MC558 — Análise de Algoritmos II

Cid C. de Souza Cândida N. da Silva Orlando Lee

2 de abril de 2023

Antes de mais nada...

- Uma versão anterior deste conjunto de slides foi preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para uma instância anterior desta disciplina.
- O que vocês tem em mãos é uma versão modificada preparada para atender a meus gostos.
- Nunca é demais enfatizar que o material é apenas um guia e não deve ser usado como única fonte de estudo. Para isso consultem a bibliografia (em especial o CLR ou CLRS).

Orlando Lee

Agradecimentos (Cid e Cândida)

- Várias pessoas contribuíram direta ou indiretamente com a preparação deste material.
- Algumas destas pessoas cederam gentilmente seus arquivos digitais enquanto outras cederam gentilmente o seu tempo fazendo correções e dando sugestões.
- Uma lista destes "colaboradores" (em ordem alfabética) é dada abaixo:
  - Célia Picinin de Mello
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofiloff
  - Pedro Rezende
  - Ricardo Dahab
  - Zanoni Dias

Busca em profundidade

Depth First Search = busca em profundidade

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

Busca em profundidade

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

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

Busca em profundidade

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

- Suponha que a busca atingiu um vértice u.
- Escolhe-se um **vizinho não visitado** v de u para prosseguir a busca.
- Empilhe u e repita o passo 1 com v no lugar de u.
- Se nenhum vértice não visitado foi encontrado, então desempilhe um vértice u e volte ao passo 1.

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.
- Um vértice x com $\pi[x] = \text{NIL}$ é raiz de uma árvore da floresta.
- 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).

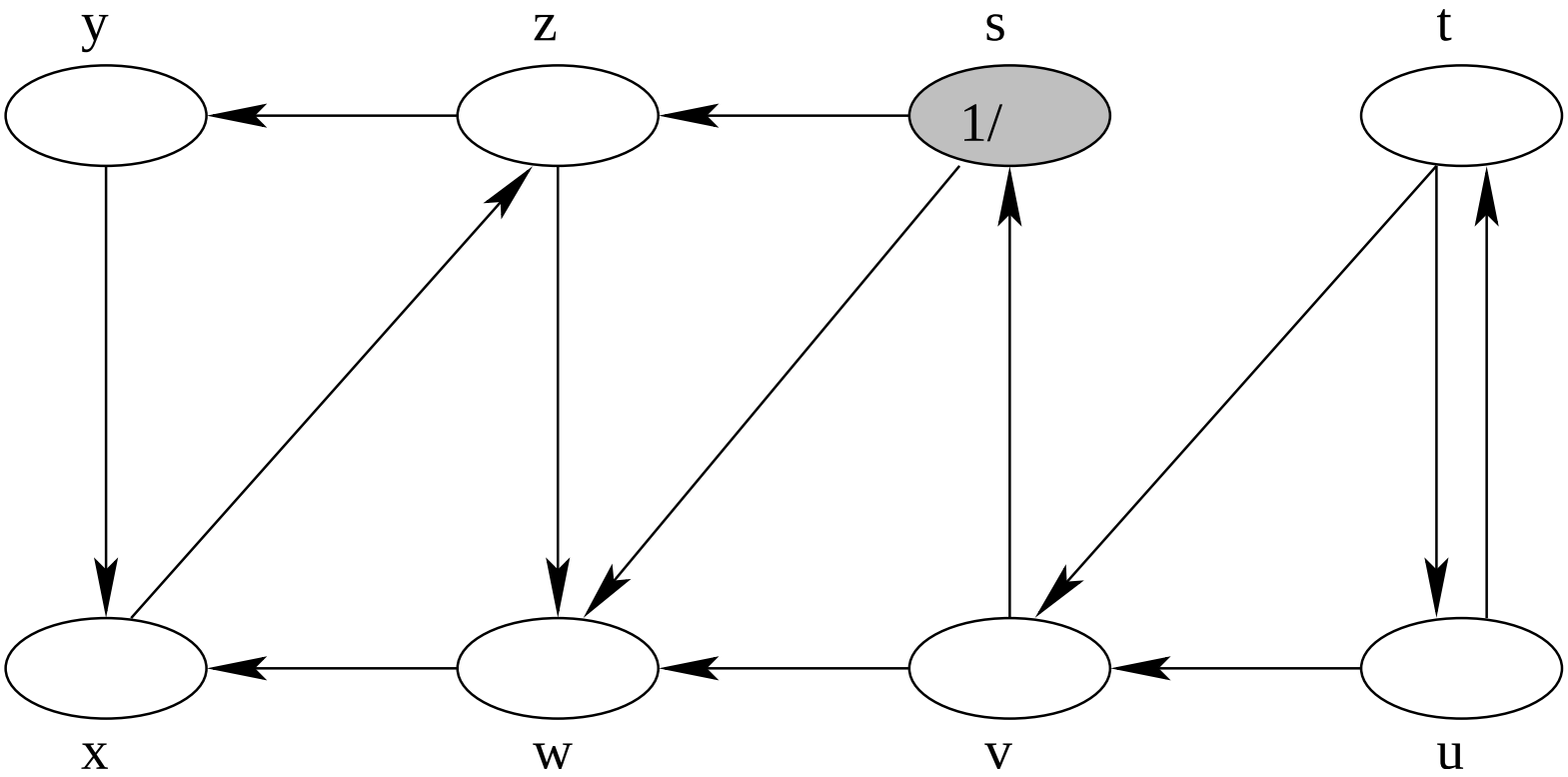
Estampas/rótulos

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

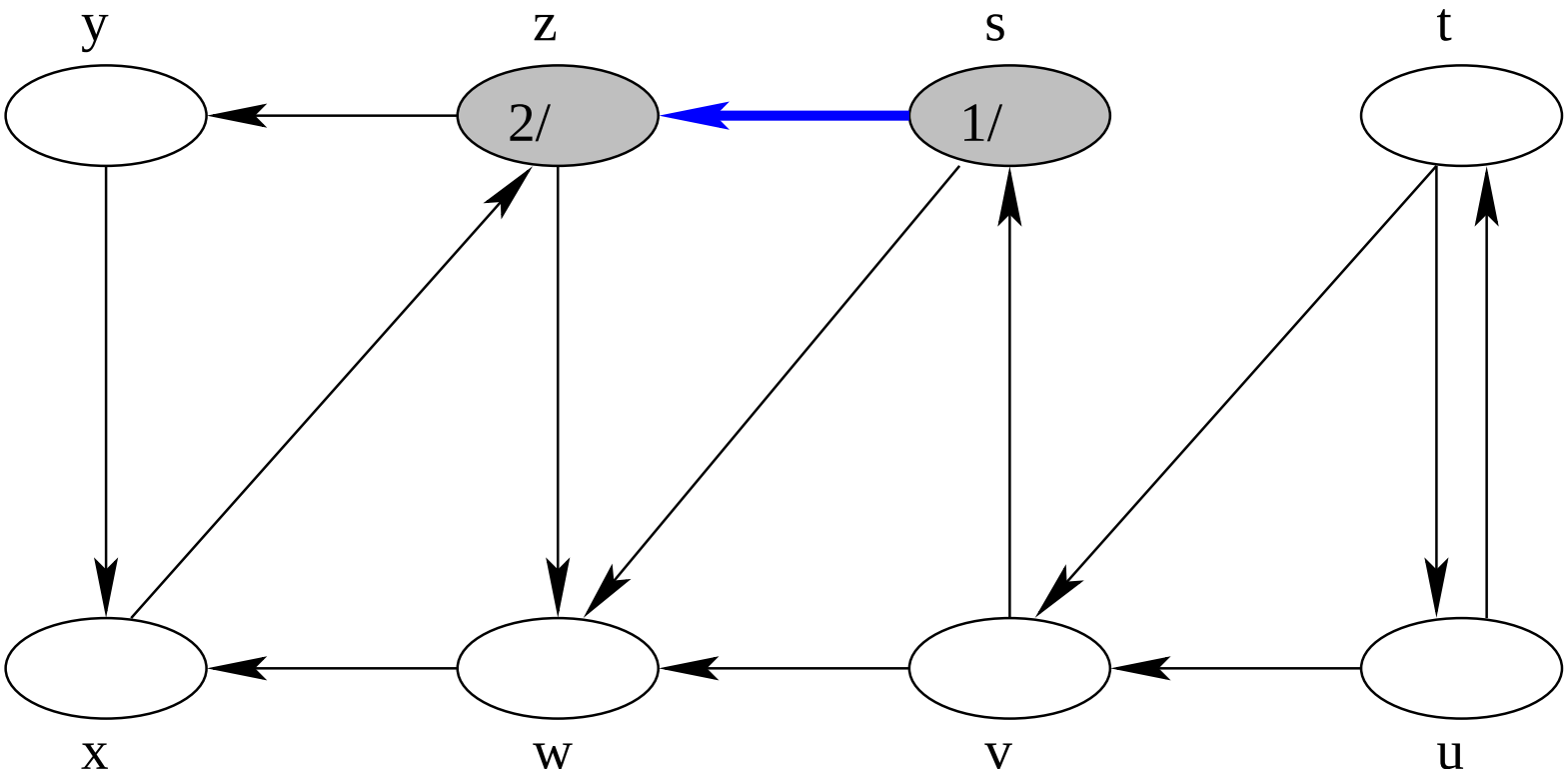
- $d[x]$: instante de descoberta de x.
  Neste instante x torna-se cinza.
- $f[x]$: instante de finalização de x.
  Neste instante x torna-se preto.

