado acíclico G e
devolve uma ordenação topológica de G.

TOPOLOGICAL-SORT(G)

1 Execute DFS(G) para calcular $f[u]$ para cada vértice u
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 $f[v]$.

Exemplo


meias
cueca
calça
sapatos
relógio
camisa
cinto
gravata
paletó
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4

Complexidade de tempo

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

A execução de DFS(G) consome tempo $O(V + E)$. Além disso, uma inserção na lista ligada consome tempo $O(1)$ e há exatamente $|V|$ inserções.

Logo, a complexidade de tempo de TOPOLOGICAL-SORT é $O(V + E)$.

Agora falta mostrar que TOPOLOGICAL-SORT funciona.

Correção

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

Prova:

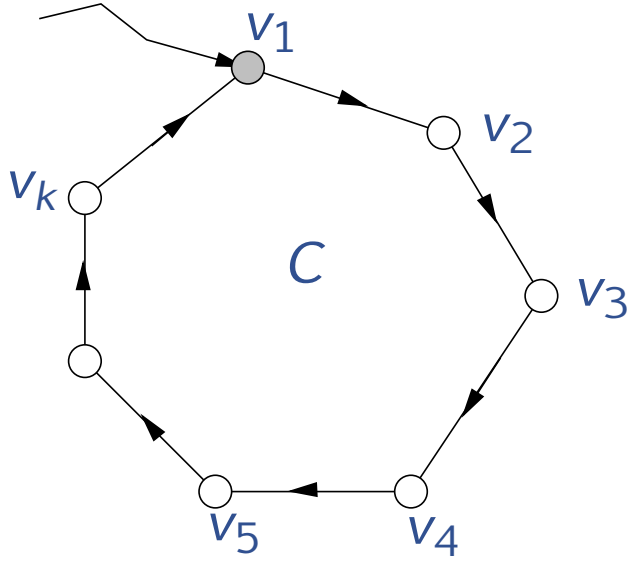
Suponha que (u, v) é uma aresta de retorno.

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 direcionado. Logo, G não é acíclico.

Correção

Agora suponha que G contém um ciclo direcionado 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.

Correção

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

Para mostrar que o algoritmo funciona, basta 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.

Correção

- 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.

Contagem de caminhos

Contagem de caminhos

Exercício (CLRS 22.4-2). Descreva um algoritmo linear que recebe um grafo direcionado acíclico G e vértices s, t e devolve o **número** de caminhos de s a t em G.

Note que só é preciso contar os caminhos, não exibi-los.

Para simplificar a apresentação, suporemos que o grafo **não possui arestas múltiplas**. Este caso pode ser tratado de modo similar e fica como exercício.

Contagem de caminhos

Ideia: combinar ordenação topológica e programação dinâmica

Seja G um grafo direcionado acíclico. Seja

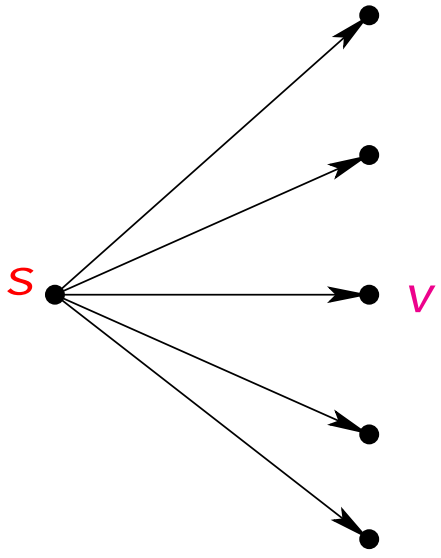
$$p(v) = \text{número de caminhos de } v \text{ a } t.$$

Queremos determinar $p(s)$.

Para isto queremos descobrir uma recorrência para $p(s)$ e de modo, mais geral uma recorrência para $p(v)$.

Contagem de caminhos

Considere $p(v)$ para cada $v \in \text{Adj}[s]$.

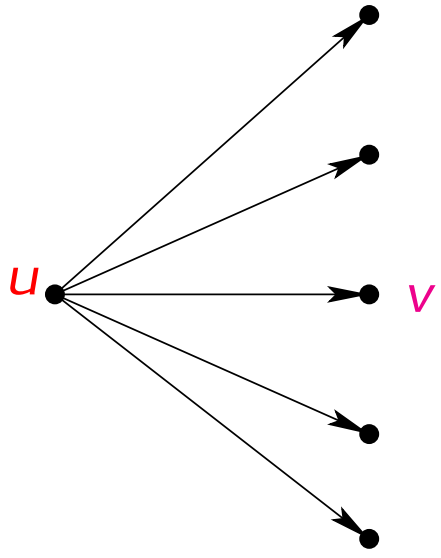


Qual é a relação entre $p(s)$ e estes valores?

$$p(s) = \sum_{v \in \text{Adj}[s]} p(v).$$

Contagem de caminhos

O mesmo vale para qualquer vértice u (em vez de s).



$$p(u) = \sum_{v \in \mathrm{Adj}[u]} p(v).$$

Pergunta: esta *recorrência* vale se G contiver ciclos?