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

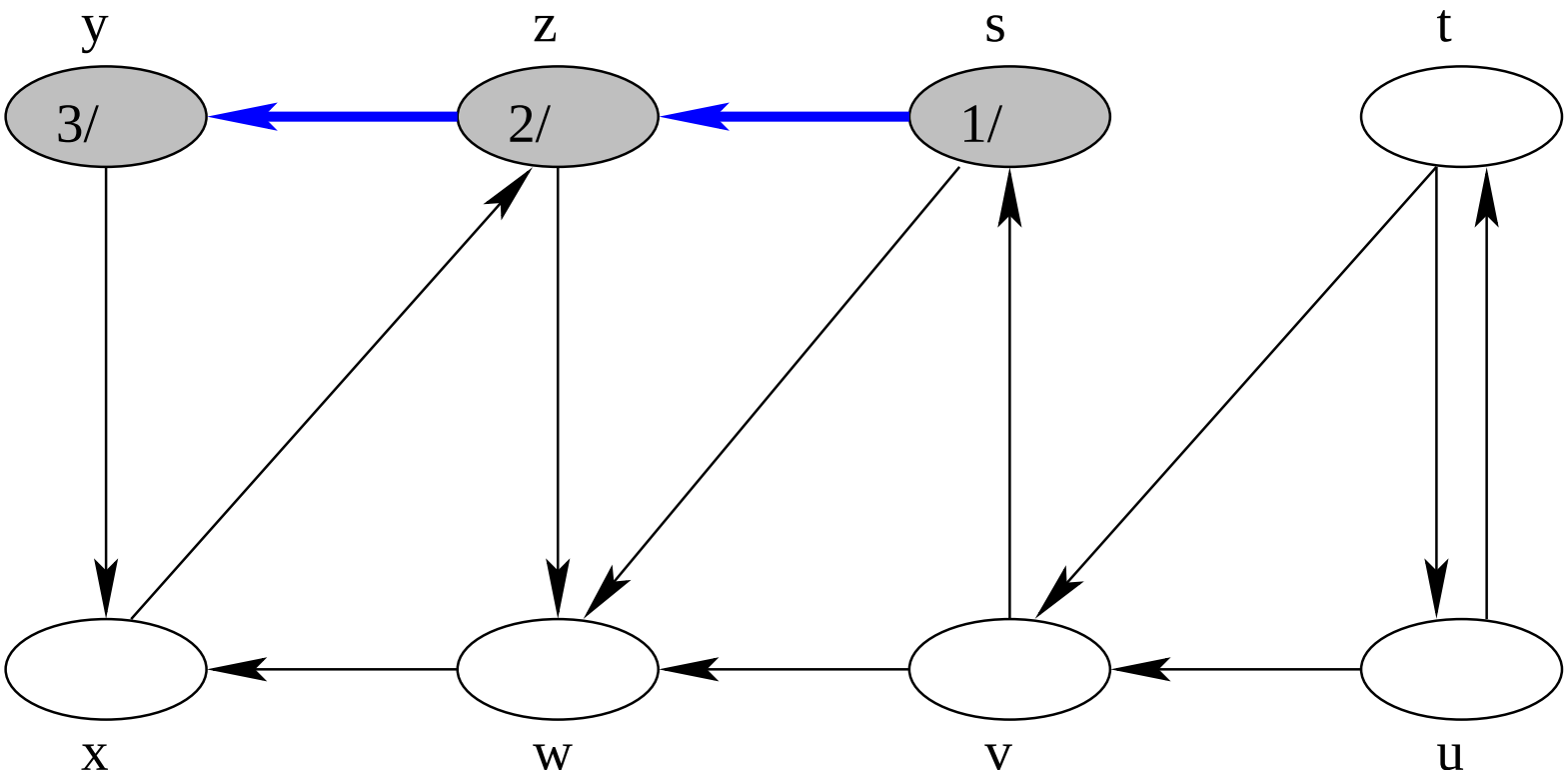
Exemplo DFS



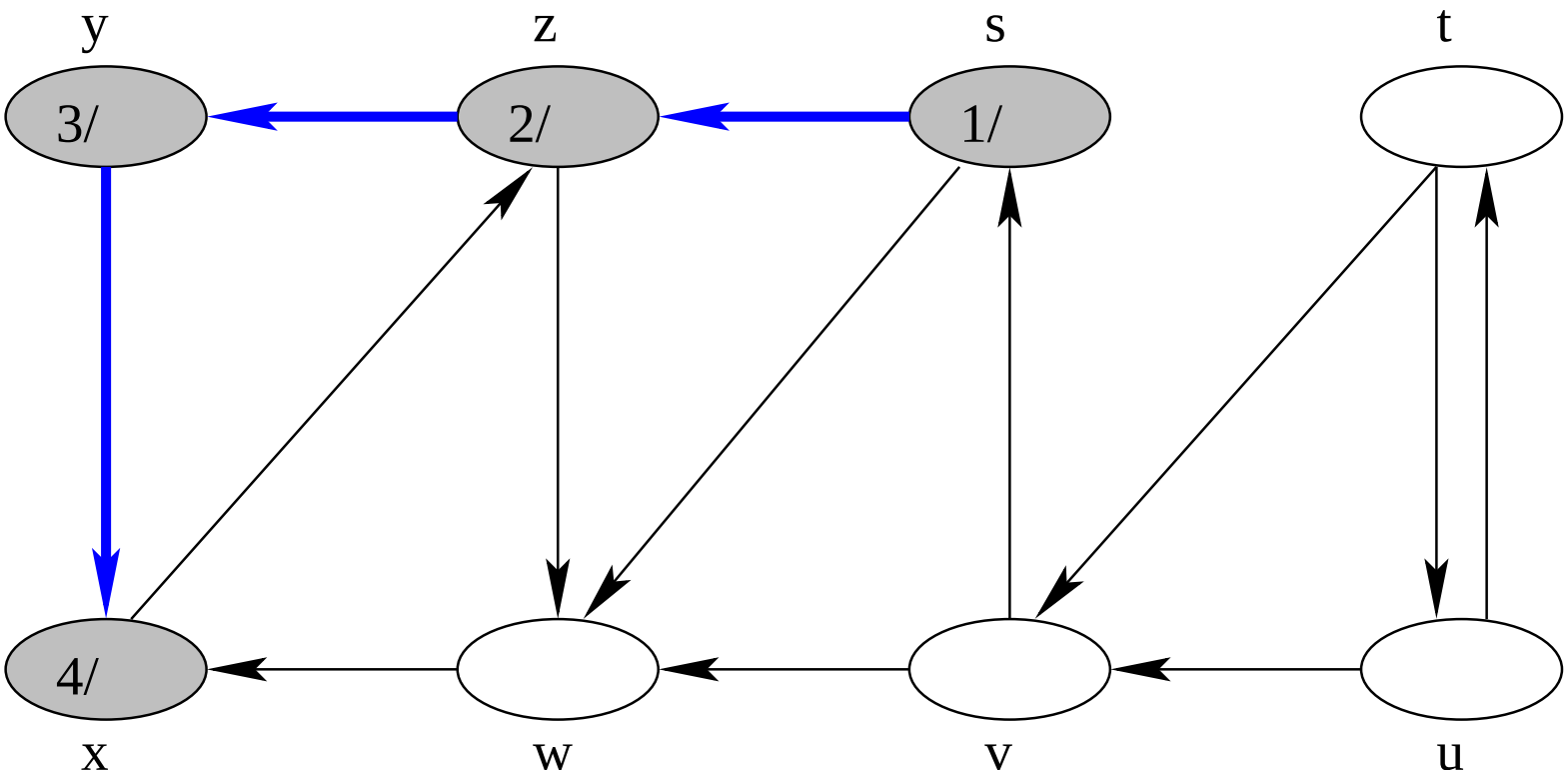
Exemplo DFS



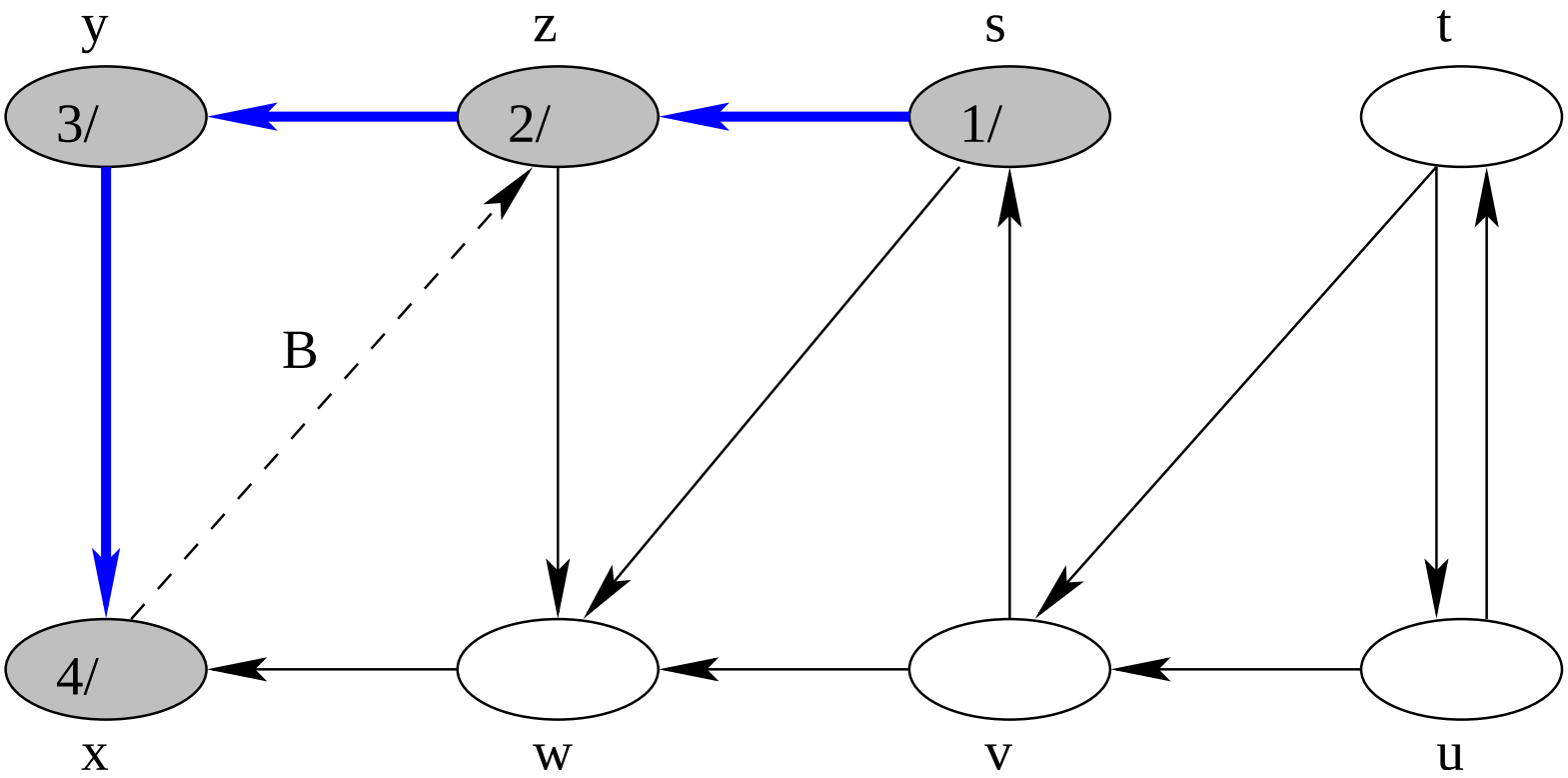
Exemplo DFS



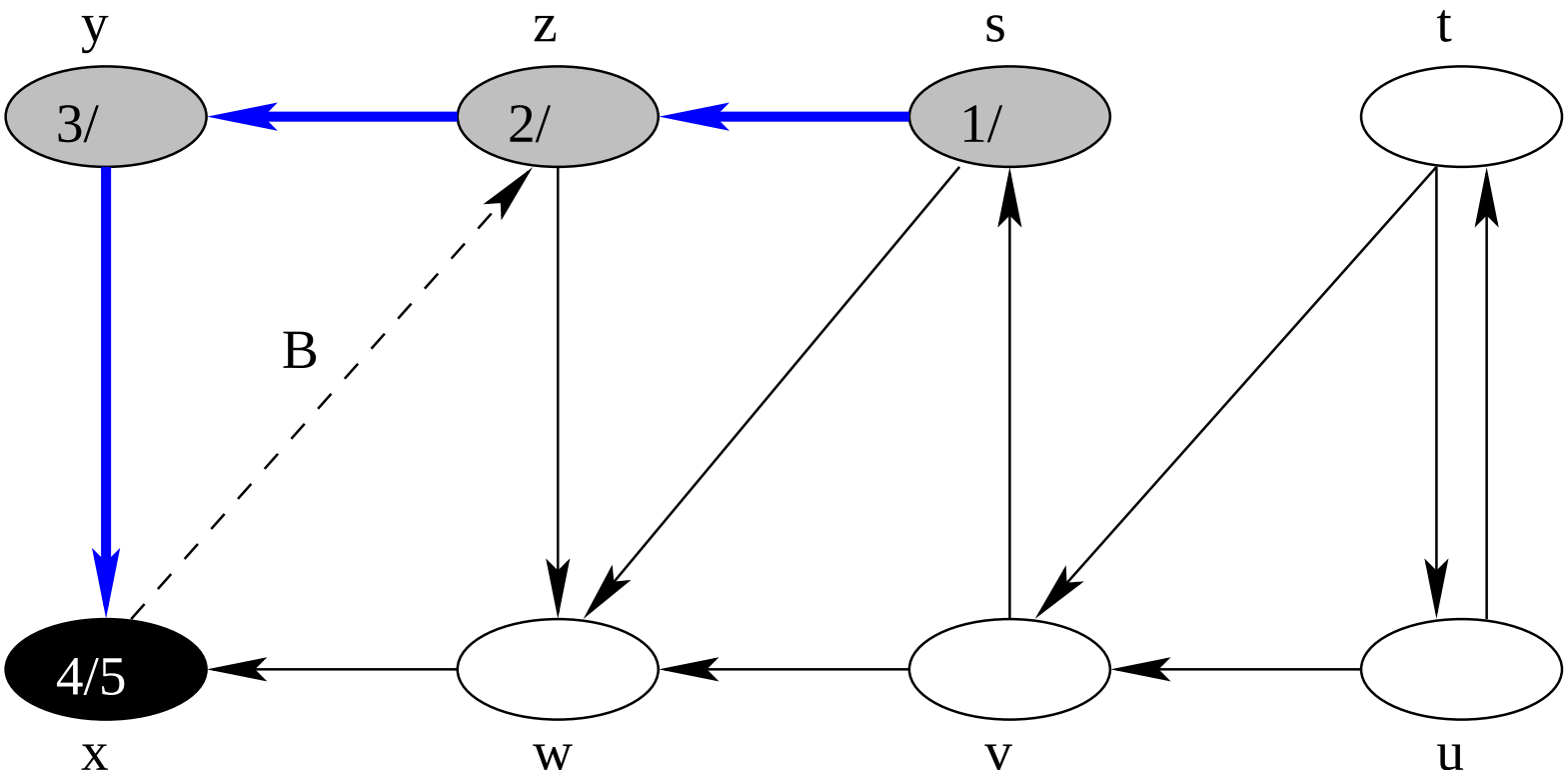
Exemplo DFS



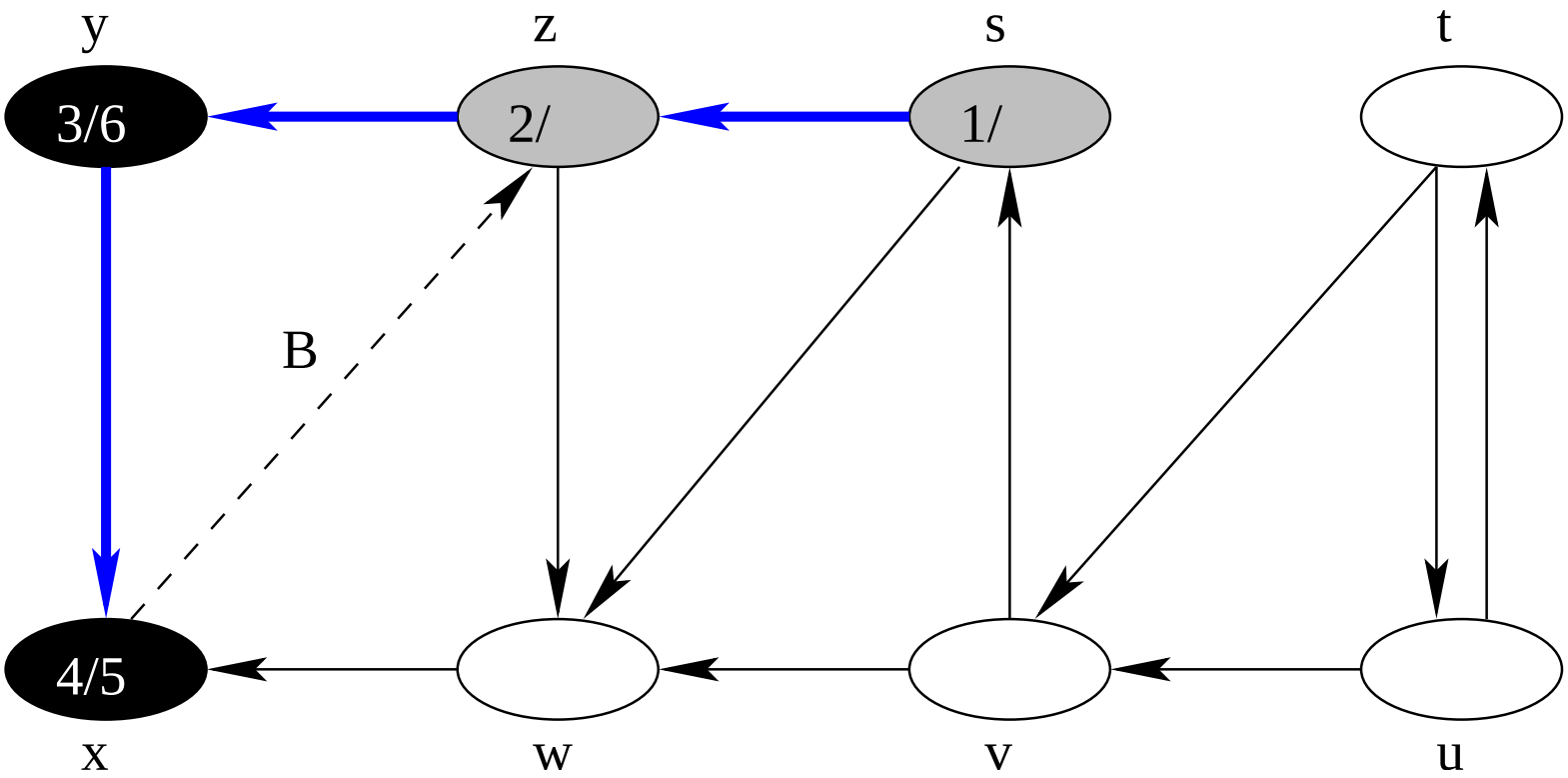
Exemplo DFS



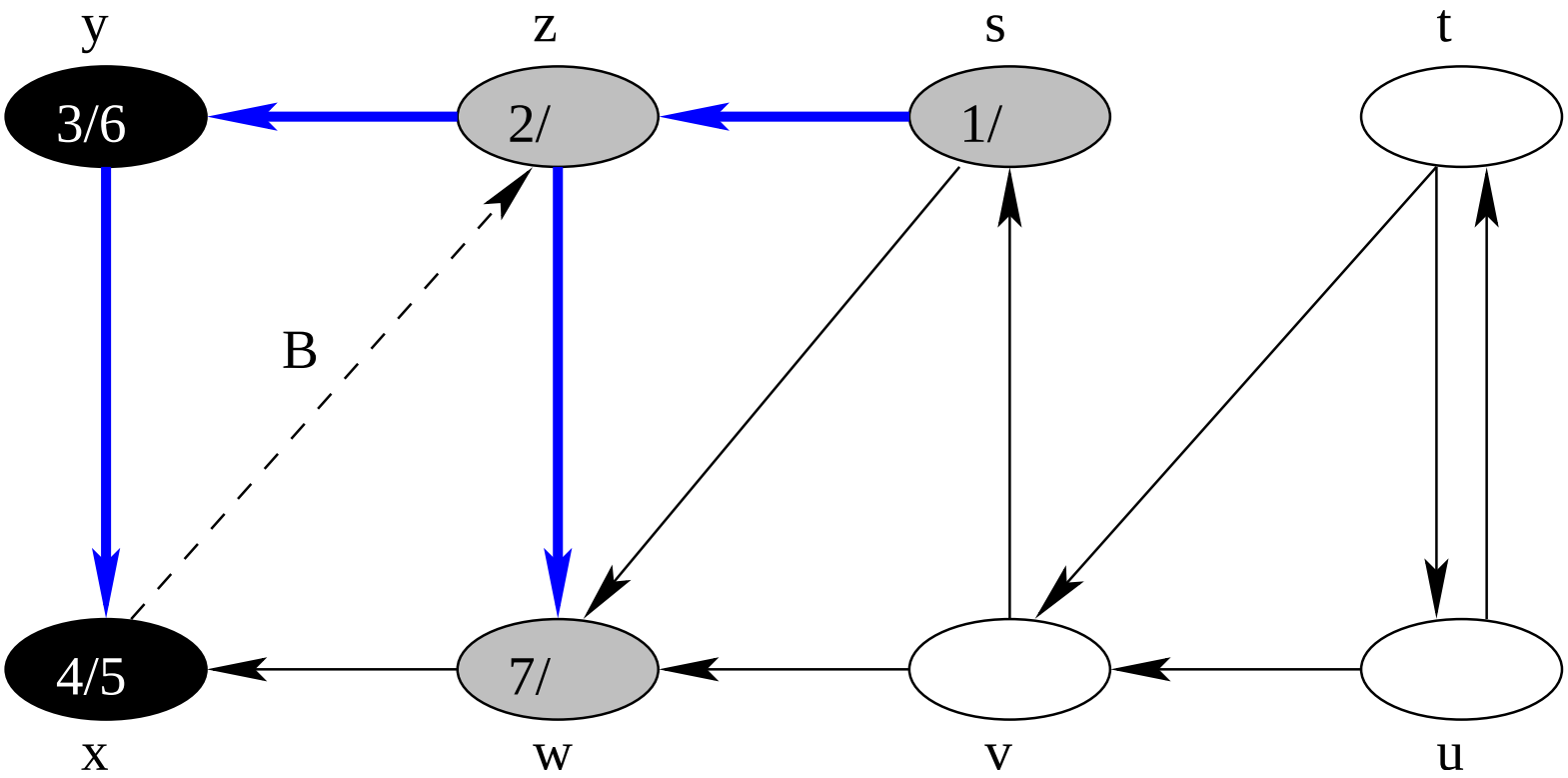
Exemplo DFS



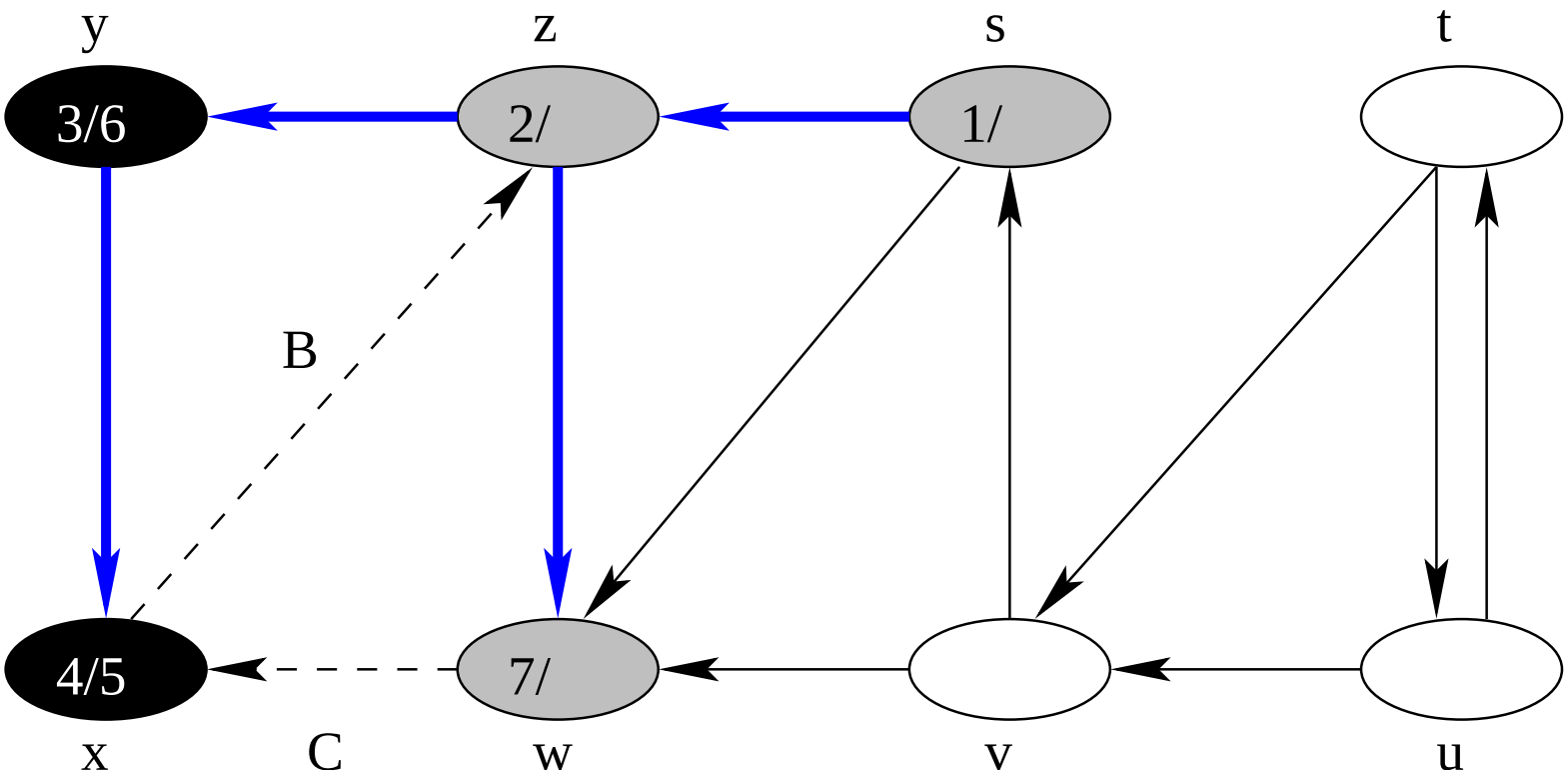
Exemplo DFS



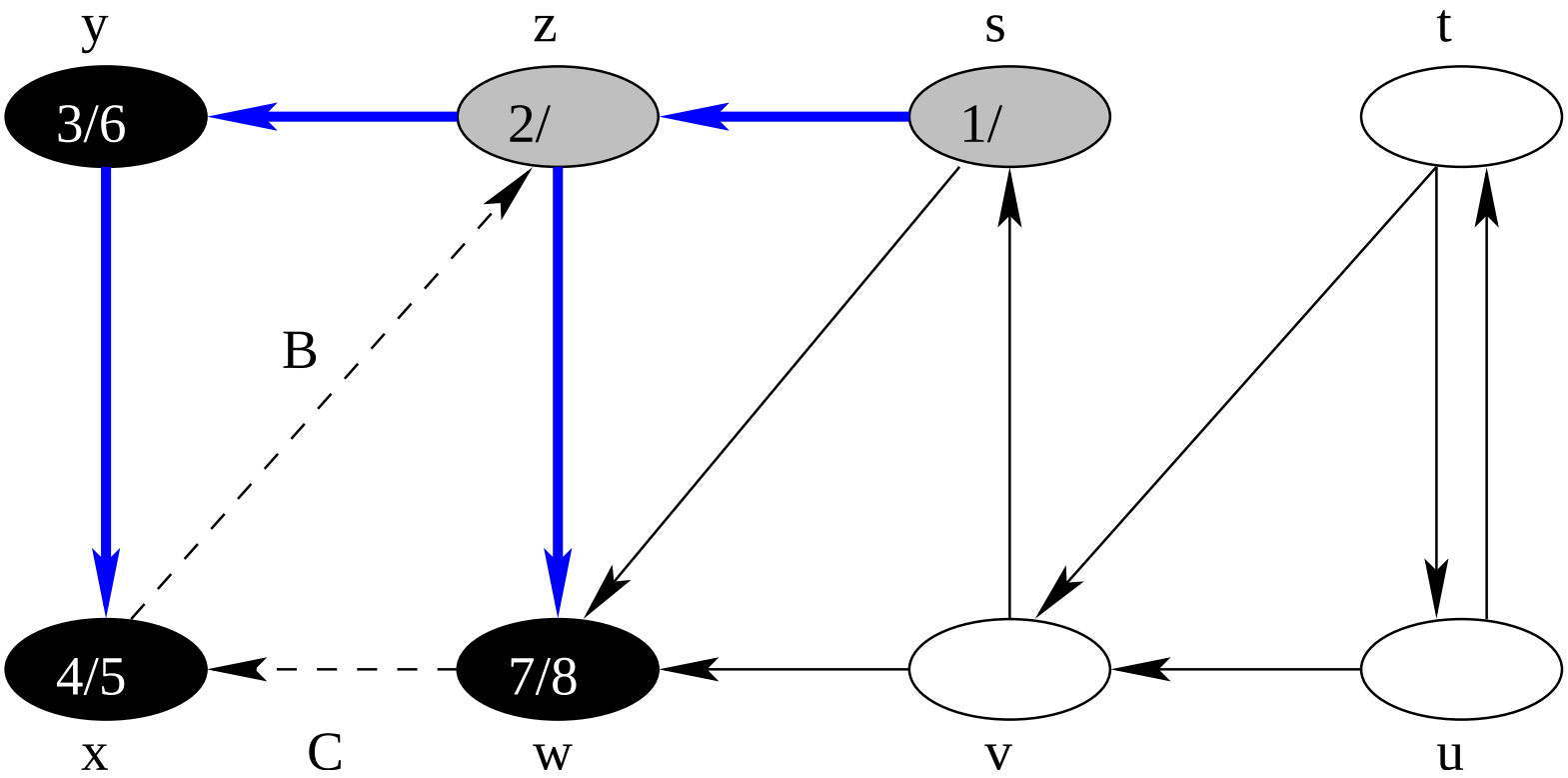
Exemplo DFS



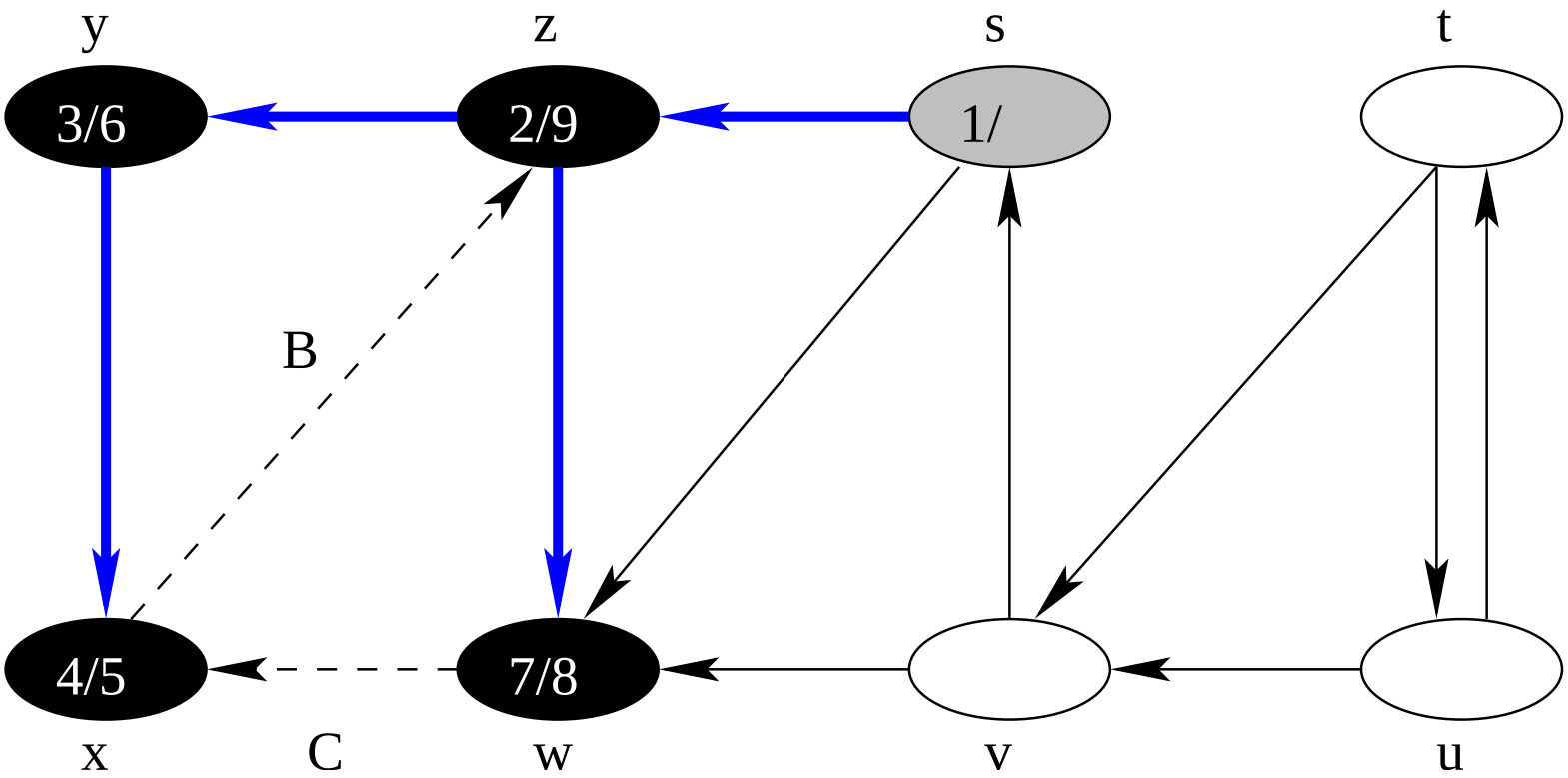
Exemplo DFS



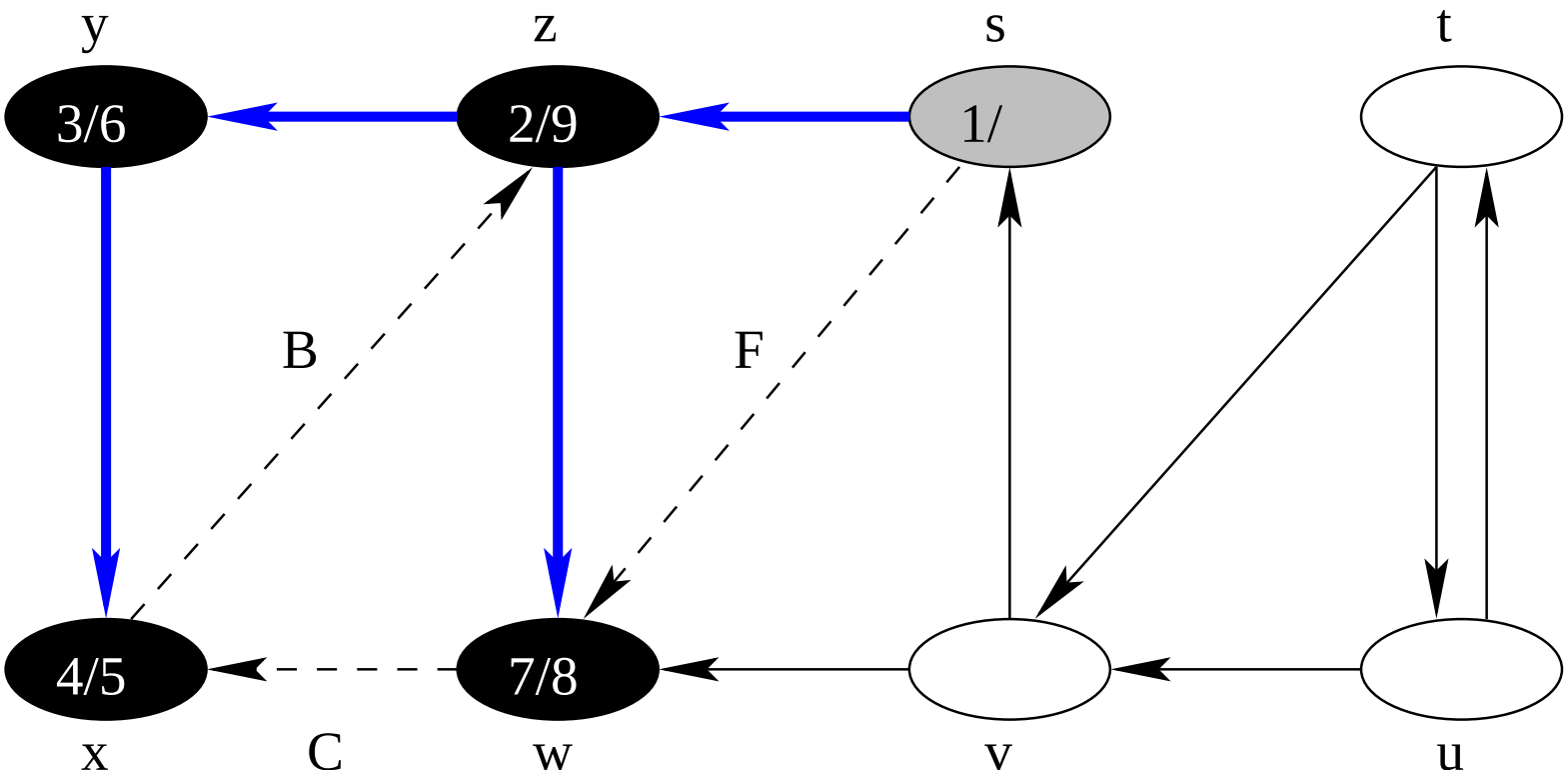
Exemplo DFS



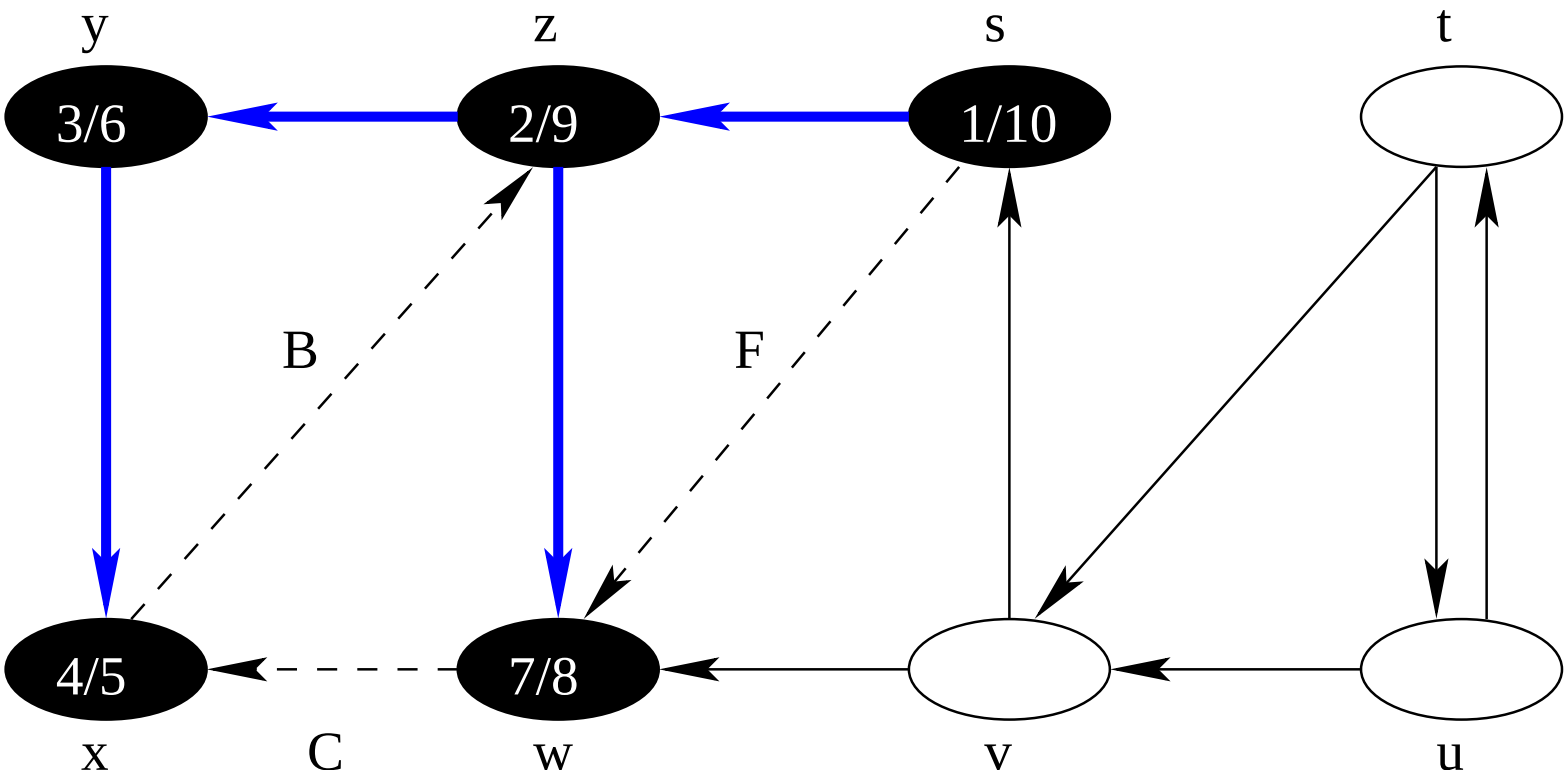
Exemplo DFS



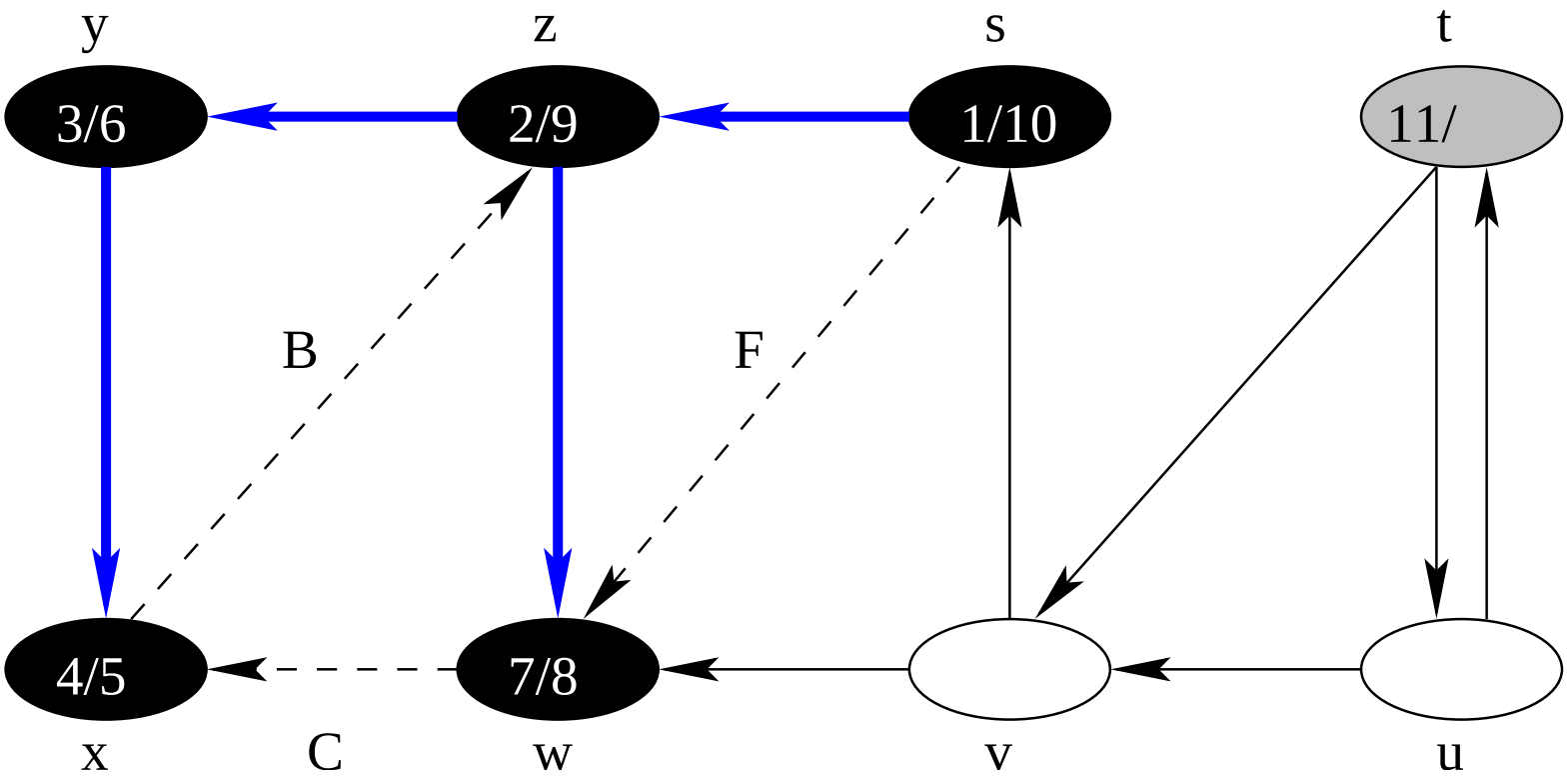
Exemplo DFS



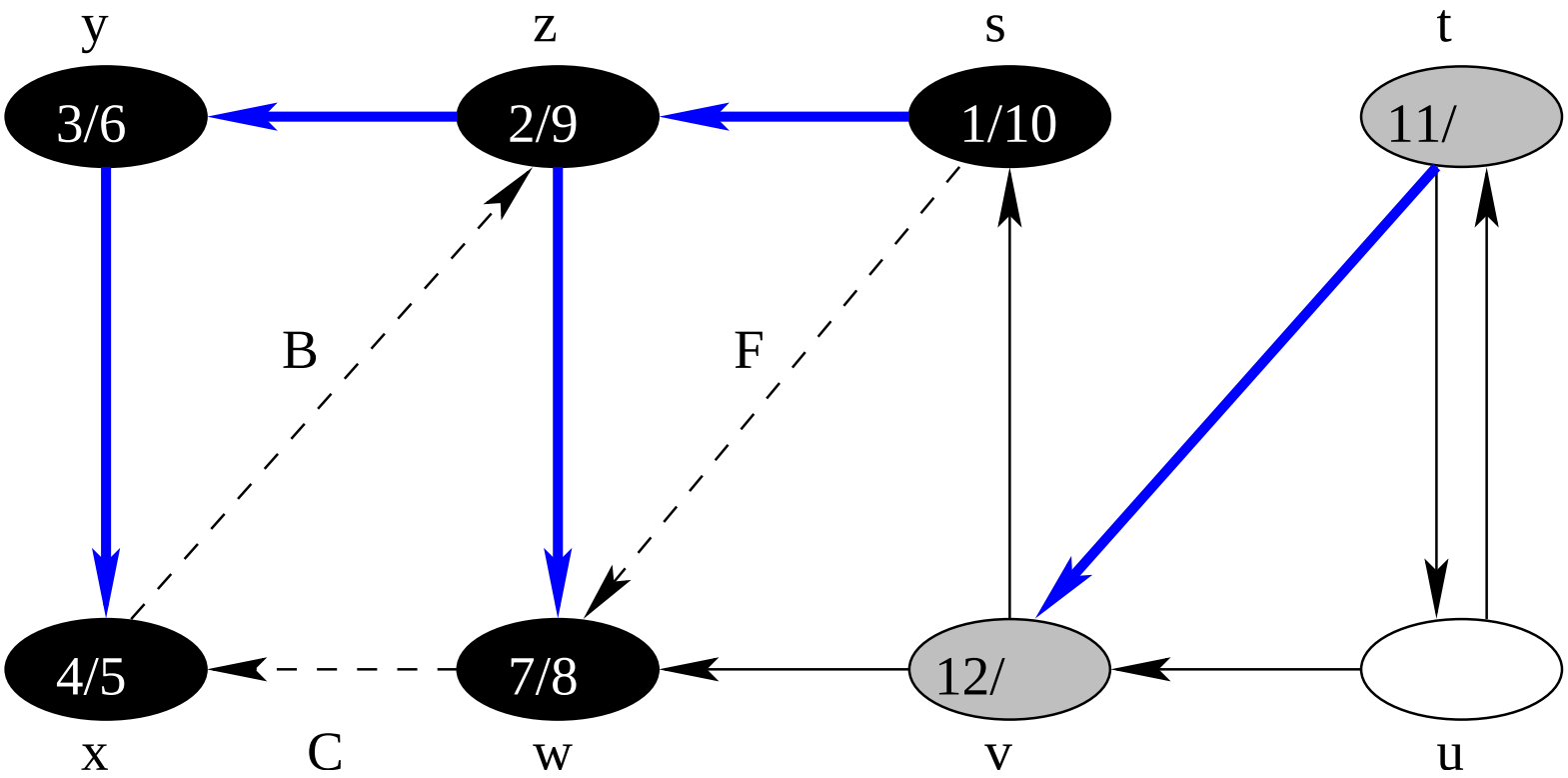
Exemplo DFS



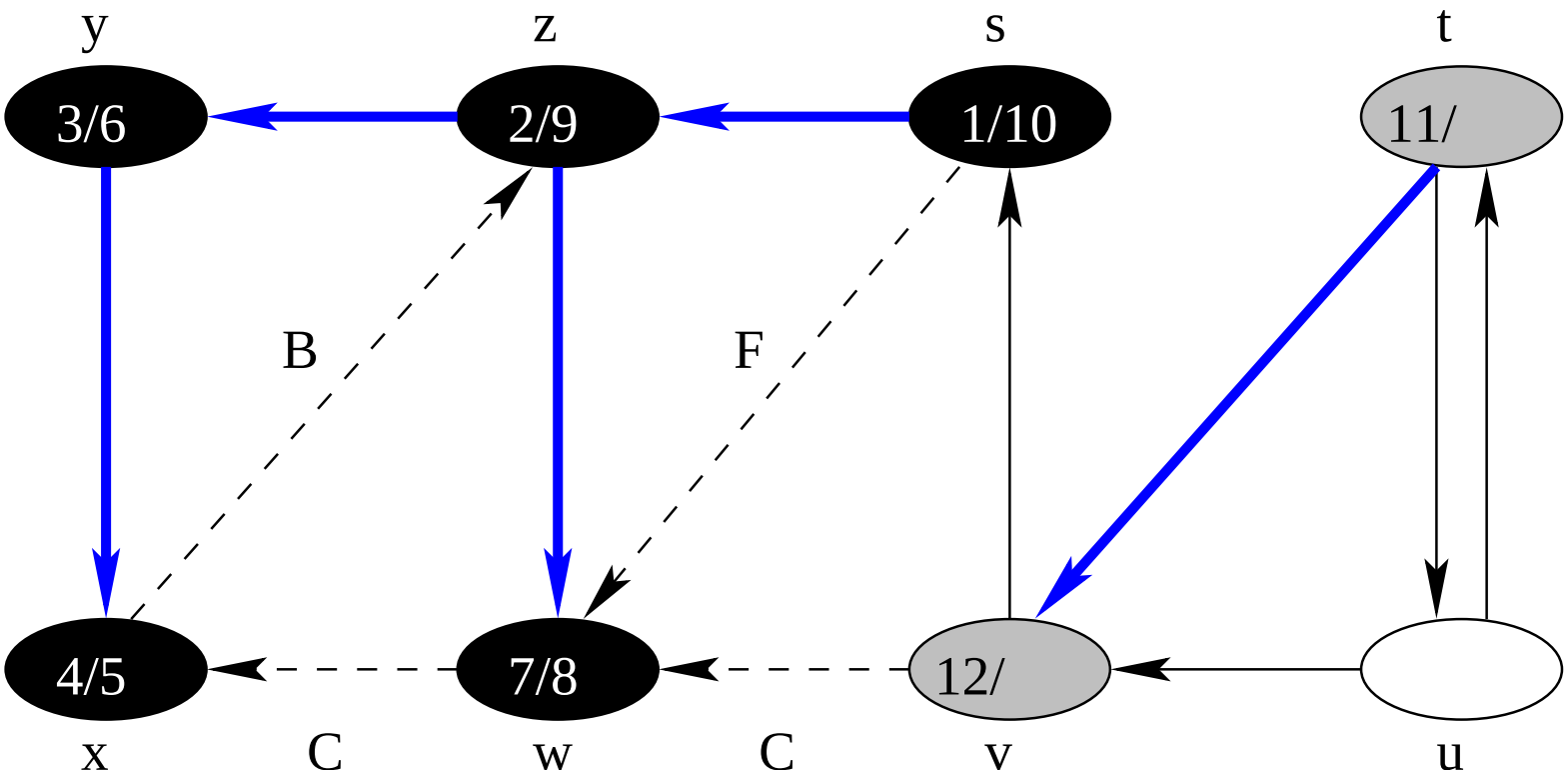
Exemplo DFS



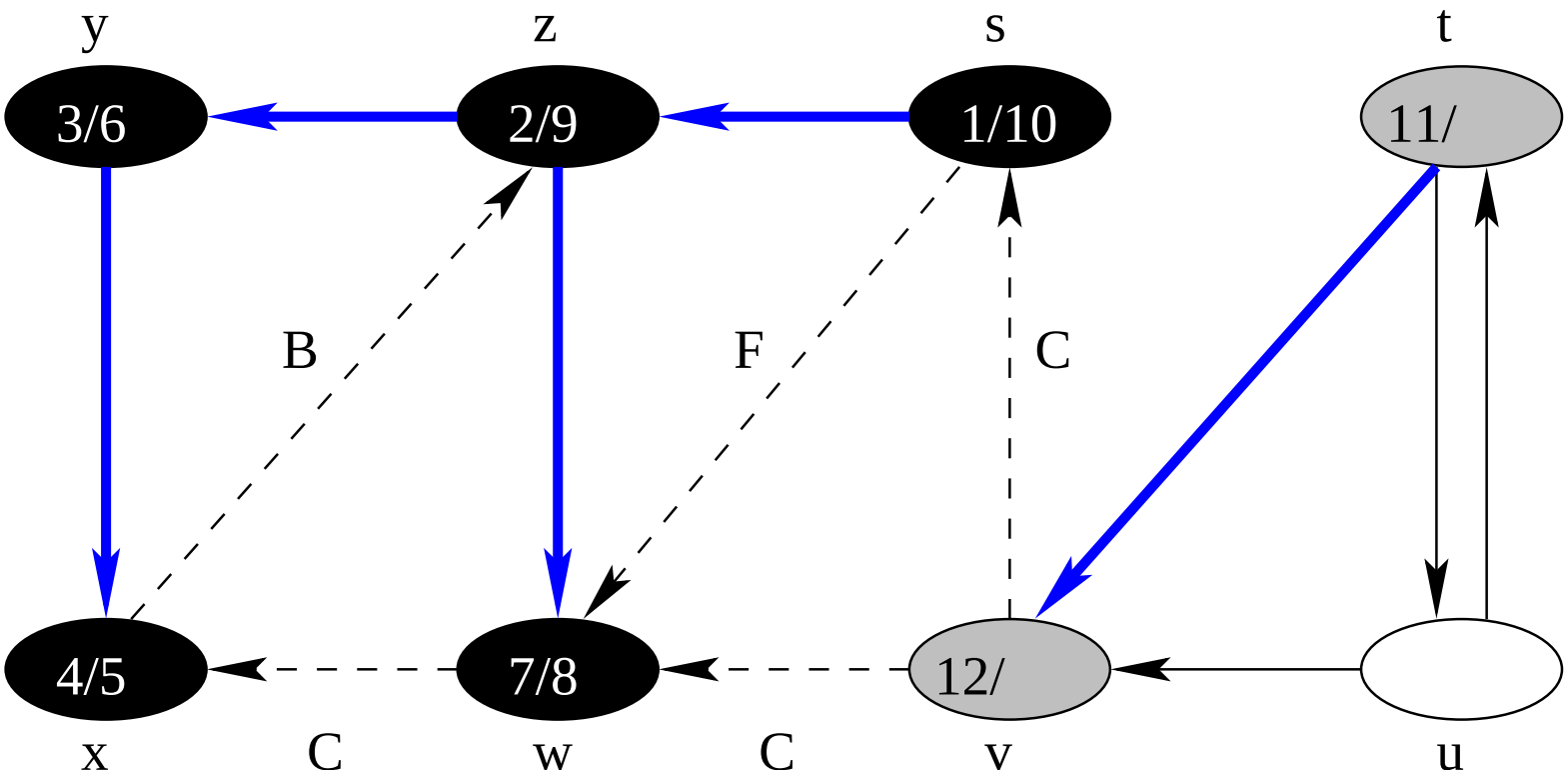
Exemplo DFS



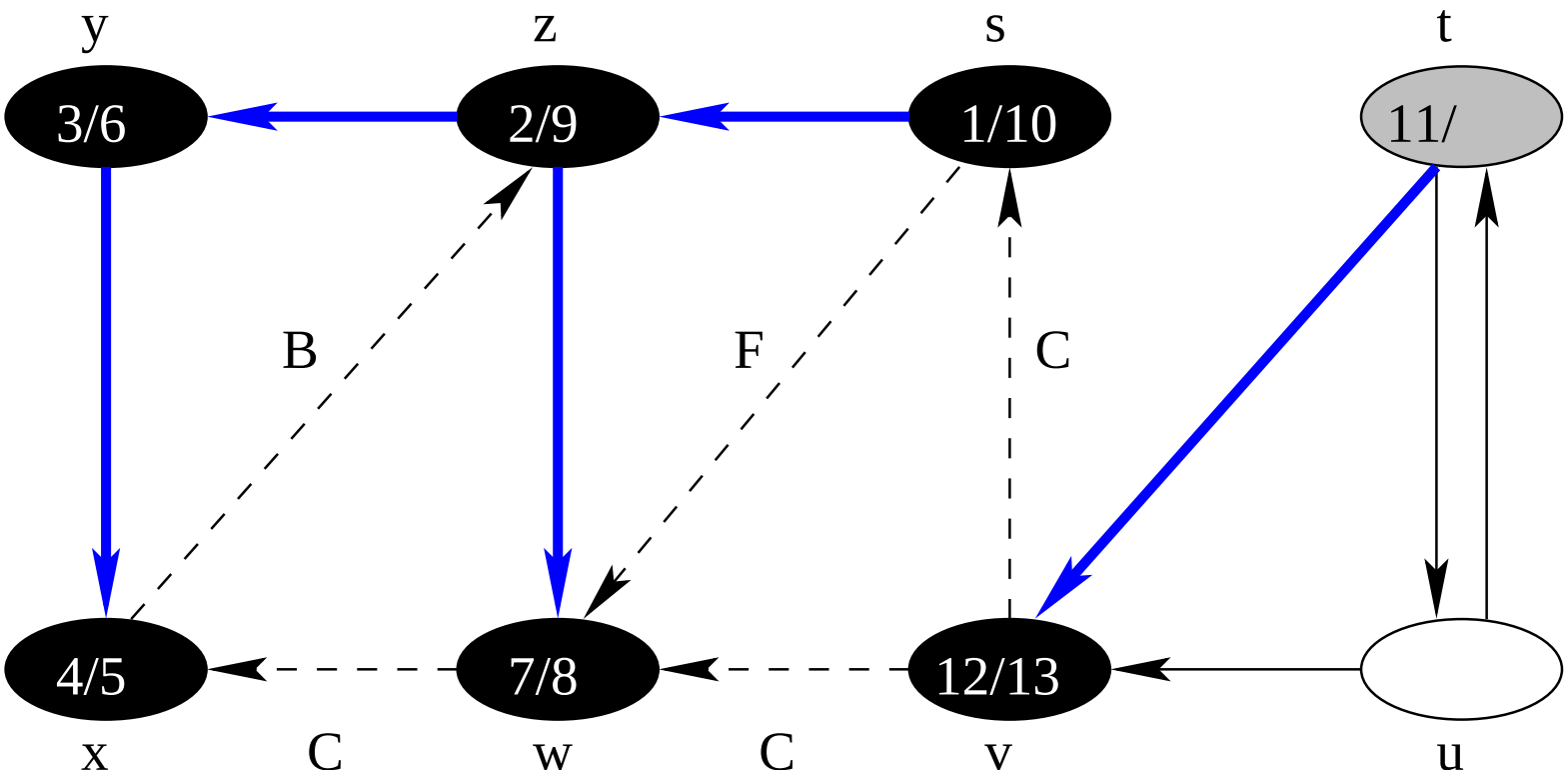
Exemplo DFS