Contagem de caminhos

$$p(u) = \begin{cases} 0 & \text{se } u \text{ é um sorvedouro,} \\ \sum_{v \in \mathrm{Adj}[u]} p(v) & \text{caso contrário.} \end{cases}$$

Se fixarmos uma ordenação topológica de G, então o valor $p(u)$ depende apenas de valores $p(v)$ onde v sucede u.

$$p(u) = \begin{cases} 1 & \text{se } u = t \\ 0 & \text{se } u \text{ sucede } t \text{ ou é sorvedouro} \\ \sum_{v \in \mathrm{Adj}[u]} p(v) & \text{caso contrário} \end{cases}$$

Contagem de caminhos

```
CONTA-CAMINHOS(G, s, t)      ▷ G é acíclico
 1 para cada u ∈ V[G] − {s} faça
 2     p[u] ← 0
 3 p[t] ← 1
 4 obtenha uma ordenação topológica v_1, v_2, ..., v_n de G
 5 para i ← n − 1 até 1 faça ▷ em ordem inversa
 6     para cada v ∈ Adj[v_i] faça
 7         p[v_i] ← p[v_i] + p[v]
 8 devolva p
```

Complexidade: $O(V + E)$

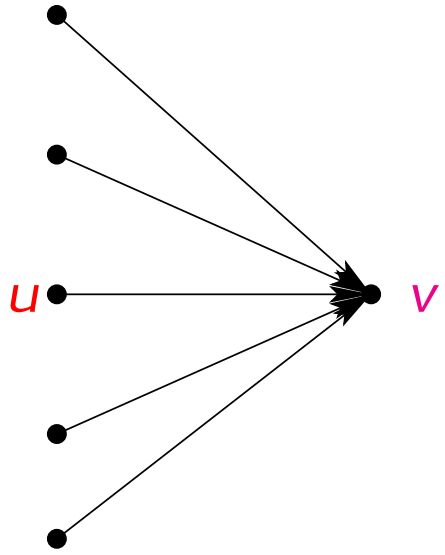
Para ver a correção basta provar que no início da linha 5 vale o seguinte invariante: $p[v_i] = p(v_i)$.

Exercício: outra forma de contar os caminhos

Seja G um grafo direcionado acíclico. Seja

$$q(v) = \text{número de caminhos de } s \text{ a } v.$$

Queremos determinar $q(t)$.



Descubra uma recorrência para $q(v)$ que envolve os valores $q(u)$ para u tal que $v \in \mathrm{Adj}[u]$.

Exercício: outra forma de contar os caminhos

Verifique que o algoritmo abaixo produz uma resposta correta.

```
CONTA-CAMINHOS(G, s, t)    ▷ G é acíclico
 1  para cada v ∈ V[G] − {s} faça
 2      q[v] ← 0
 3  q[s] ← 1
 4  obtenha uma ordenação topológica v_1, v_2, ..., v_n de G
 5  para i ← 1 até n − 1 faça
 6      para cada v ∈ Adj[v_i] faça
 7          q[v] ← q[v] + q[v_i]
 8  devolva q
```

Note atentamente a diferença com o algoritmo anterior no modo como os valores $q[v]$ são preenchidos.

Componentes fortemente conexos

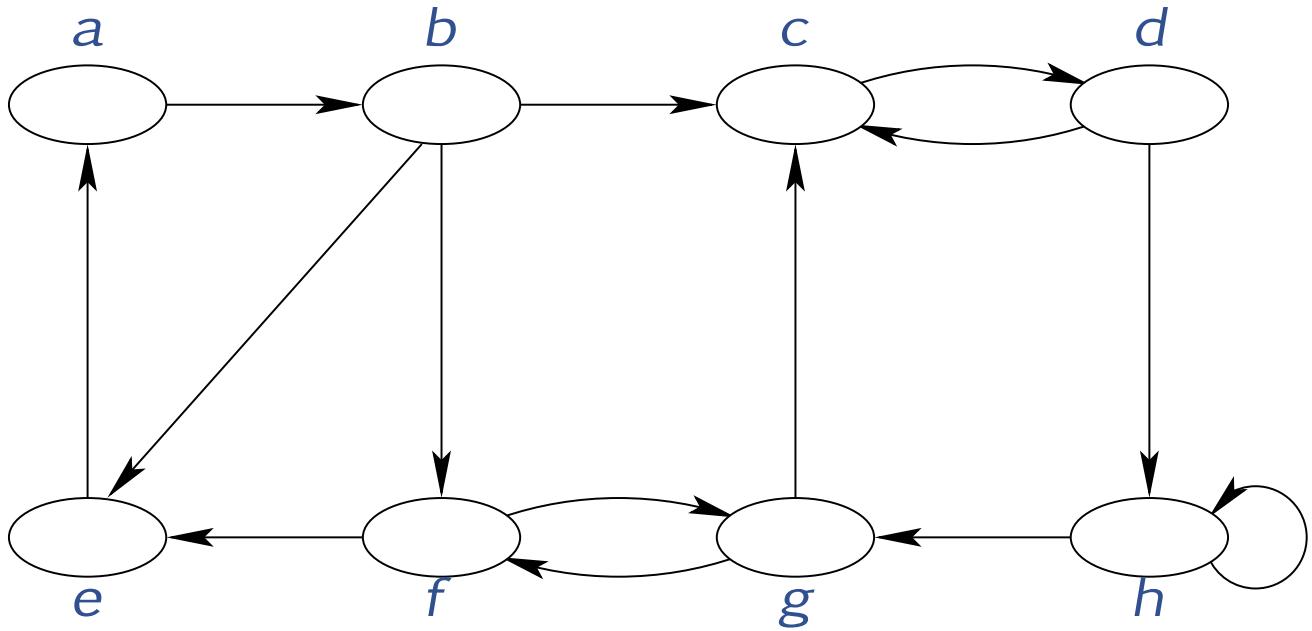
Componentes fortemente conexos (CFC)

- Uma aplicação clássica de busca em profundidade:

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

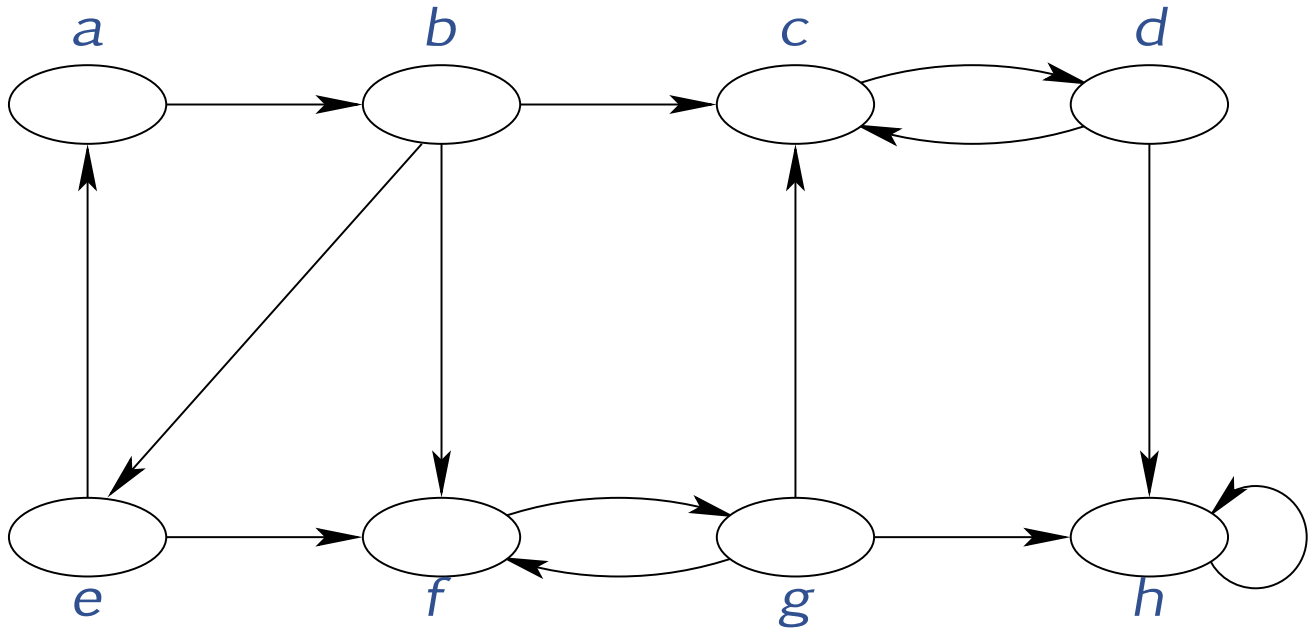
Grafo fortemente conexo

Um grafo direcionado $G = (V, E)$ é fortemente conexo se para todo par de vértices u, v de G, existe um caminho direcionado de u a v.



Grafo fortemente conexo

Nem todo grafo direcionado é fortemente conexo.