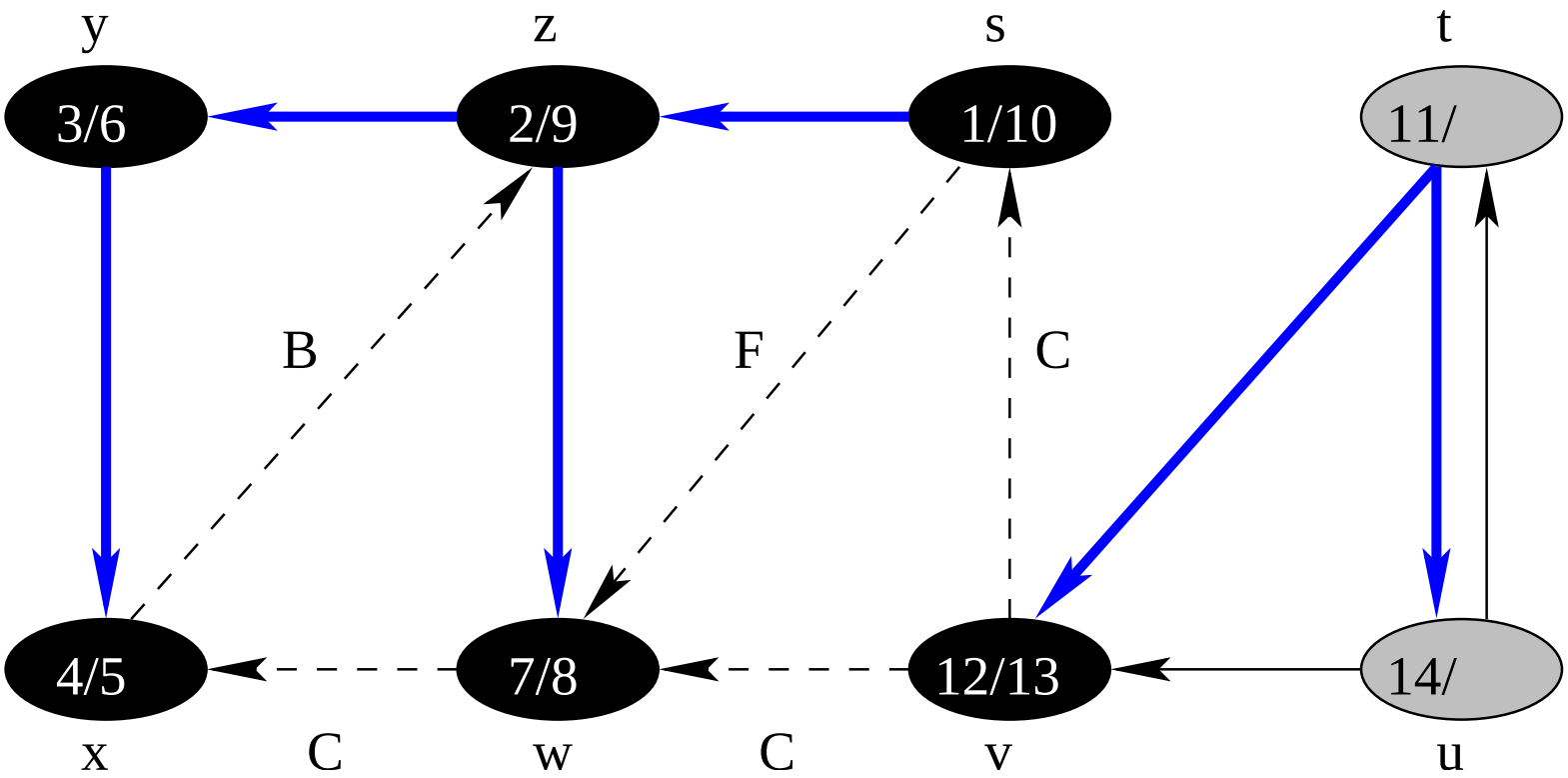
Exemplo DFS



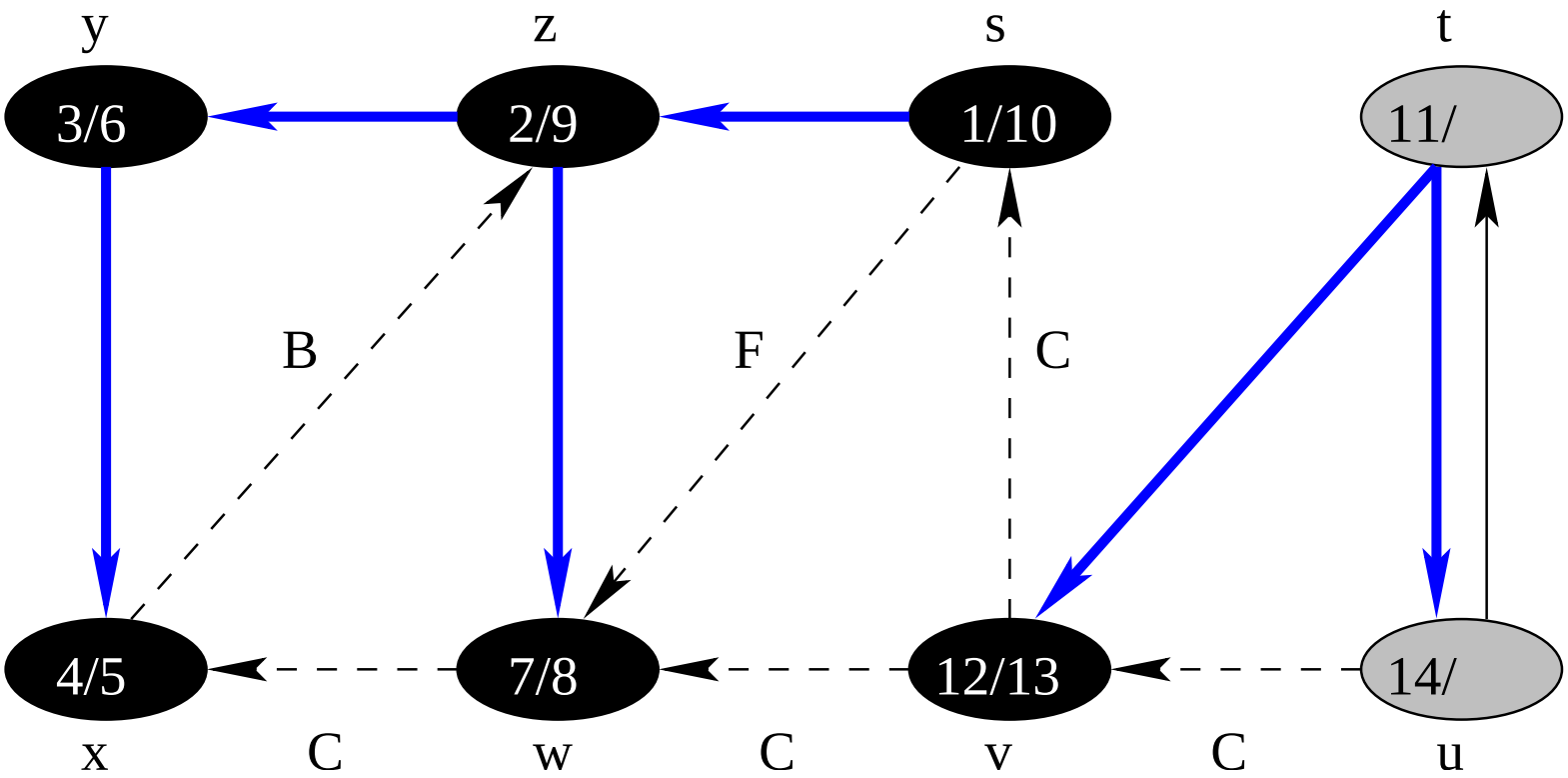
Exemplo DFS



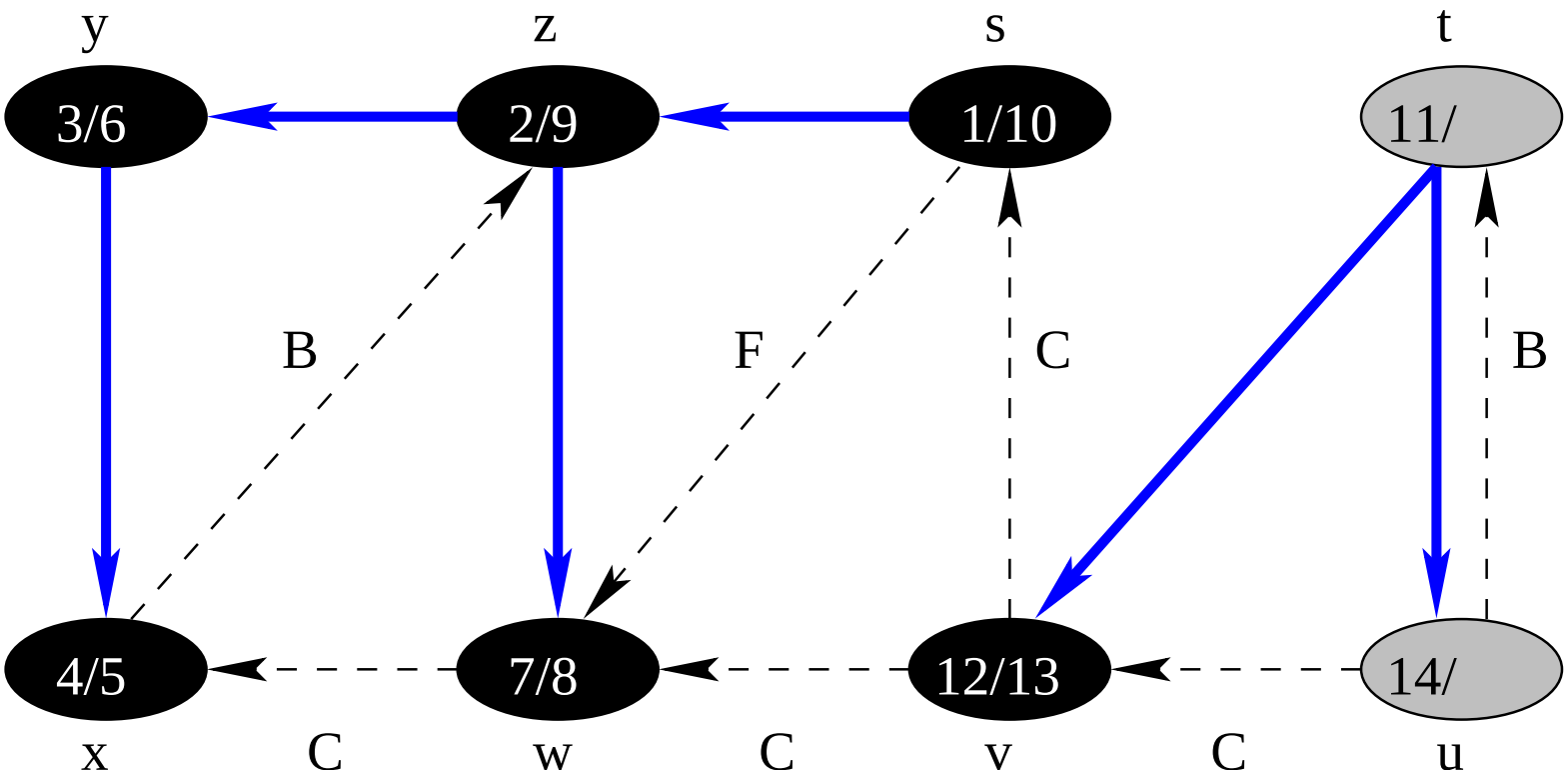
Exemplo DFS



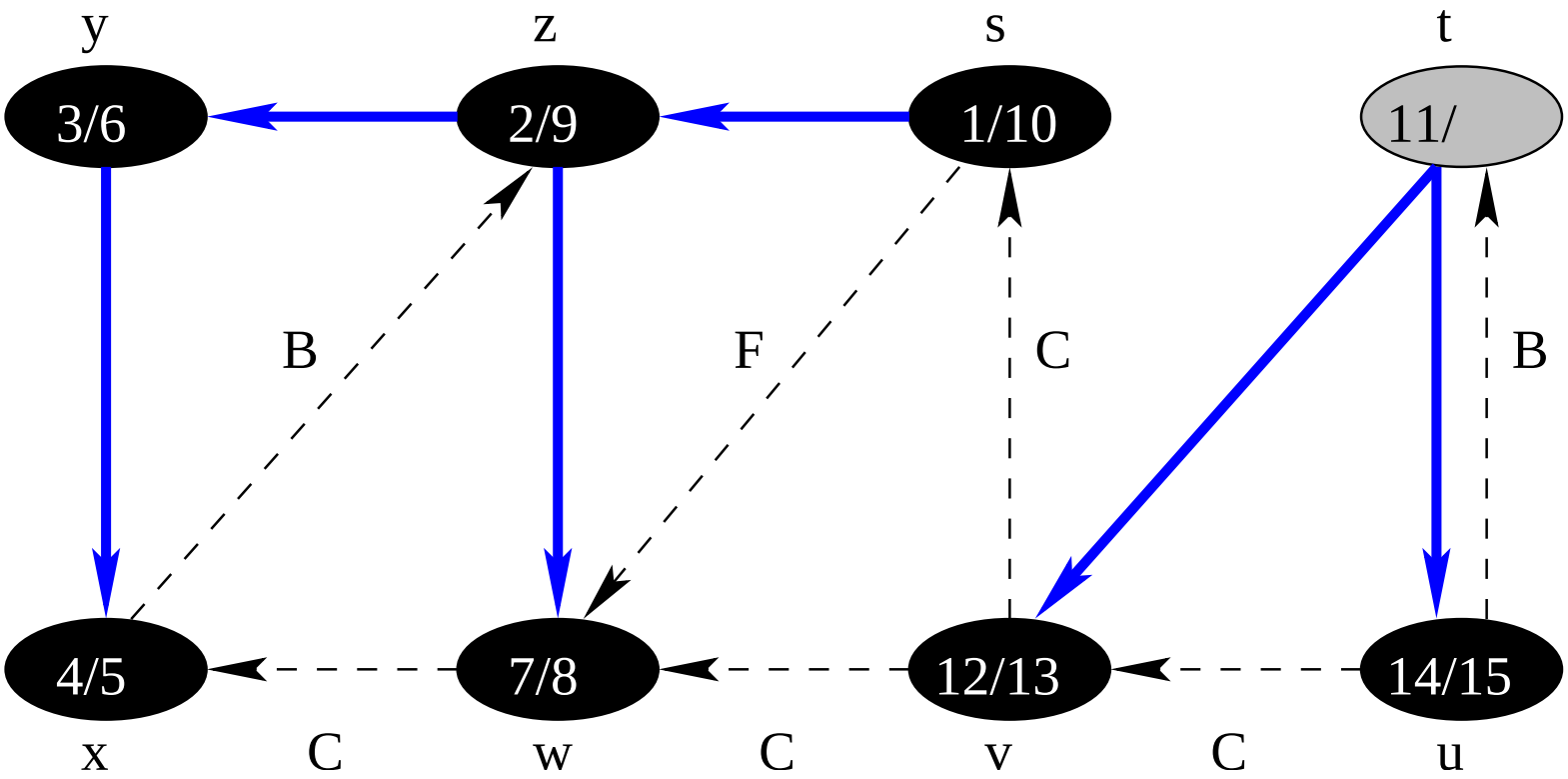
Exemplo DFS



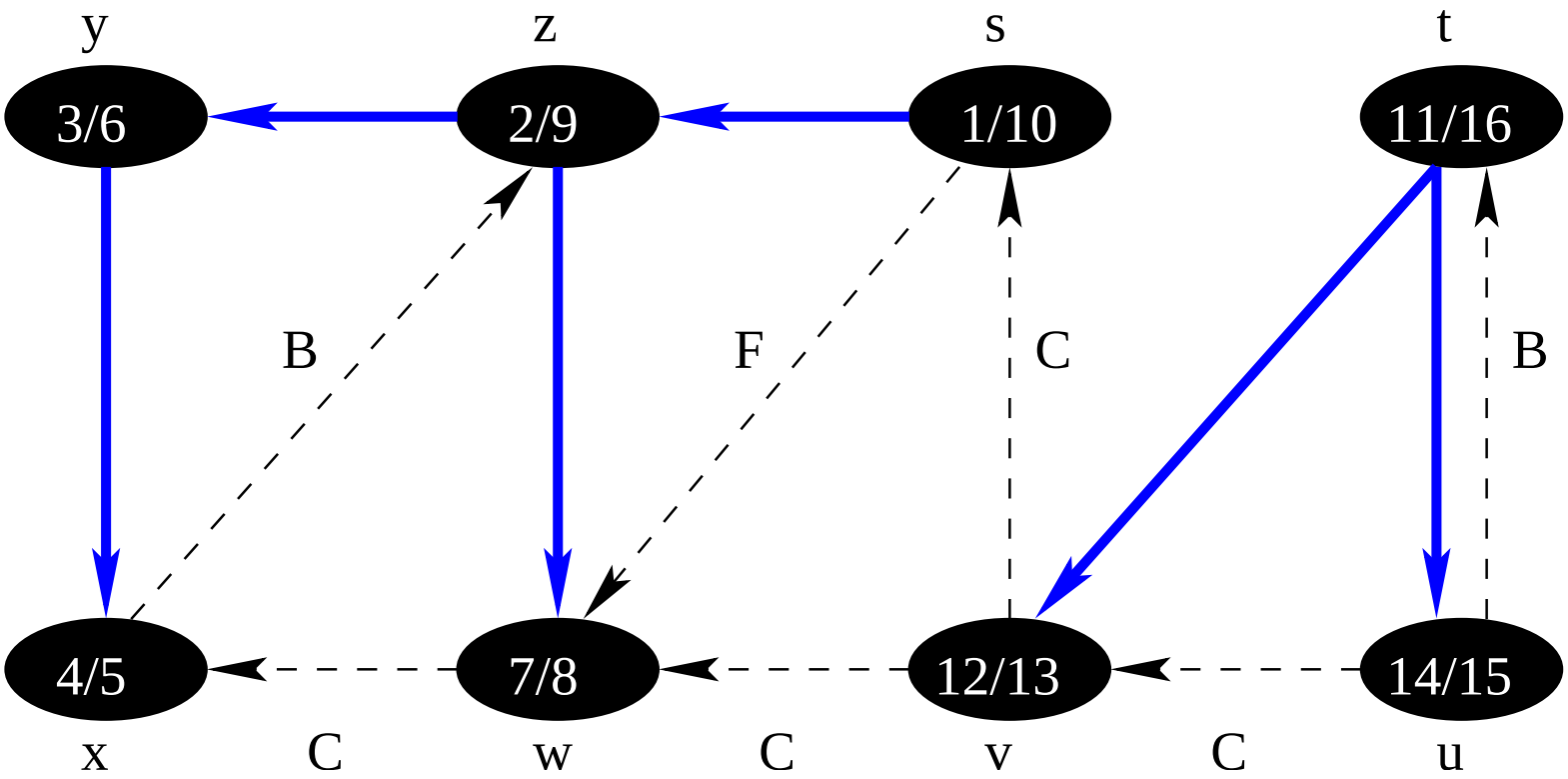
Exemplo DFS



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

Algoritmo DFS

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

Consumo de tempo

$O(V)$ + no máximo V chamadas a DFS-VISIT().