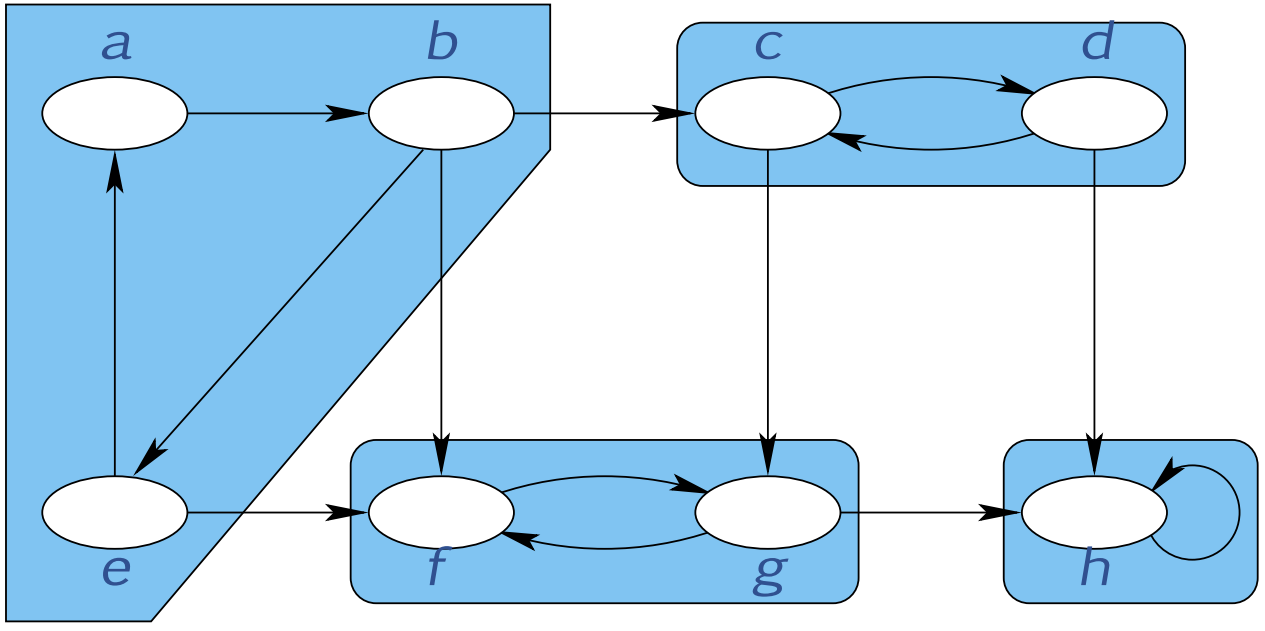


Componentes fortemente conexos

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

1. O subgrafo induzido por C é fortemente conexo.
2. C é maximal com respeito à propriedade (1).

Componentes fortemente conexos

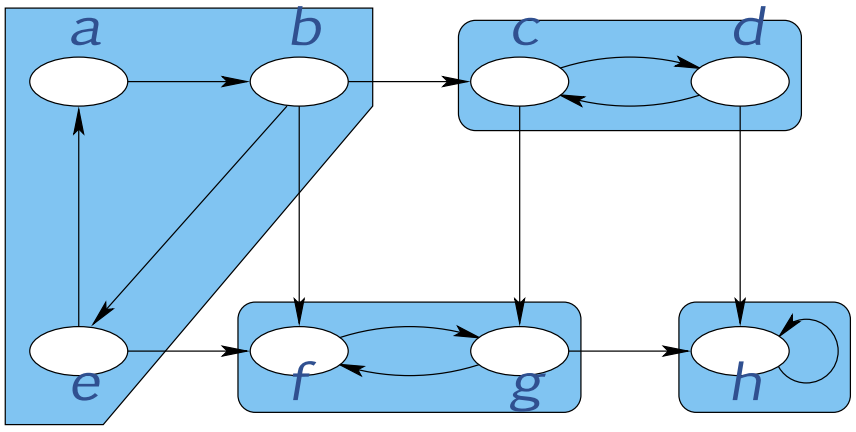


Um grafo direcionado e seus componentes fortemente conexos.

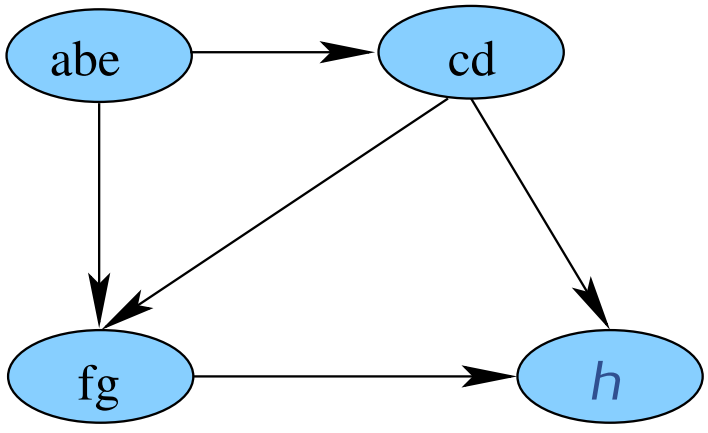
Problema: dado um grafo G, determinar seus componentes fortemente conexos.

Grafo Componente

Para entender melhor o algoritmo que veremos, considere o grafo componente G^{CFC} obtido a partir de G **contraindo-se** seus componentes fortemente conexos.



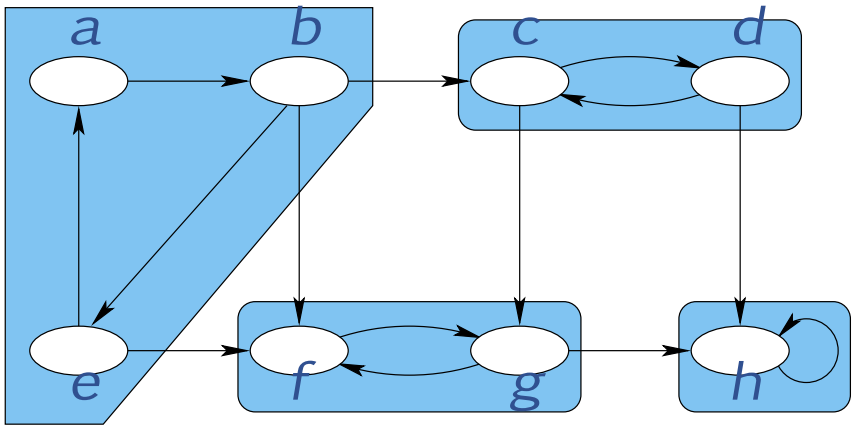
G



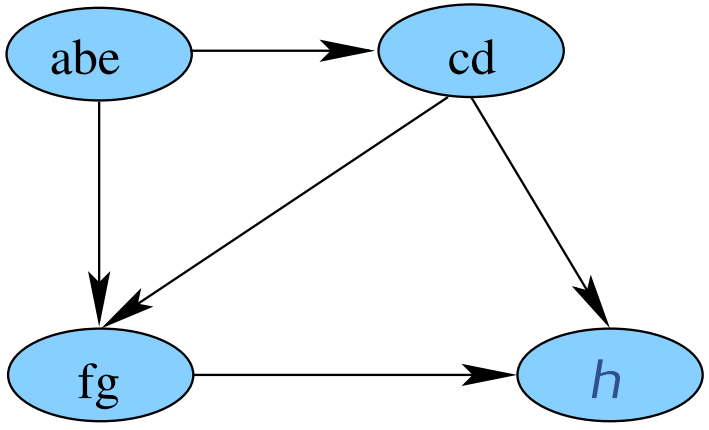
G^{CFC}

Mais precisamente, cada vértice de G^{CFC} corresponde a um componente fortemente conexo de G e existe uma aresta (C, C') em G^{CFC} se existem $u \in C$ e $v \in C'$ tais que $(u, v) \in E[G]$. Note que G^{CFC} é **acíclico**. (Por quê?)

Grafo Componente



G



G^{CFC}

Considere uma busca em profundidade sobre G e seja u o **último** vértice a ser **finalizado**. O vértice u tem que pertencer a um componente fortemente conexo de G que corresponde a uma fonte de G^{CFC}. (Por quê?)

Como podemos usar esta informação?