Algoritmo DFS-Visit

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

Consumo de tempo

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

Complexidade de DFS

- DFS-VISIT(u) é executado no máximo uma vez para cada $u \in V$.
- Em uma execução de DFS-VISIT(u), o laço das linhas 4-7 é executado $|\mathrm{Adj}[u]|$ vezes.
  Assim, o custo total de todas as chamadas é

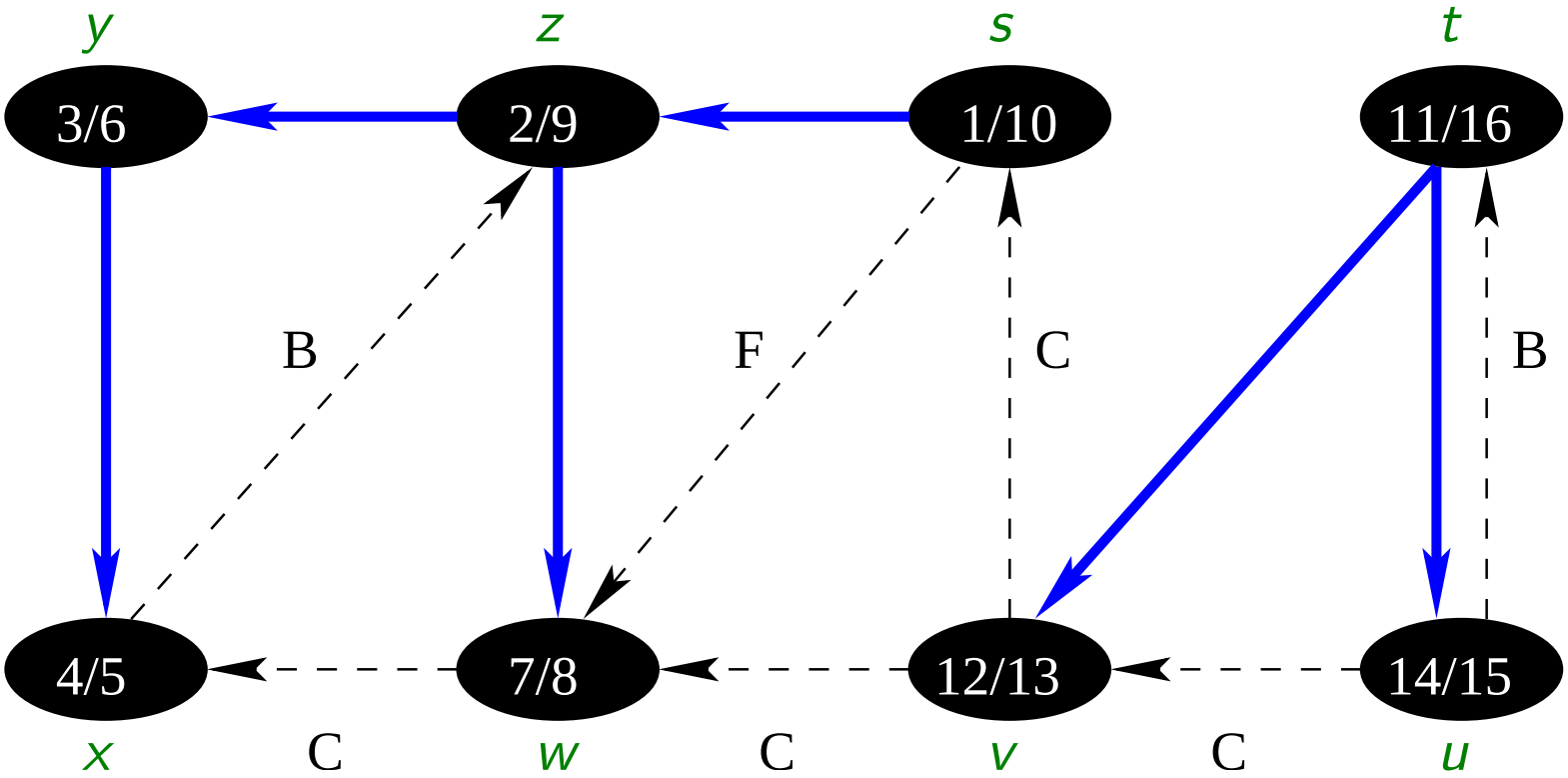
  $O(\sum_{v \in V} |\mathrm{Adj}(v)|) = O(E)$.

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

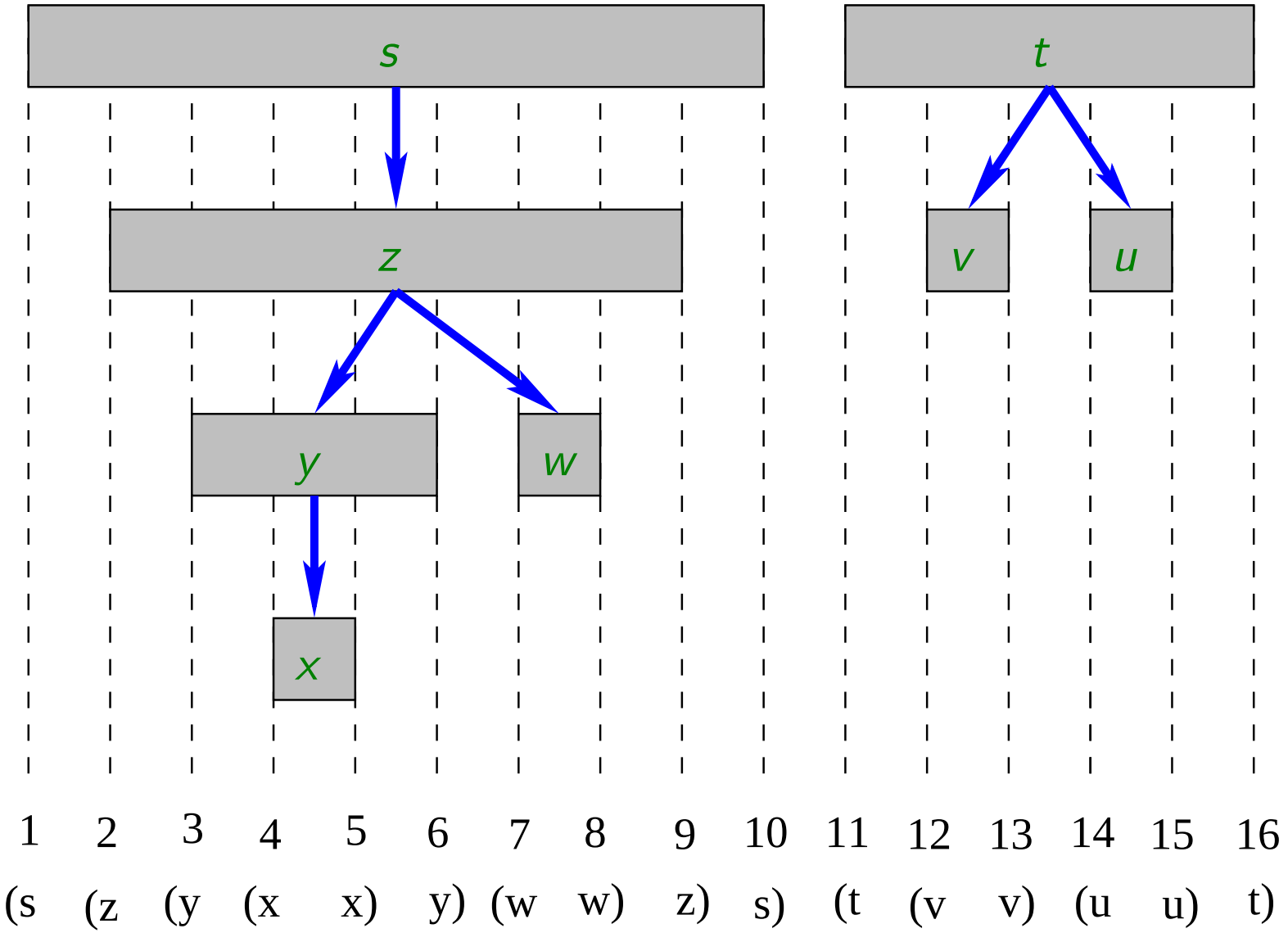
Estrutura de parênteses

- 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:

- $d[v] < f[u]$: 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 ($\mathrm{cor}[v] = \texttt{preto}$) antes da busca voltar a u. Logo, o intervalo $[d[v], f[v]]$ está contido em $[d[u], f[u]]$.
- $f[u] < d[v]$: 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)

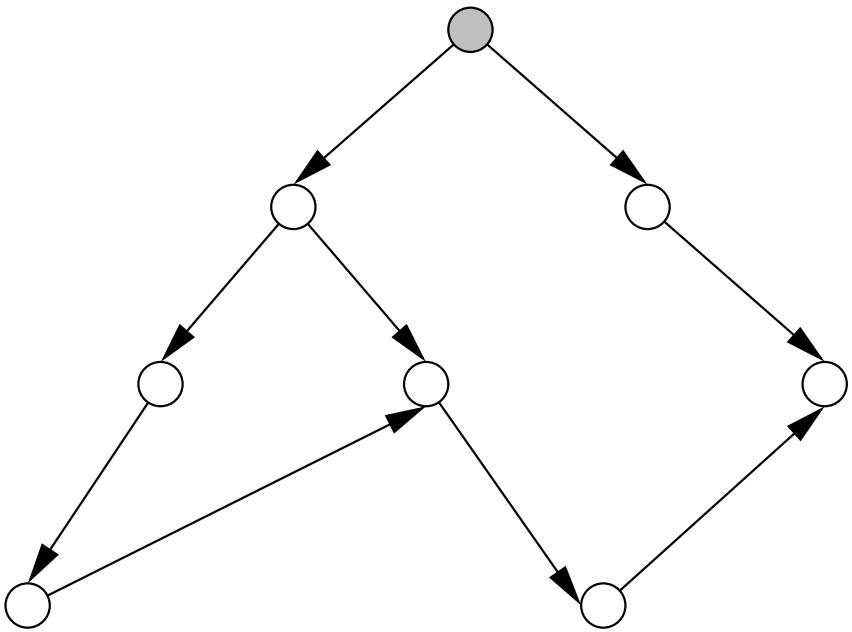
Um vértice v é um descendente próprio de u na Floresta de BP se, e somente se, $d[u] < d[v] < f[v] < f[u]$.

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).

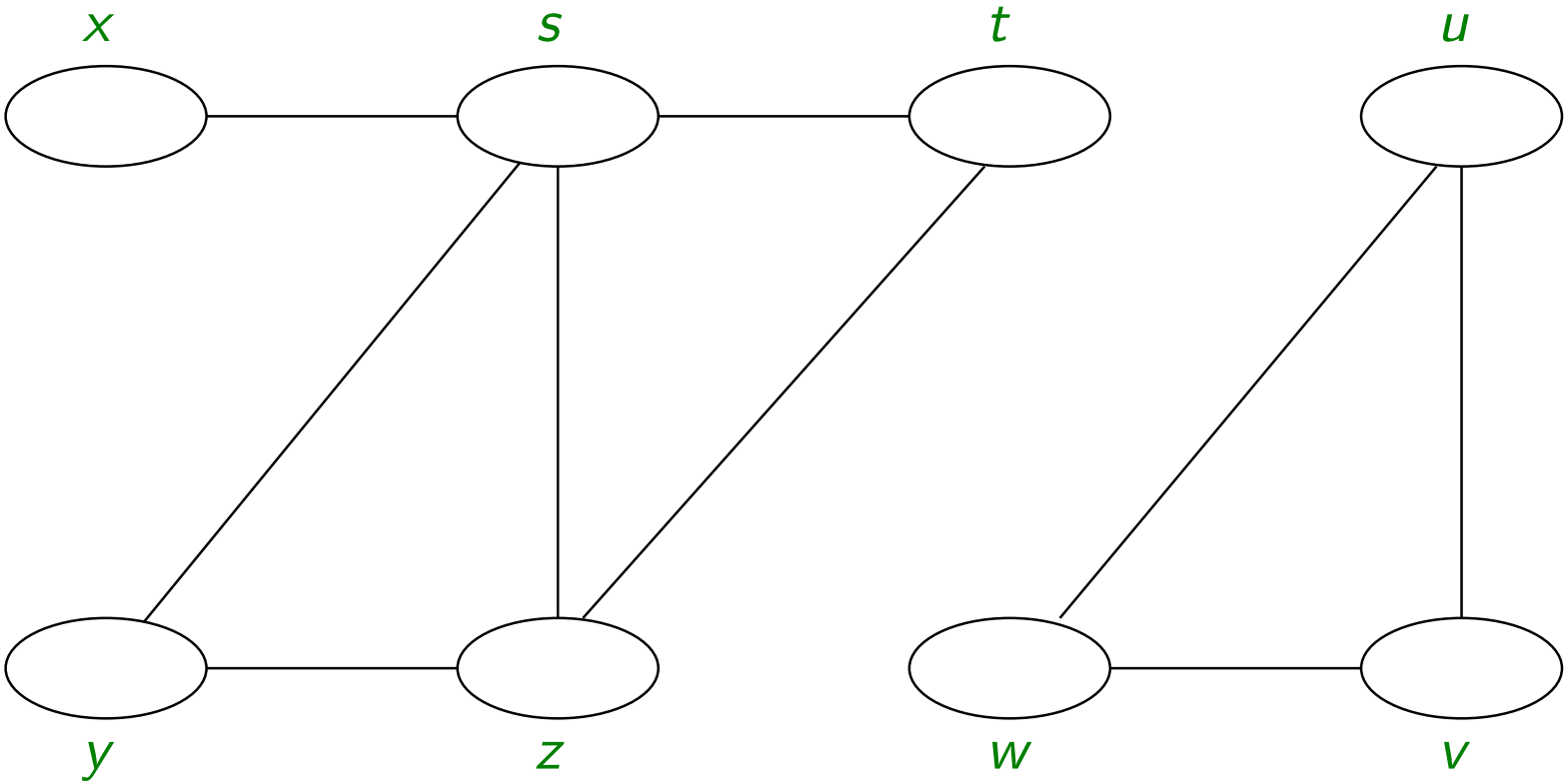
Teorema do Caminho Branco

$\Leftarrow$: 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 precede imediatamente v nesse caminho. Assim, $f[w] < f[u]$. Como v é descoberto depois de u, mas antes de w ser finalizado, temos

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

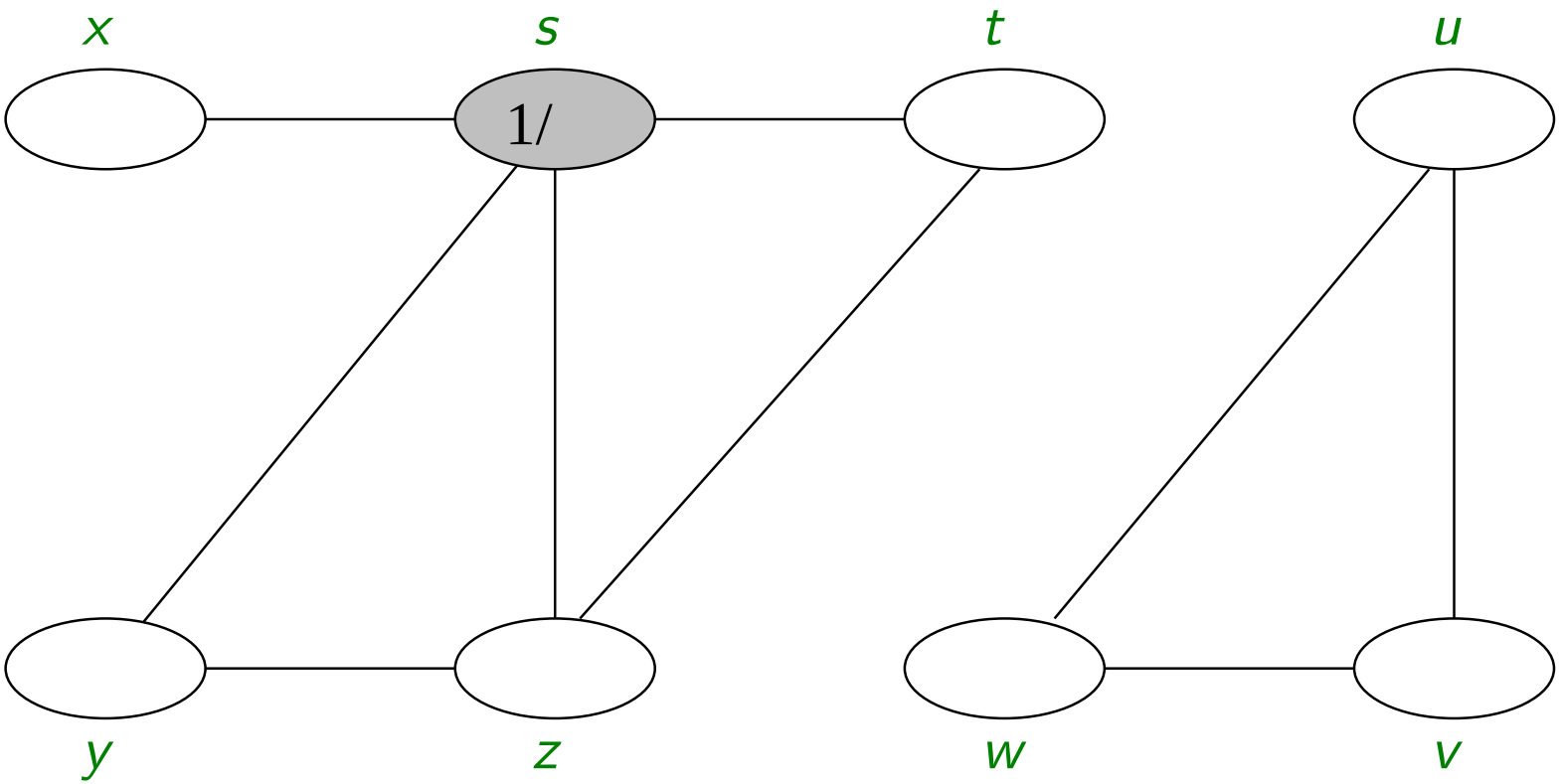
Logo, o intervalo $[d[v], f[v]]$ está contido em $[d[u], f[u]]$ e v é descendente de u, uma contradição.

Aplicação: componentes conexos

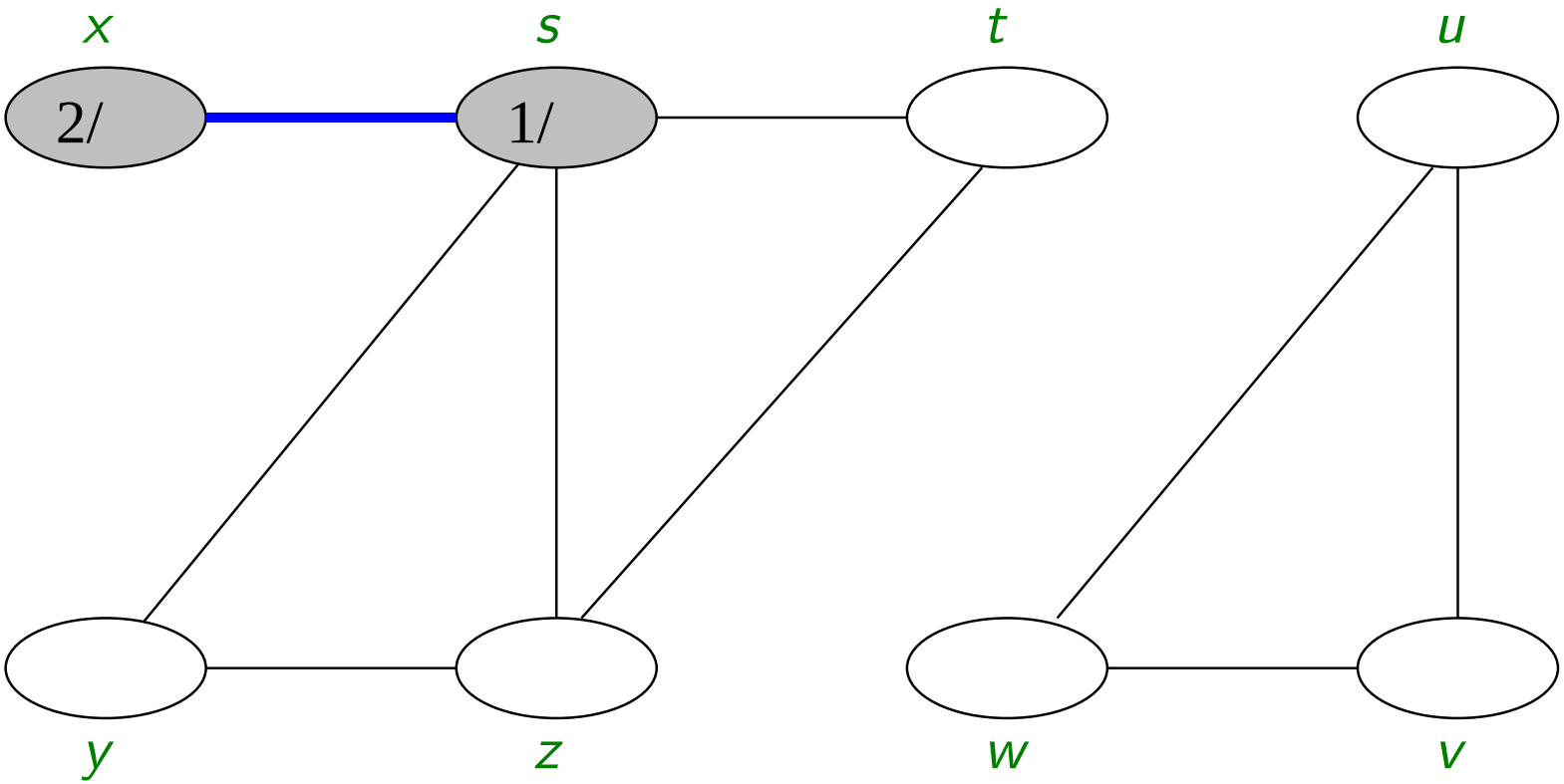


Problema: determinar os componentes conexos de um grafo.

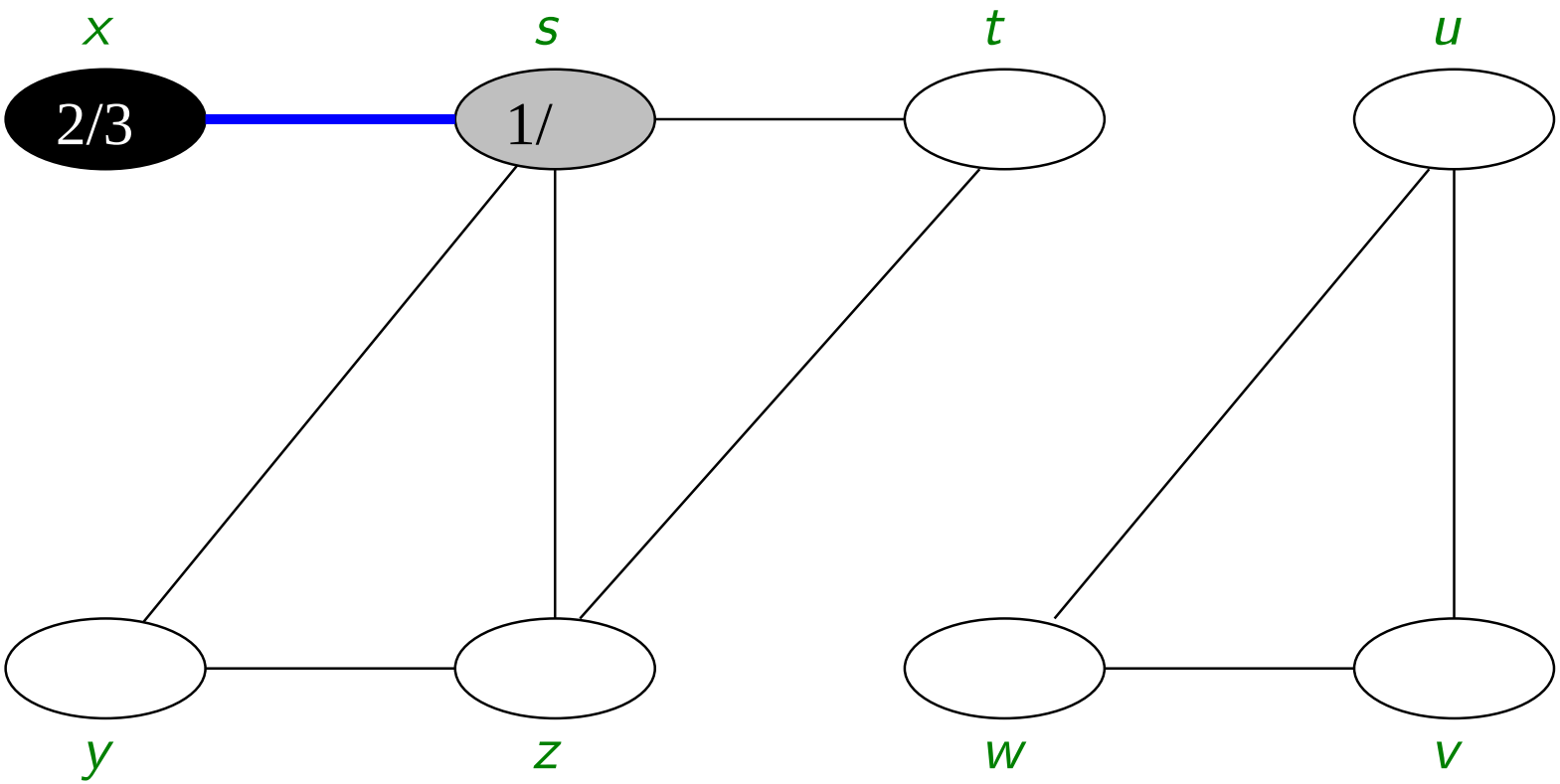
Aplicação: componentes conexos



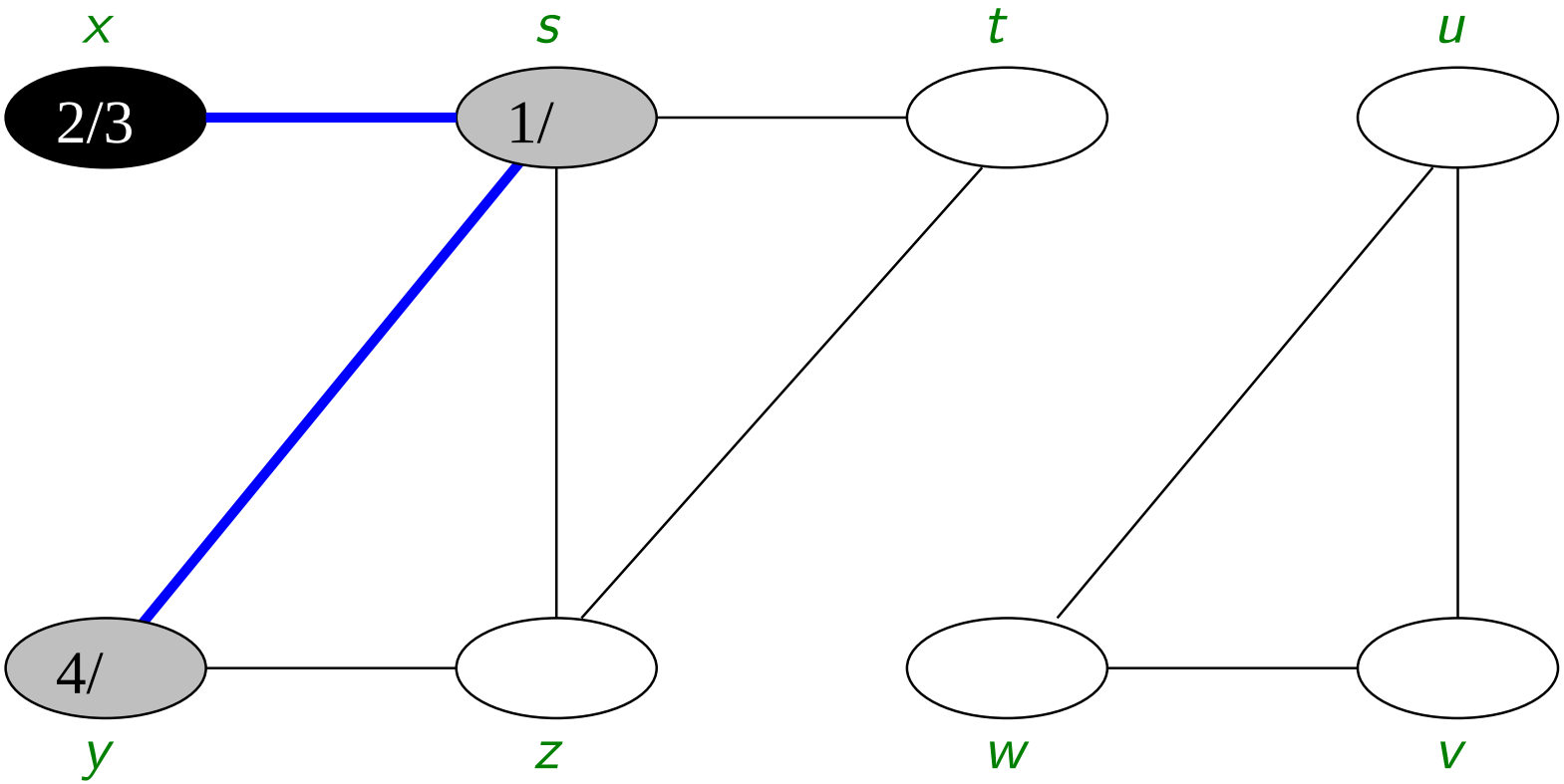
Aplicação: componentes conexos



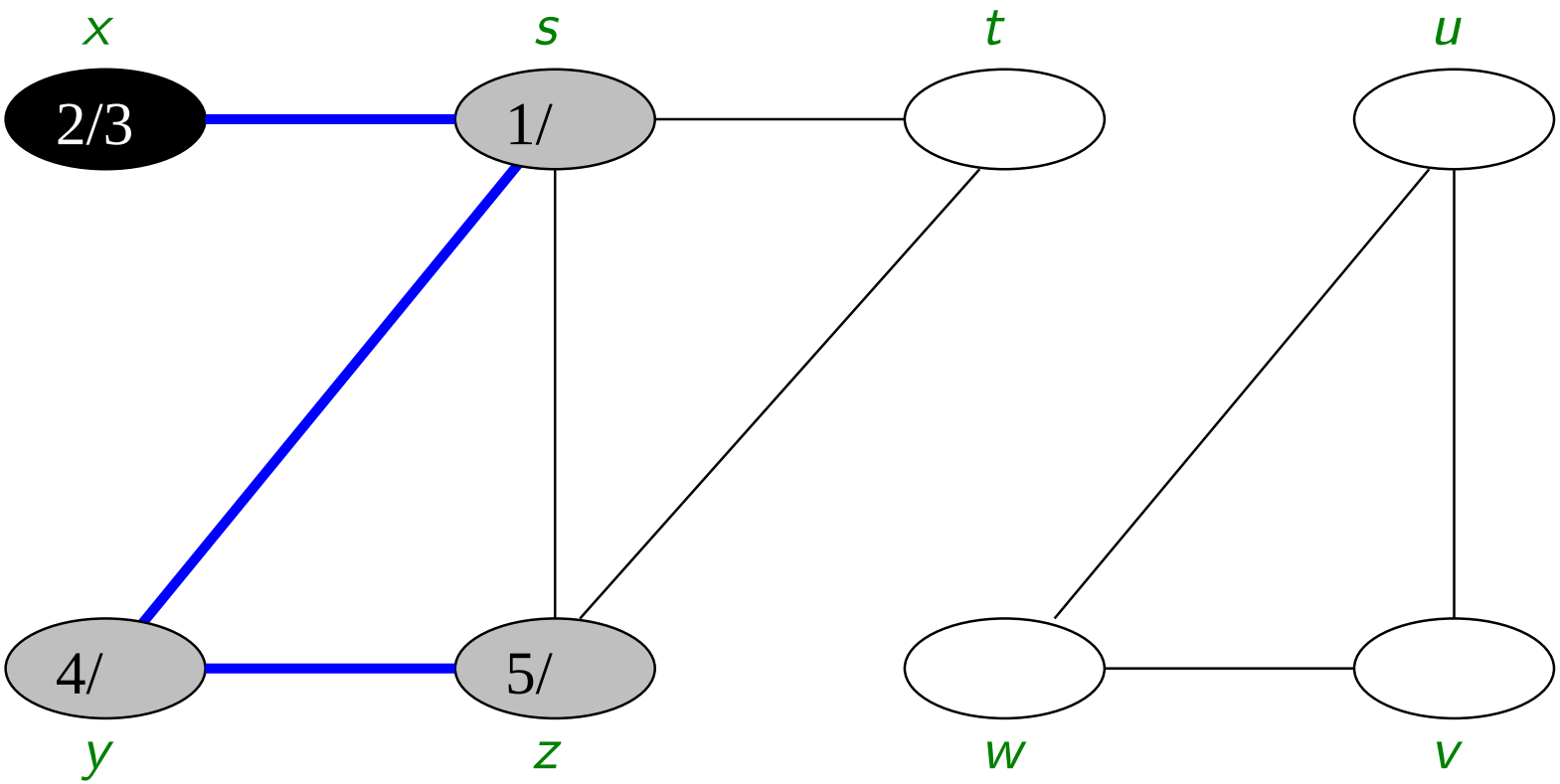
Aplicação: componentes conexos



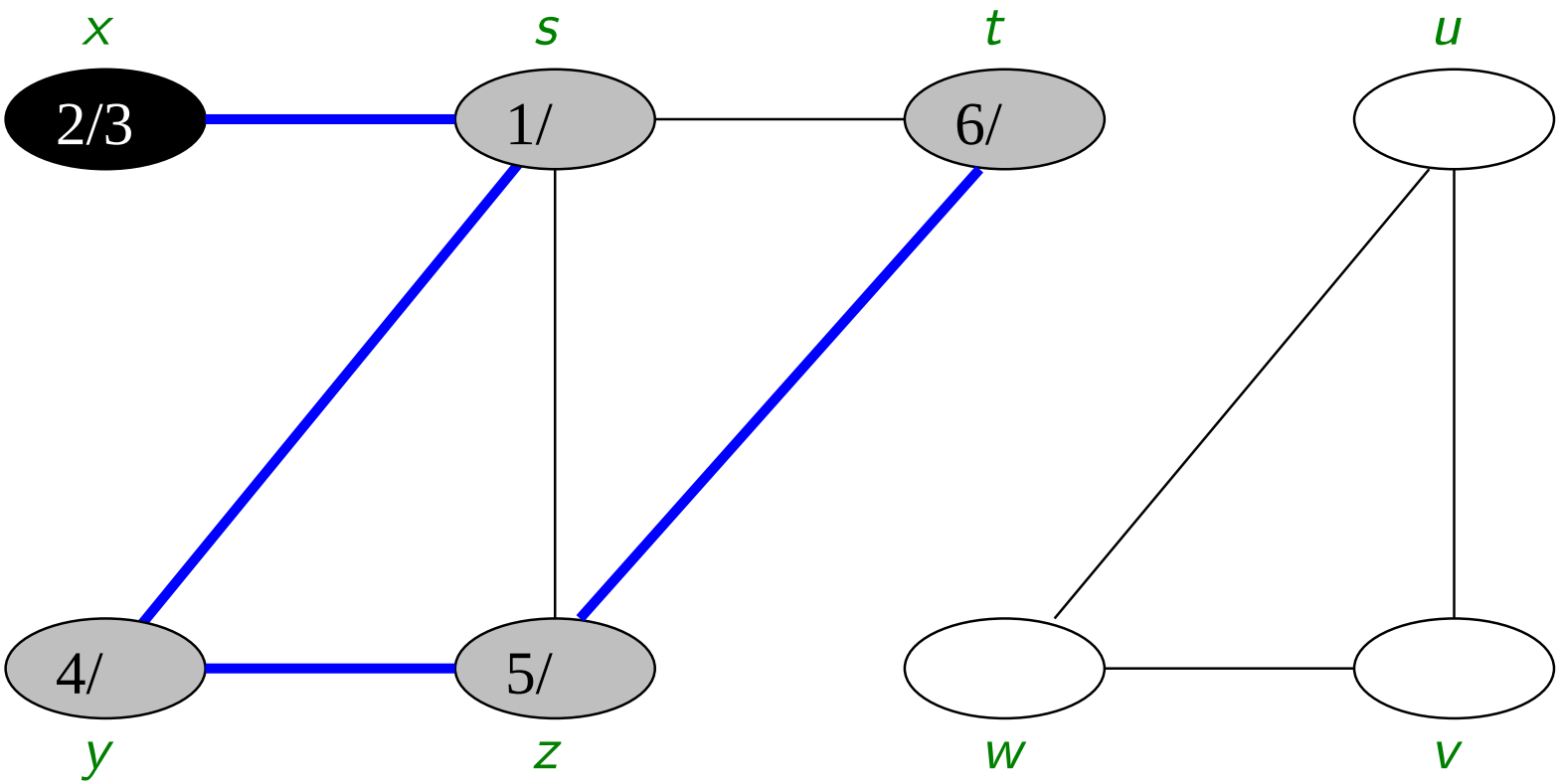
Aplicação: componentes conexos



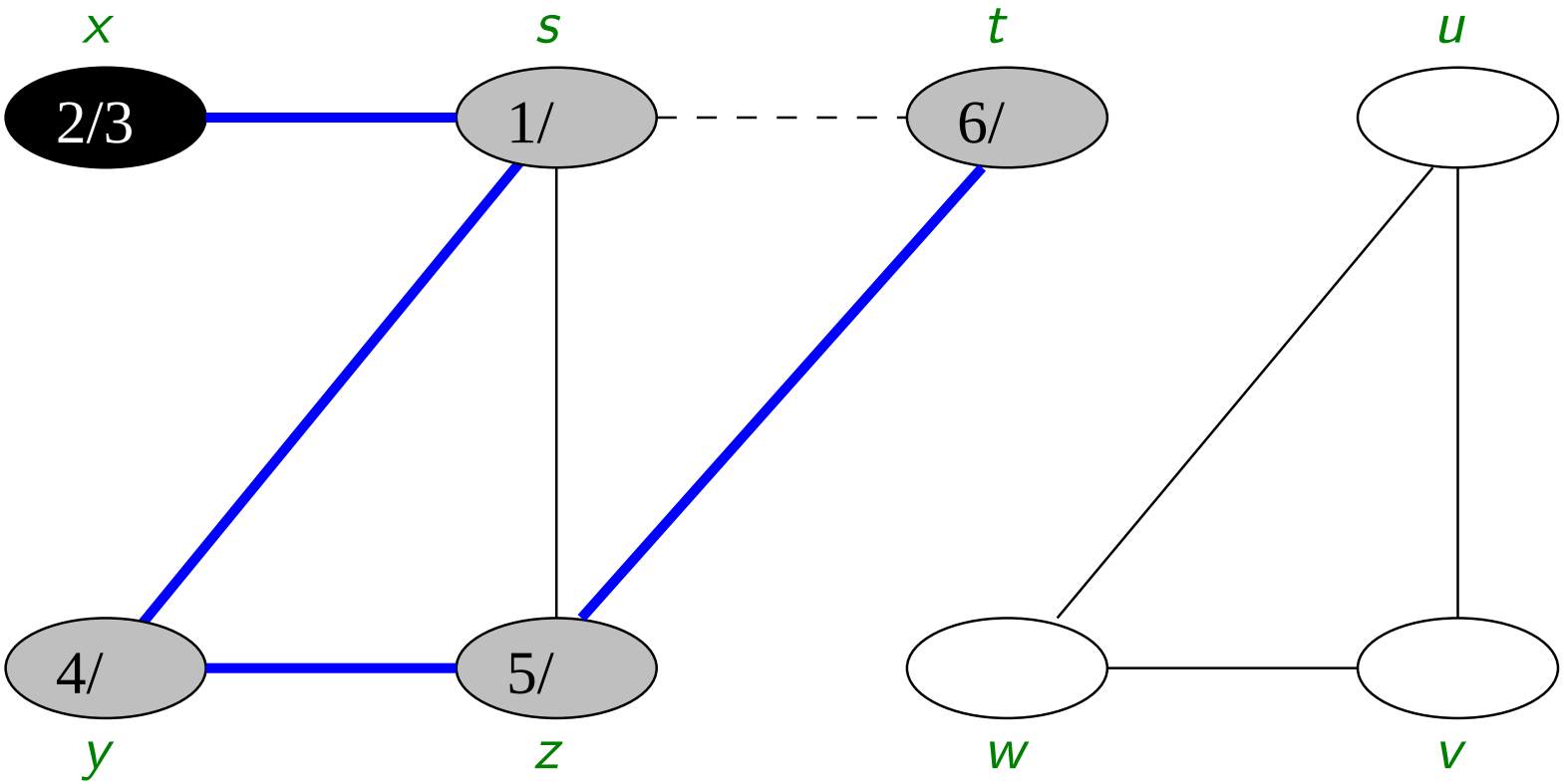
Aplicação: componentes conexos



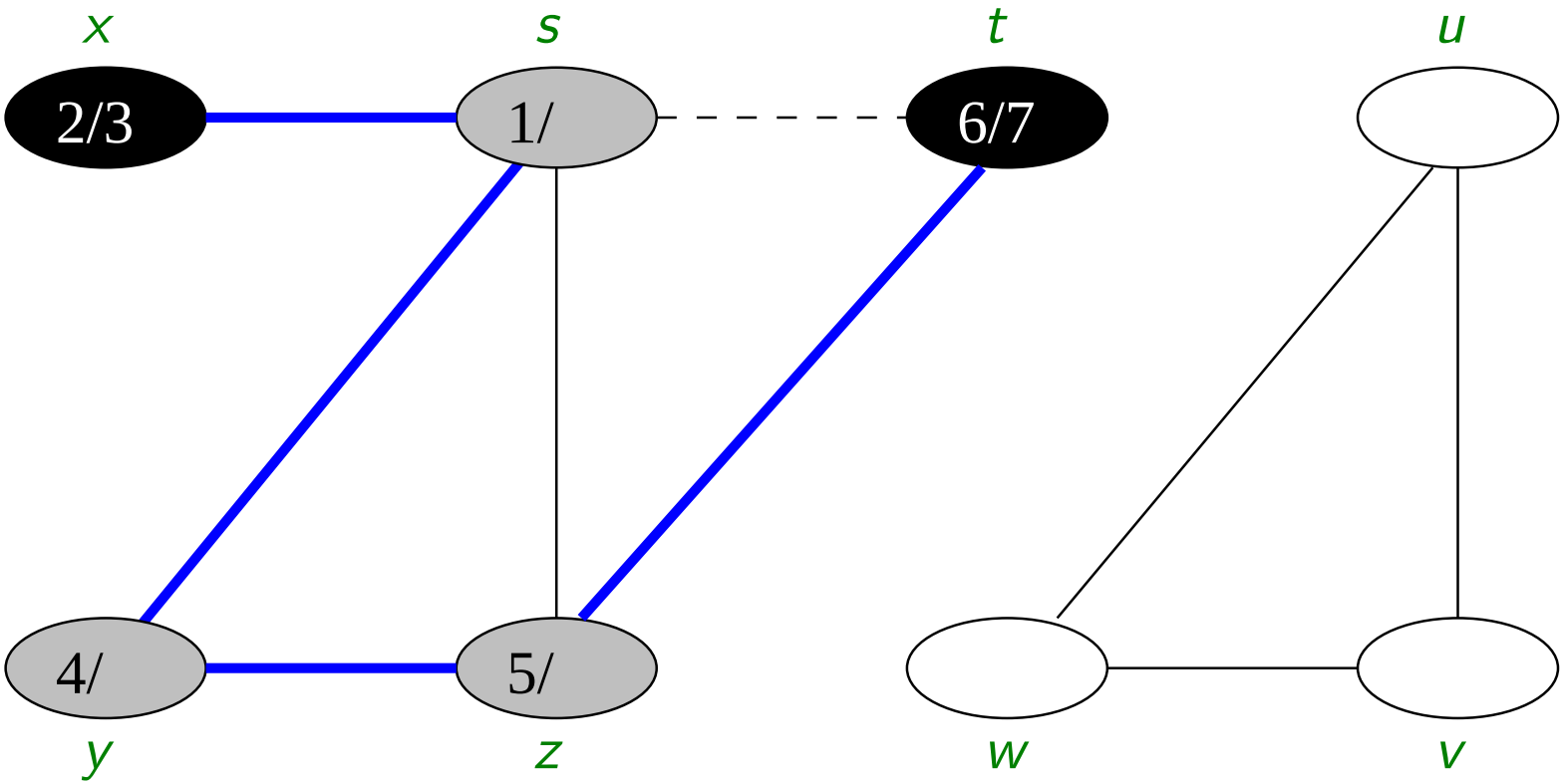
Aplicação: componentes conexos



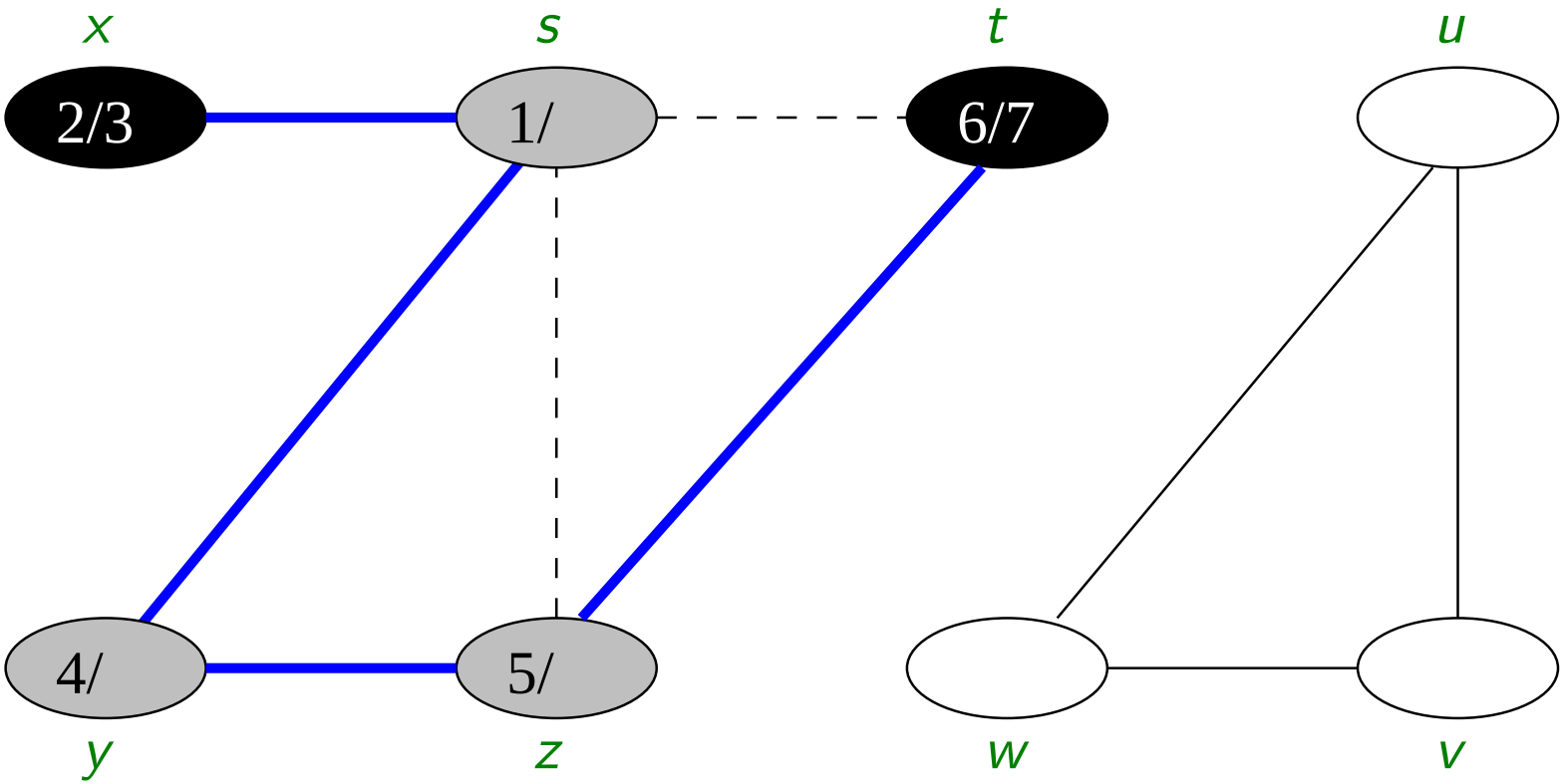
Aplicação: componentes conexos