Grafo transposto

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

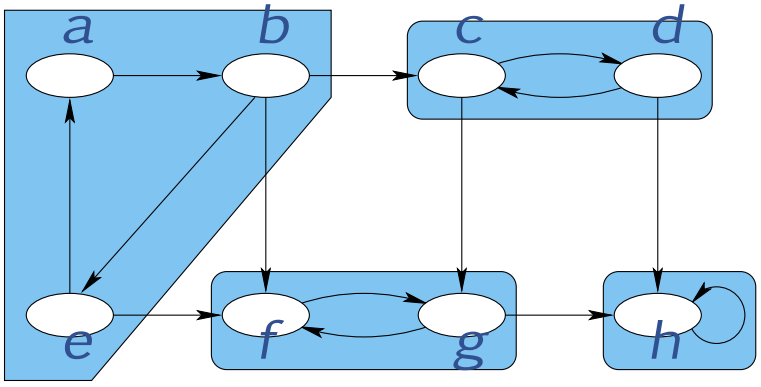
O grafo transposto de G é o grafo $G^T = (V^T, E^T)$ tal que

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

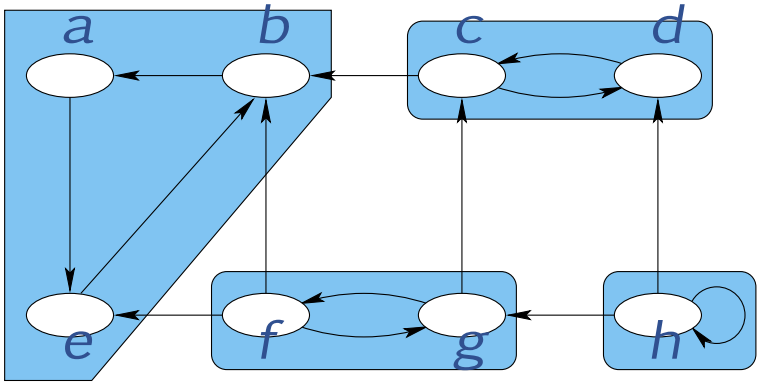
Ou seja, G^T é obtido a partir de G invertendo as orientações das arestas.

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

Grafo transposto



G



G^T

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

Note que agora se começarmos uma nova busca em profundidade, mas agora começando no vértice u que foi **finalizado** mais tarde (na primeira busca), ela encontraria o componente fortemente conexo que contém u (os vértices da primeira árvore encontrada.)

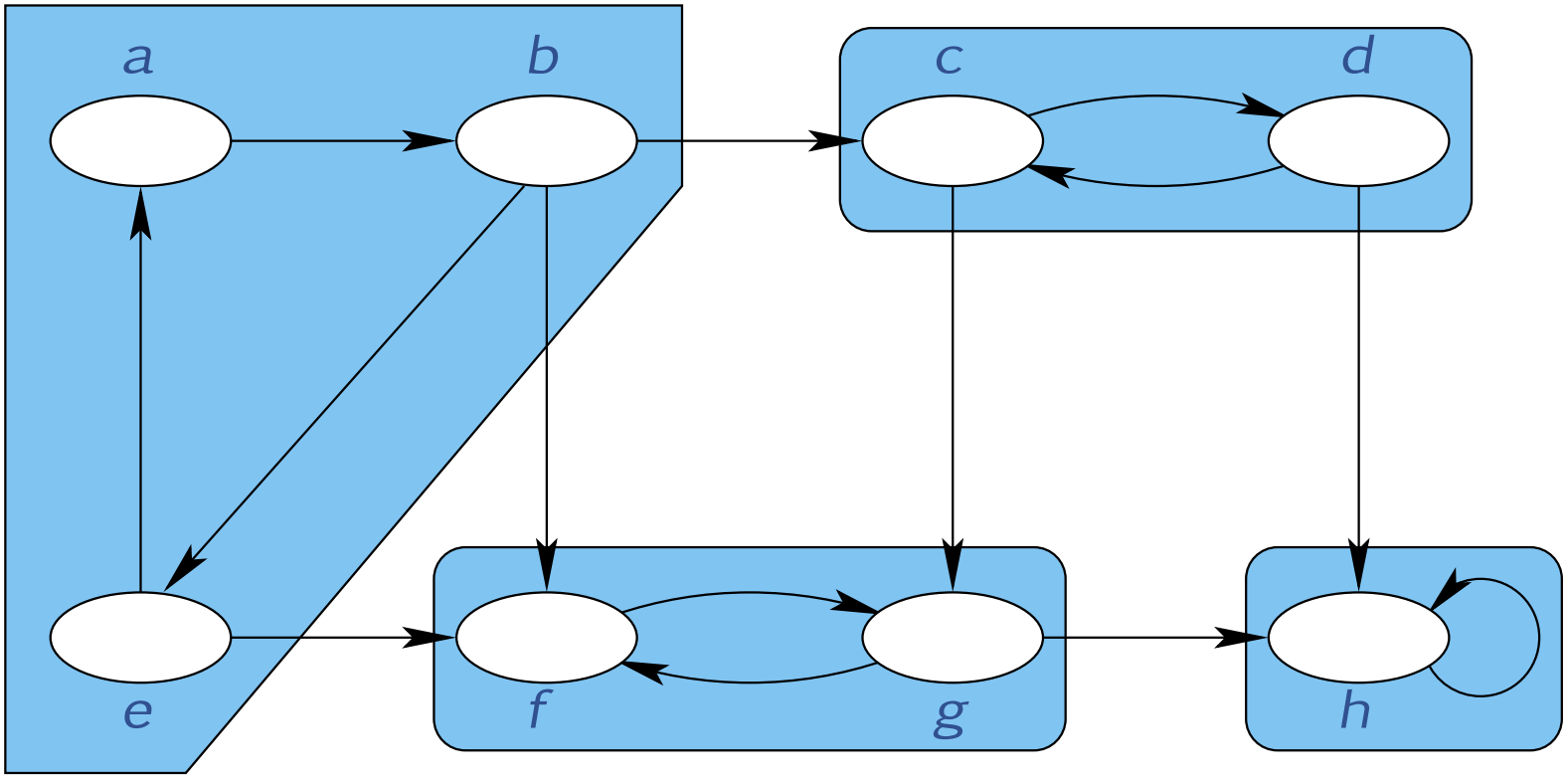
Algoritmo

Componentes-Fortemente-Conexos(G)