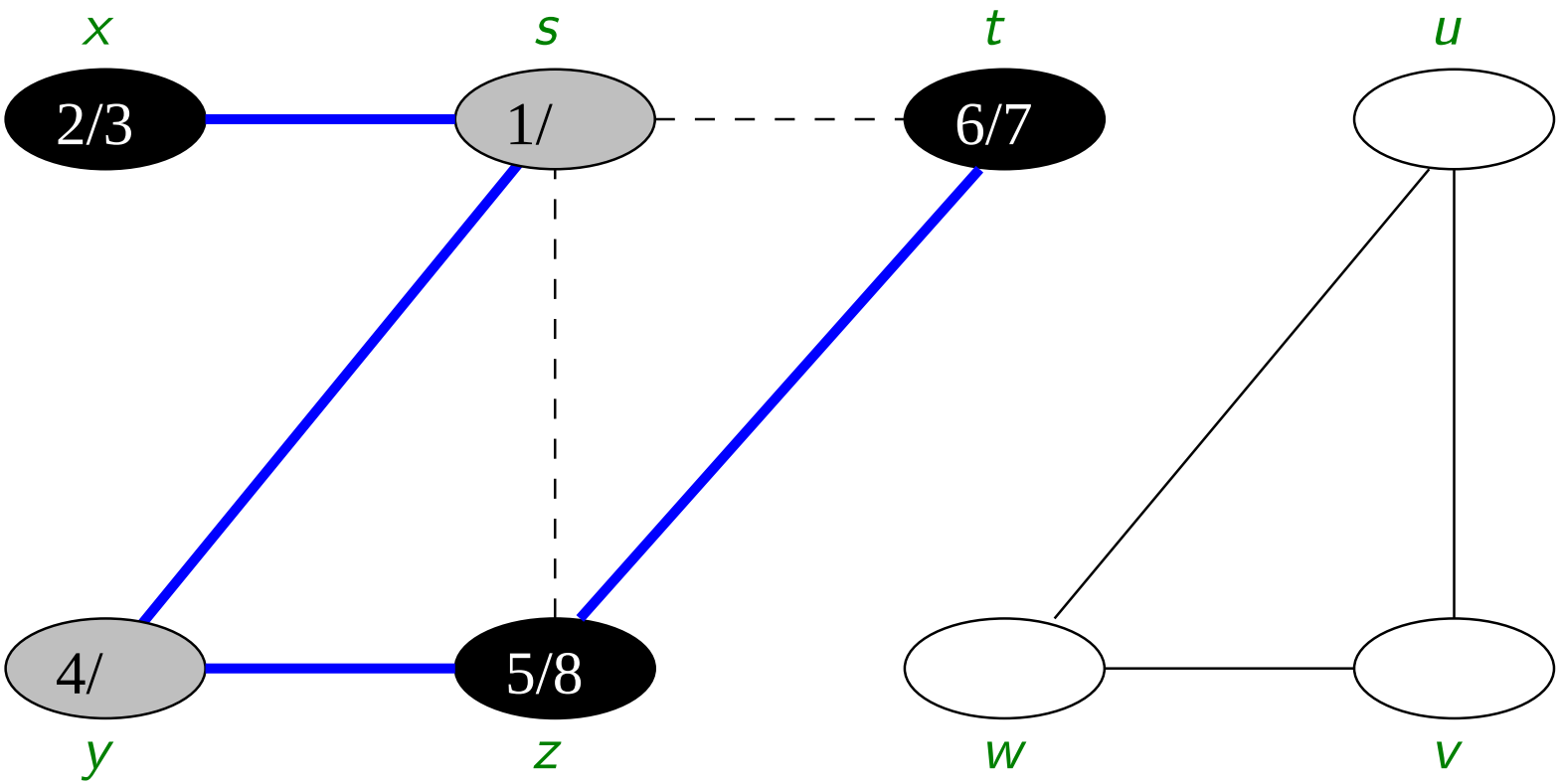
Aplicação: componentes conexos



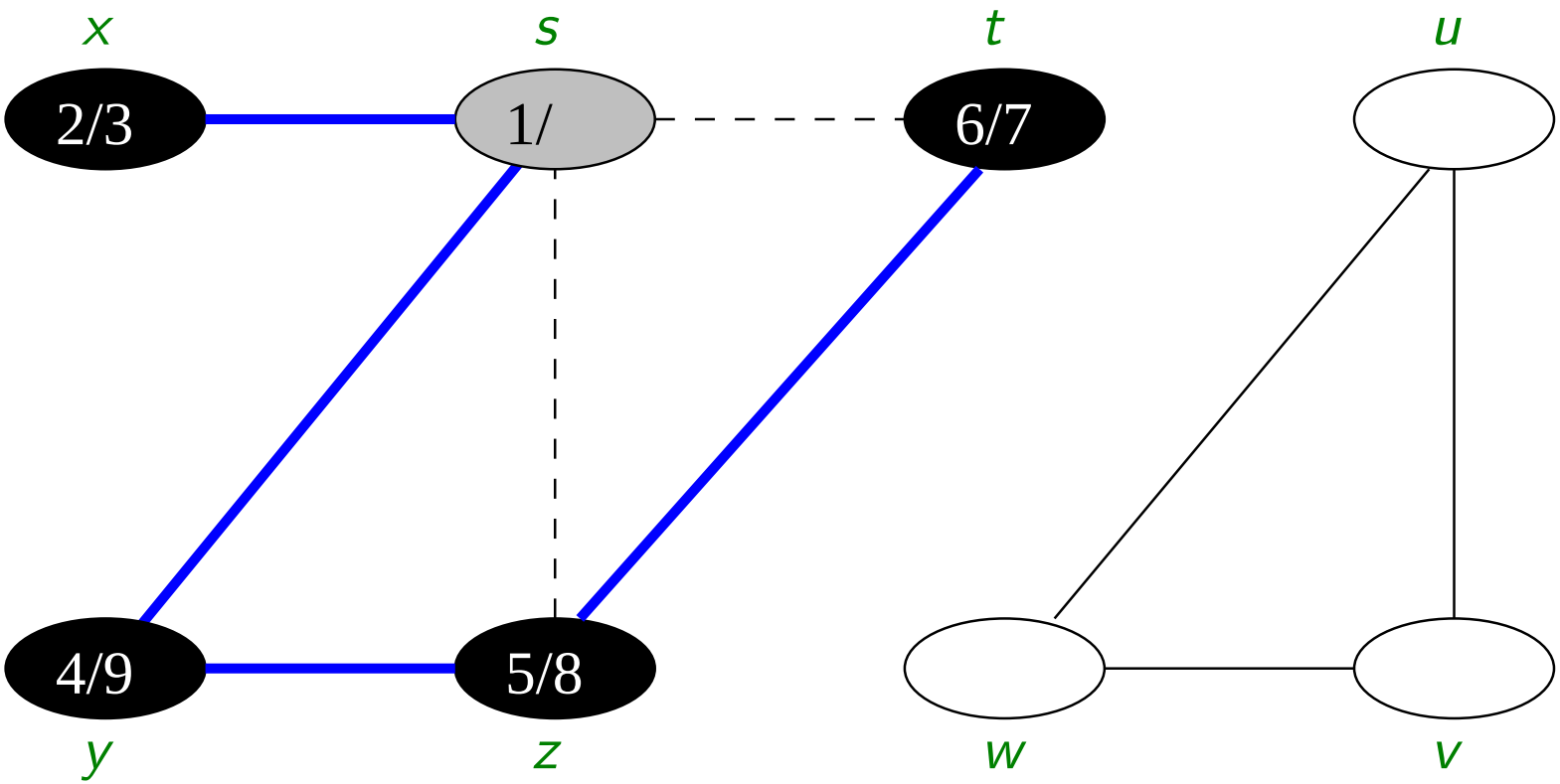
Aplicação: componentes conexos



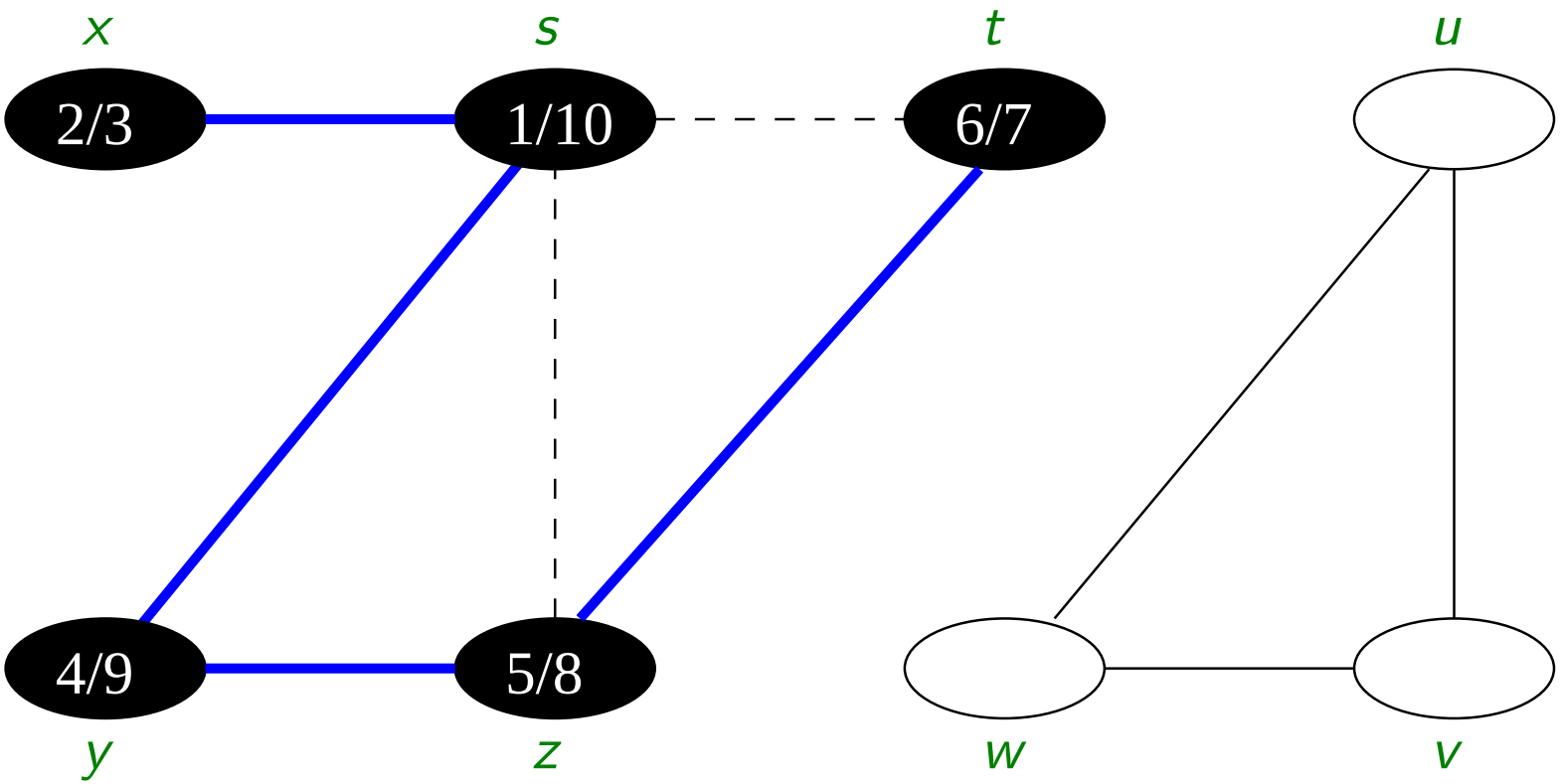
Aplicação: componentes conexos



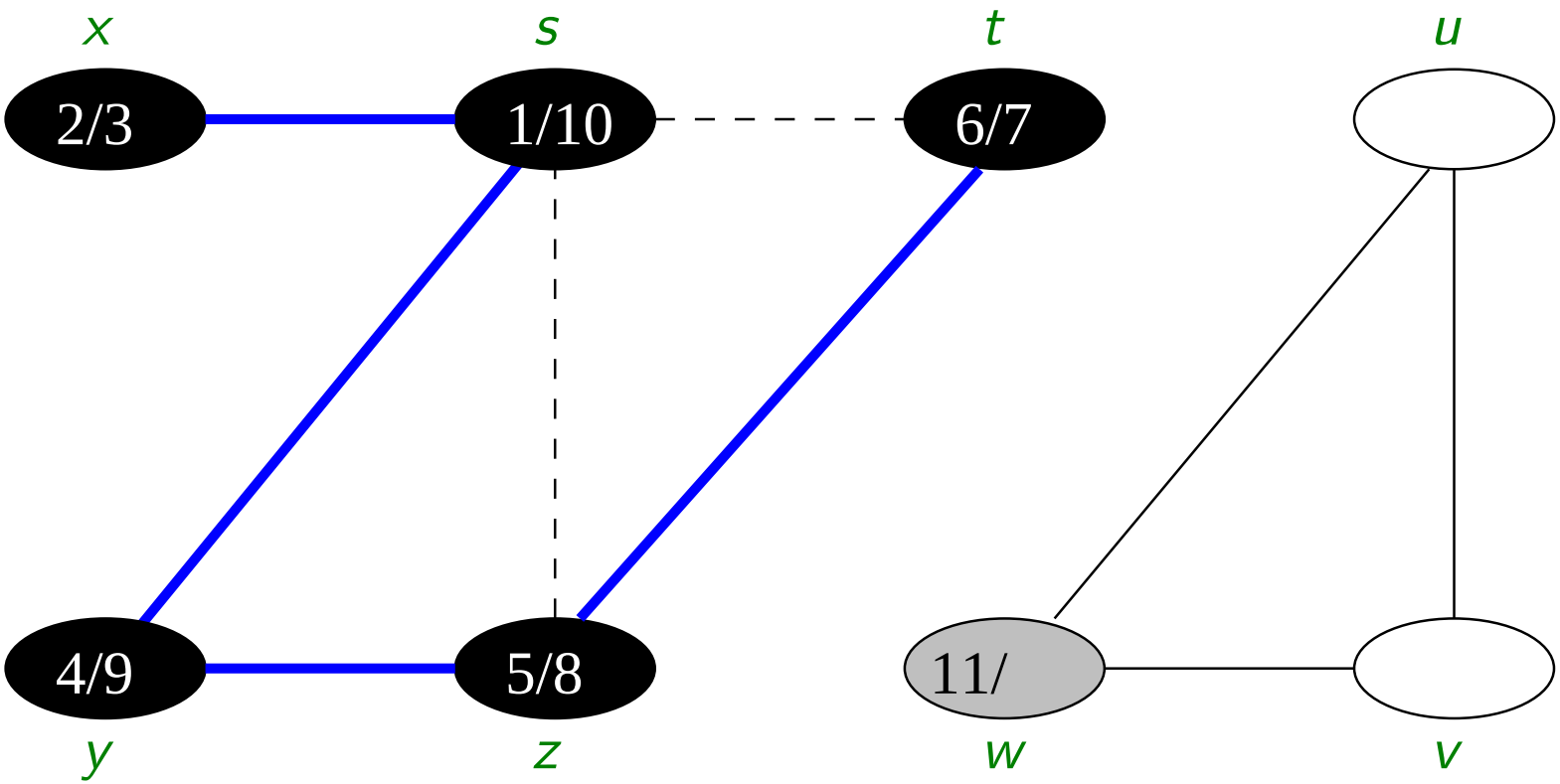
Aplicação: componentes conexos



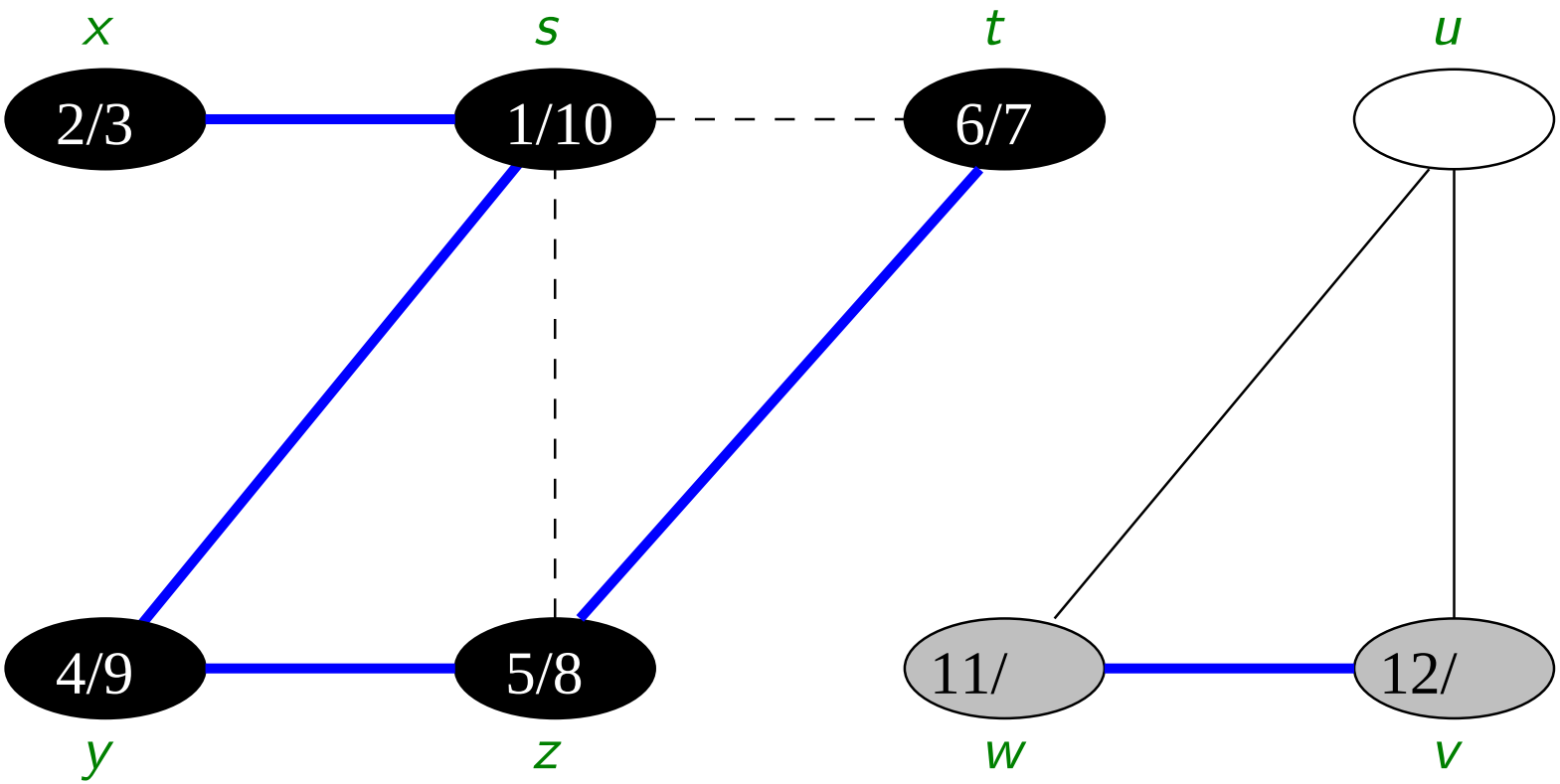
Aplicação: componentes conexos



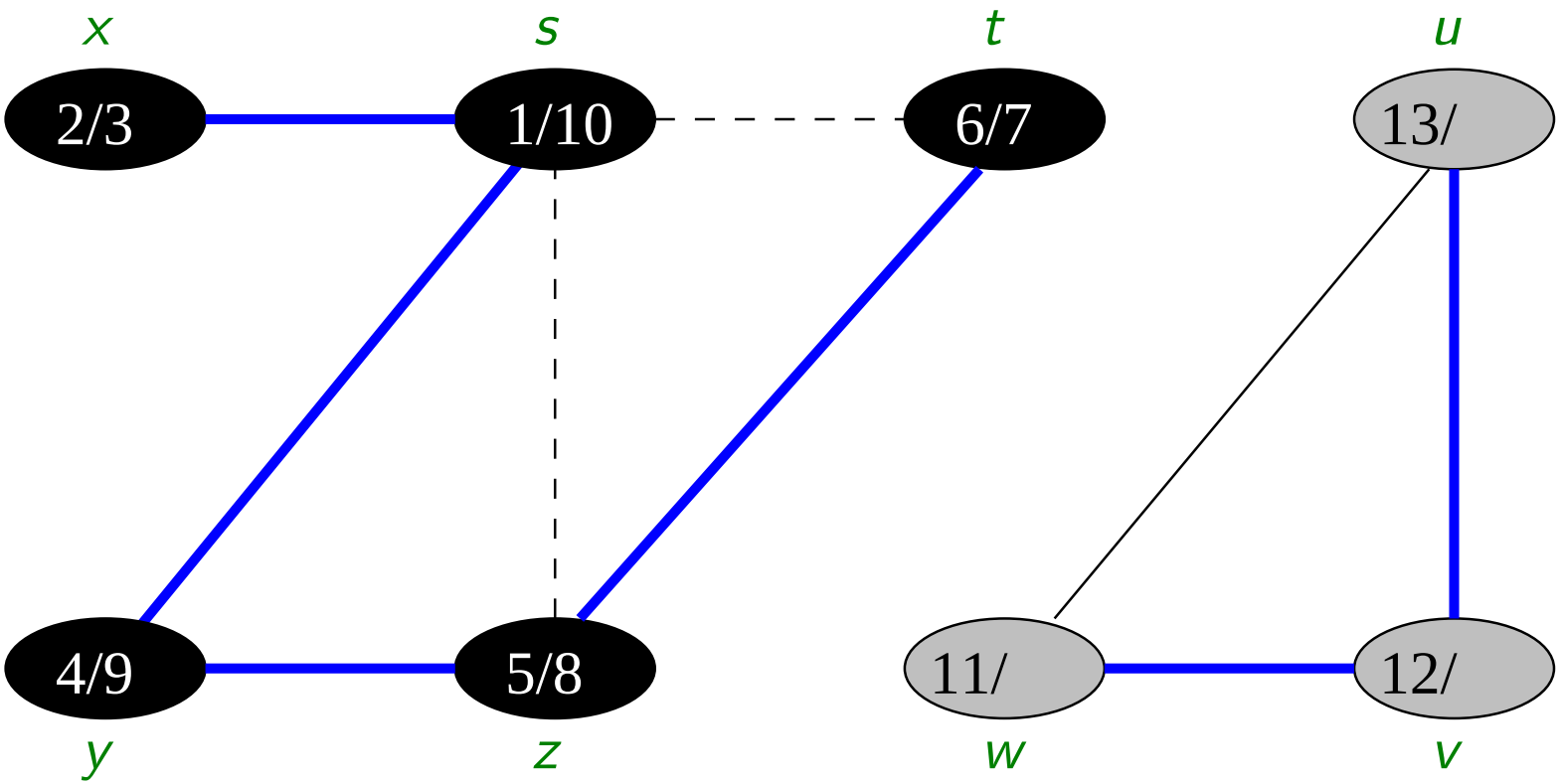
Aplicação: componentes conexos



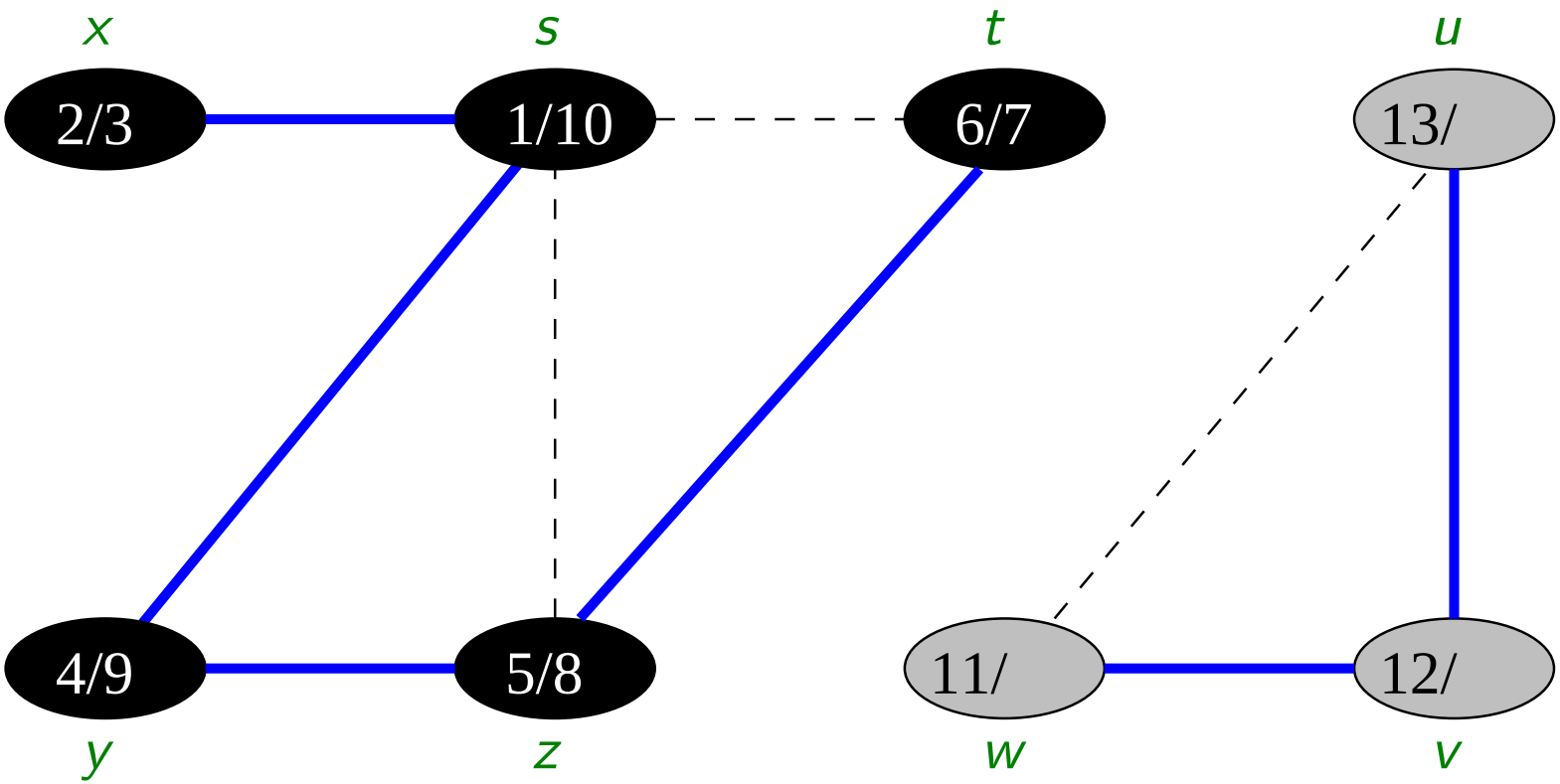
Aplicação: componentes conexos