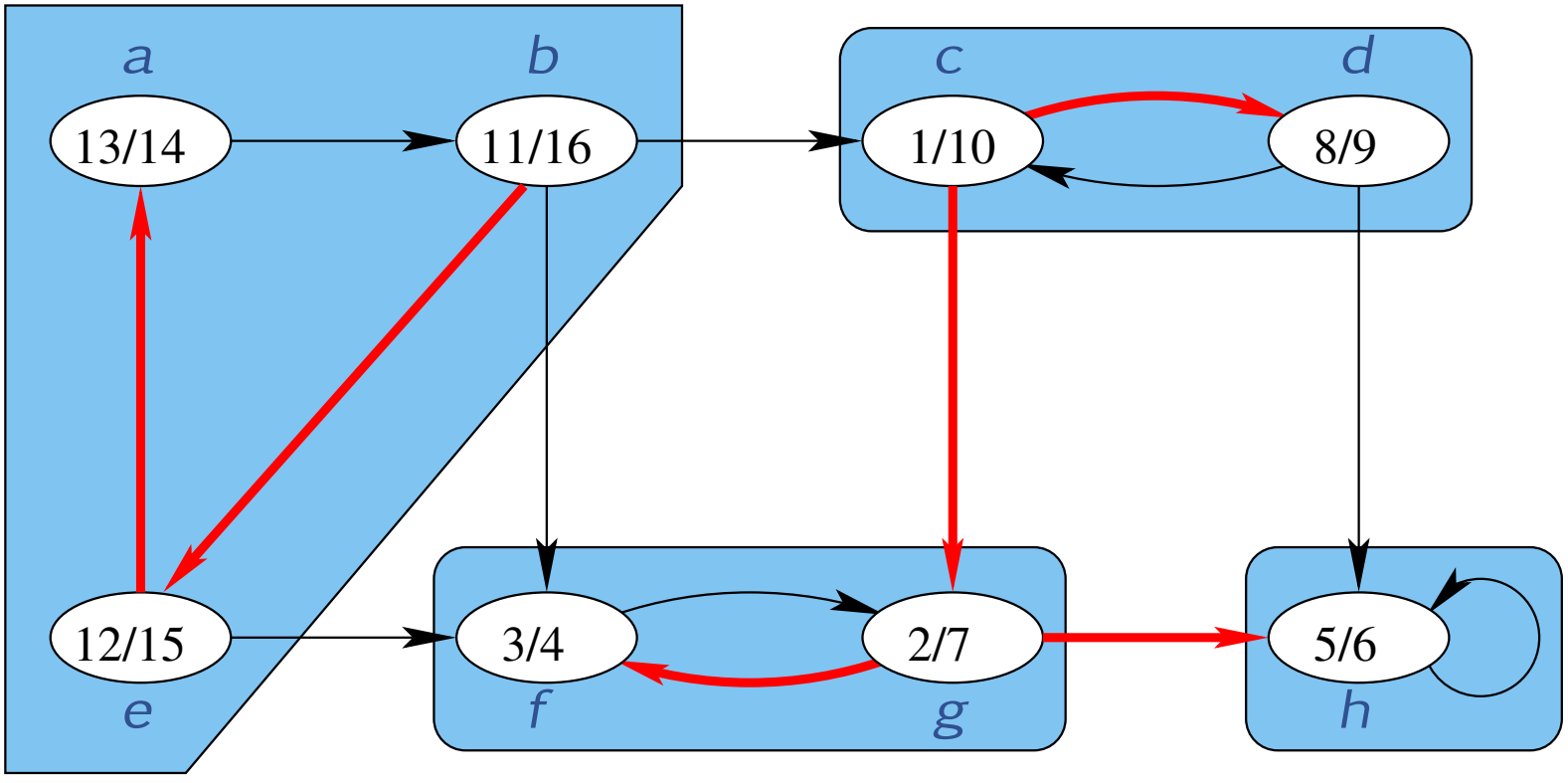
1. Execute DFS(G) para obter $f[v]$ para $v \in V$.
2. Execute DFS(G^T) considerando os vértices em ordem decrescente de $f[v]$.
3. Devolva os conjuntos de vértices de cada árvore da Floresta de Busca em Profundidade obtida.

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

Exemplo CLRS

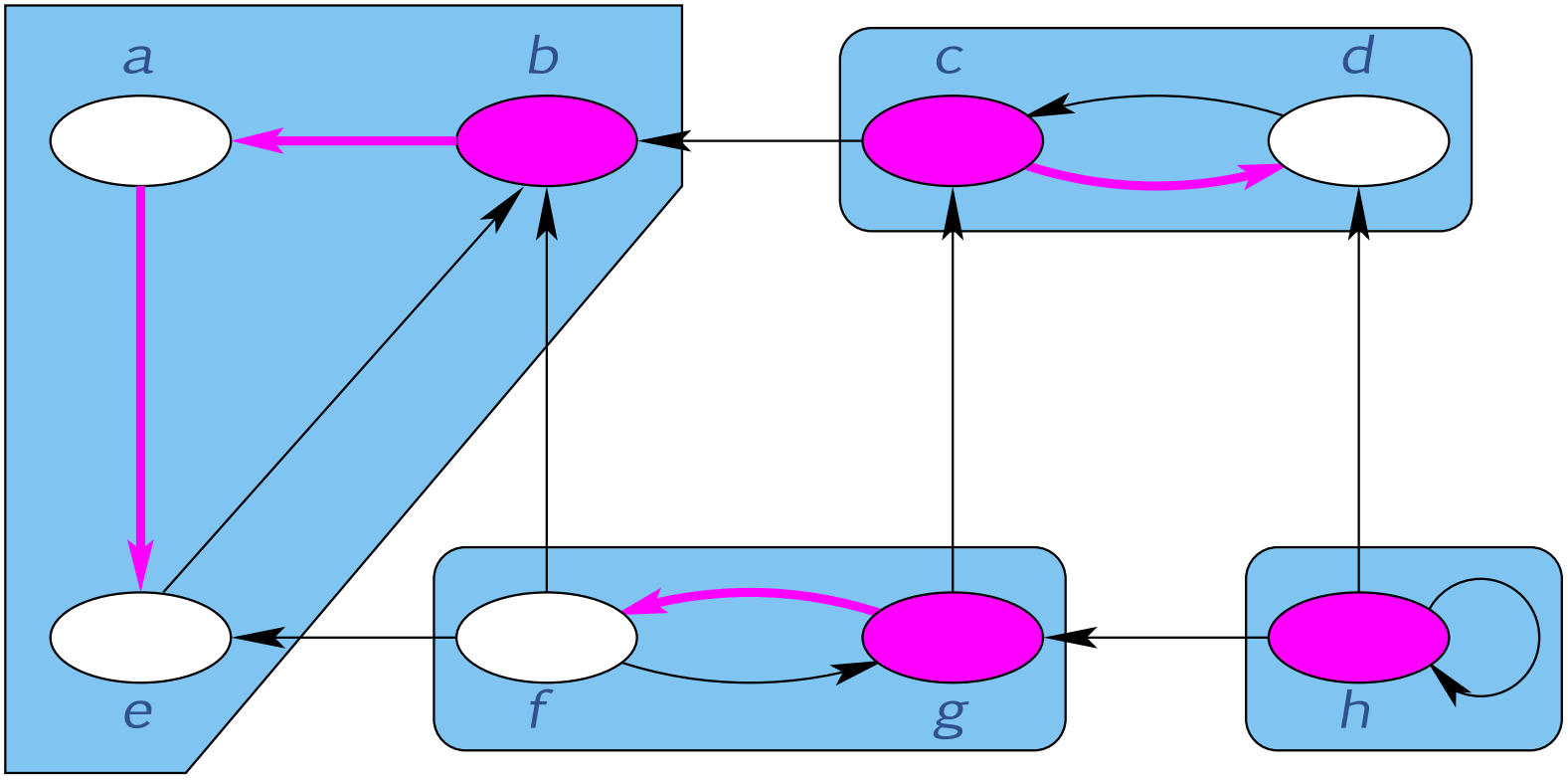


Exemplo CLRS



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

Exemplo CLRS



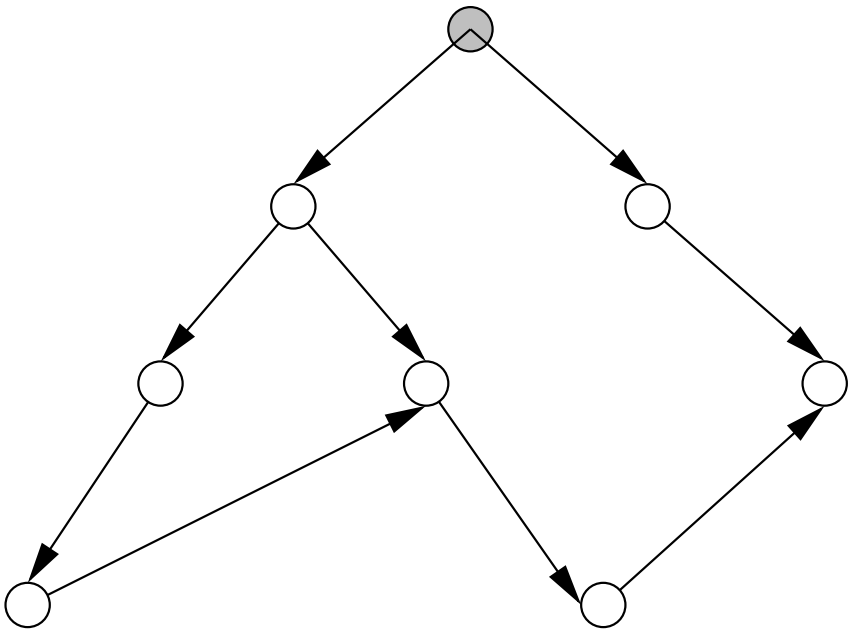
2 Depois execute DFS(G^T) considerando os vértices em ordem decrescente de $f[v]$.

3 Os componentes fortemente conexos correspondem aos vértices de cada árvore da Floresta de Busca em Profundidade.

Relembrando

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.



Correção

Lema 22.13 (CLRS)

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

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

Agora mostraremos que o algoritmo Componentes-Fortemente-Conexos funciona.

Correção

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

Definição:

Para todo subconjunto U de vértices sejam

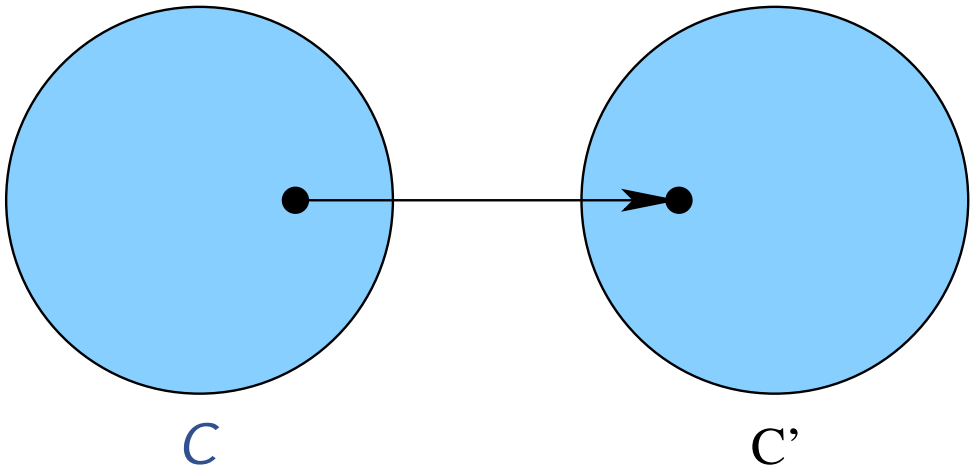
$$d(U) := \min_{u \in U}\{d[u]\} \quad \text{e} \quad f(U) := \max_{u \in U}\{f[u]\}.$$

Ou seja,
$d(U)$ é o instante em que o primeiro vértice de U foi descoberto e
$f(U)$ é o instante em que o último vértice de U foi finalizado.

Correção

Lema 22.14 (CLRS):

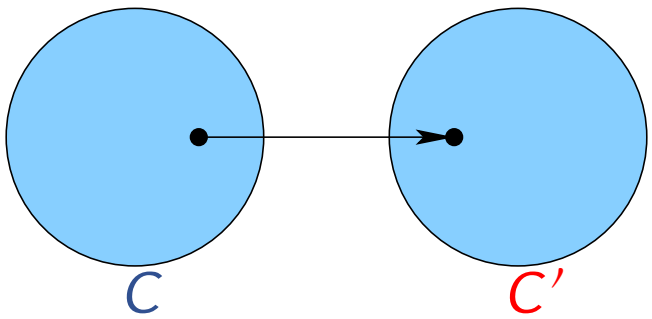
Sejam C e C' dois componentes f.c. distintos de G.
Suponha que existe (u, v) em E onde $u \in C$ e $v \in C'$.
Então $f(C) > f(C')$.



Correção

Lema 22.14 (CLRS):

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



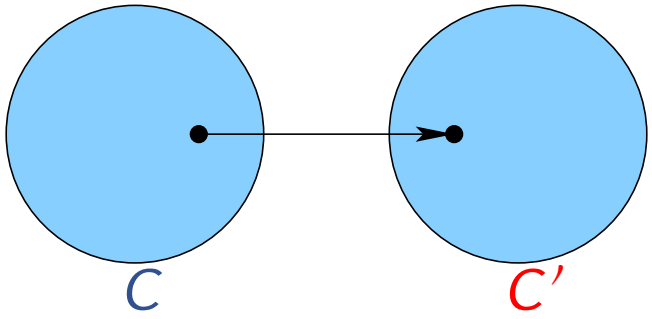
Prova:

Caso 1: $d(C) < d(C')$. Seja x o primeiro vértice de C que foi descoberto. Logo $d[x] = d(C)$. No instante $d[x]$, existe um caminho branco de de x a todo vértice de $C \cup C'$. Então todos os vértices de $C \cup C'$ são descendentes de x e portanto, $f(C') < f[x] \leq f(C)$.

Correção

Lema 22.14 (CLRS):

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



Caso 2: $d(C) > d(C')$. O primeiro vértice de $C \cup C'$ a ser descoberto pertence a C'. Logo, todos os vértices de C' serão finalizados antes de qualquer vértice de C ser descoberto. Isso mostra que $f(C) > f(C')$. □

Correção

Lema 22.14 (CLRS):

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

Corolário 22.15 (CLRS):

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

Segue do fato de que G e G^T terem os mesmos componentes fortemente conexos e do Lema 22.14.

Correção

Componentes-Fortemente-Conexos(G)

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

Teorema 22.16 (CLRS):

O algoritmo Componentes-Fortemente-Conexos determina os componentes fortemente conexos de G em tempo $O(V + E)$.

Correção

Prova: (CLRS)

Vamos provar por **indução** no número de árvores produzidas na linha 3 que os vértices de cada árvore são componentes fortemente conexos.

Base: $k = 0$ (trivial)

Hipótese de indução: as primeiras k árvores produzidas na linha 3 são componentes fortemente conexos.

Correção

Passo de indução: considere a $(k+1)$-ésima árvore produzida pelo algoritmo. Vamos mostrar que seu conjunto de vértices é um componente fortemente conexo.

Seja u a raiz desta árvore e seja C o componente fortemente conexo ao qual u pertence.

Pela escolha do algoritmo, $f(C) > f(C')$ para qualquer outro componente fortemente conexo C' que consiste de vértices ainda não visitados em DFS(G^T).

Correção

Pela hipótese de indução, no instante $d[u]$ todos os vértices de C são brancos. Assim, todos os vértices de C tornam-se descendentes de u na árvore de busca de G^T.

Pela hipótese de indução e pelo Corolário 22.15, qualquer aresta que sai de C só pode entrar em uma das k componentes já visitadas.

Logo, apenas os vértices de C estão na árvore de busca em profundidade de G^T com raiz u. □