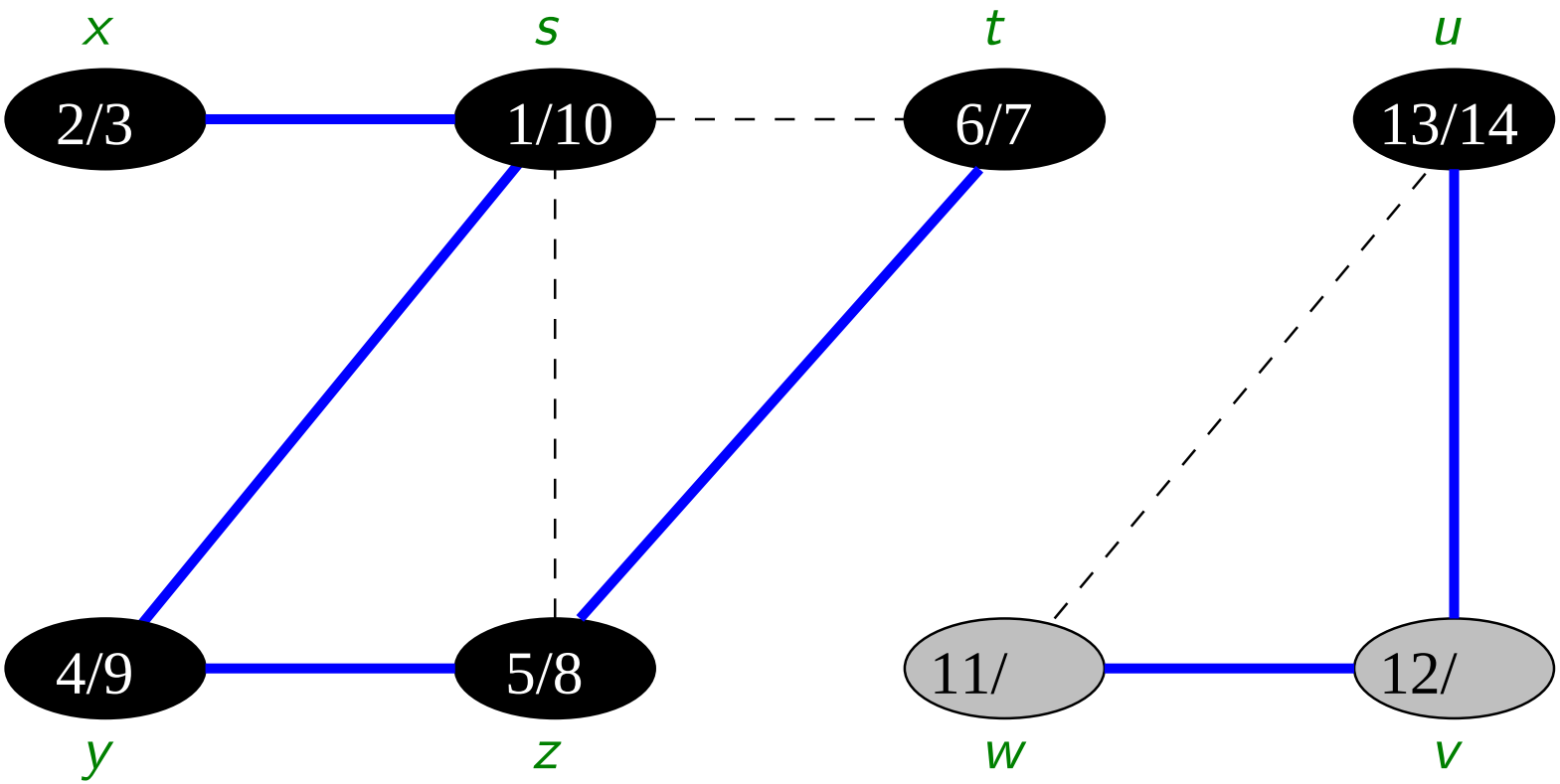
Aplicação: componentes conexos



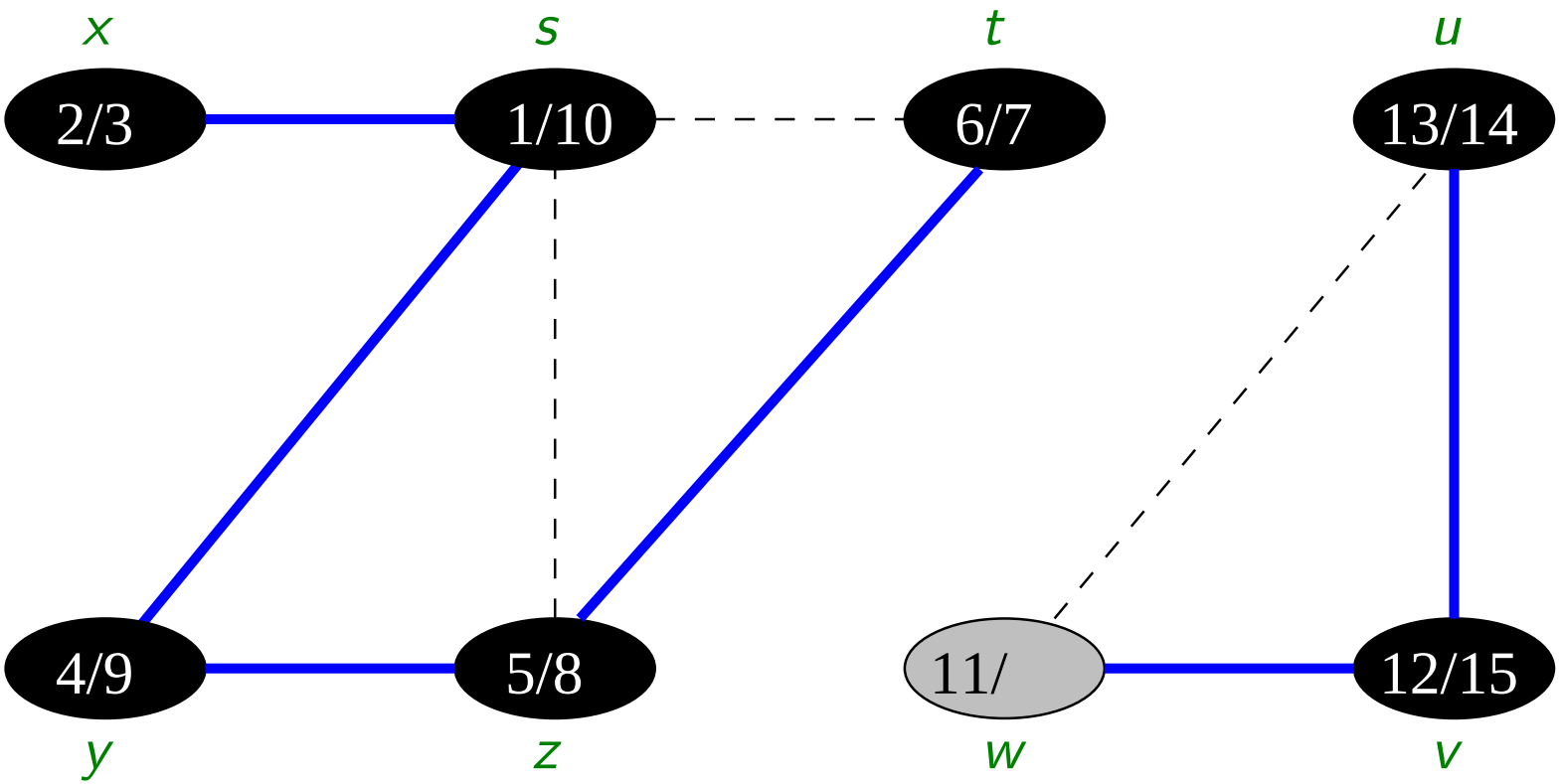
Aplicação: componentes conexos



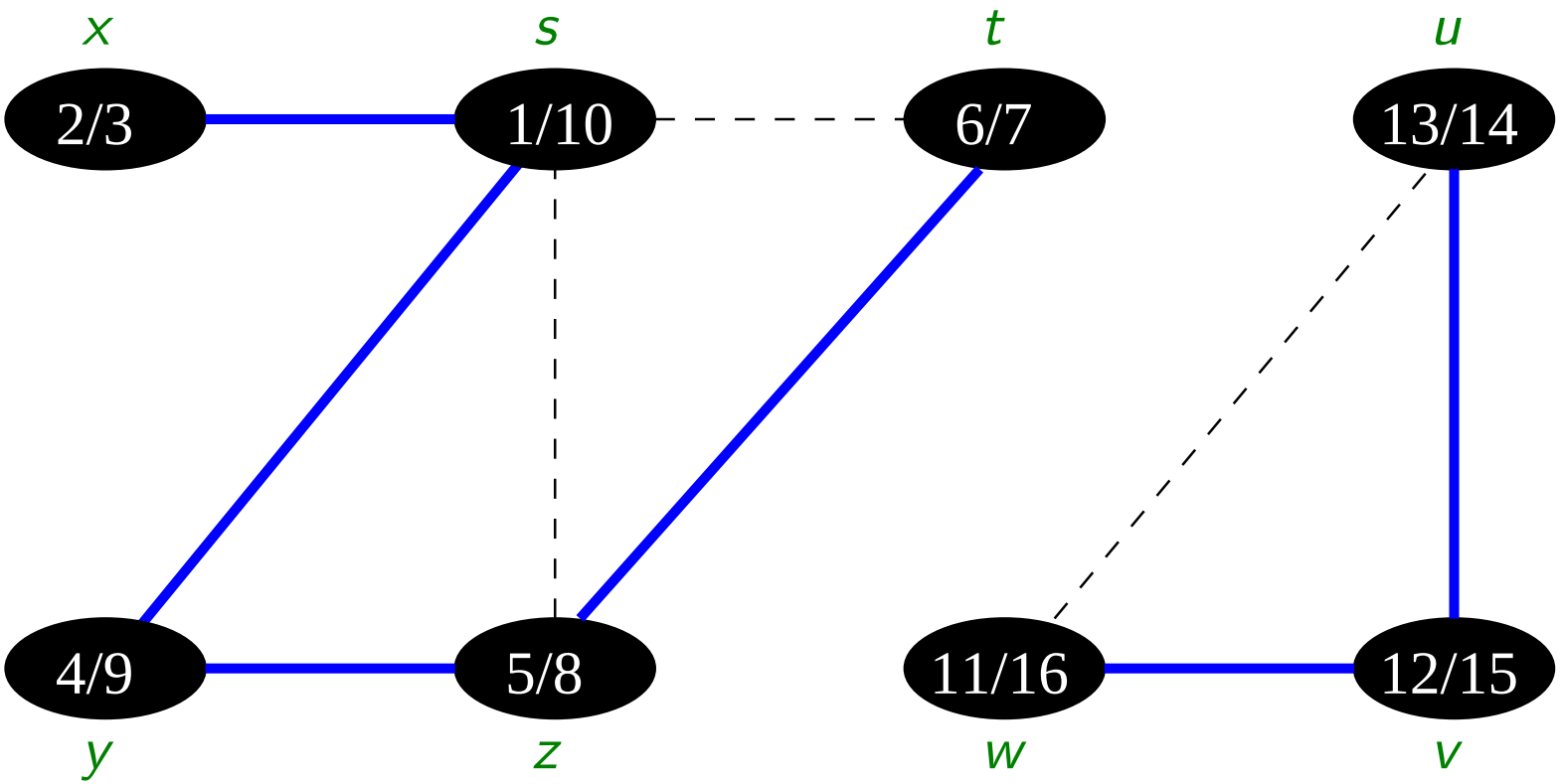
Aplicação: componentes conexos



Aplicação: componentes conexos



Aplicação: componentes conexos



Componentes conexos

O número de componentes é o **número de vezes** que DFS-VISIT(u) é chamado na linha 7 de 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   f[u] ← tempo ← tempo + 1
10   comp[u] ← ncomps;
```

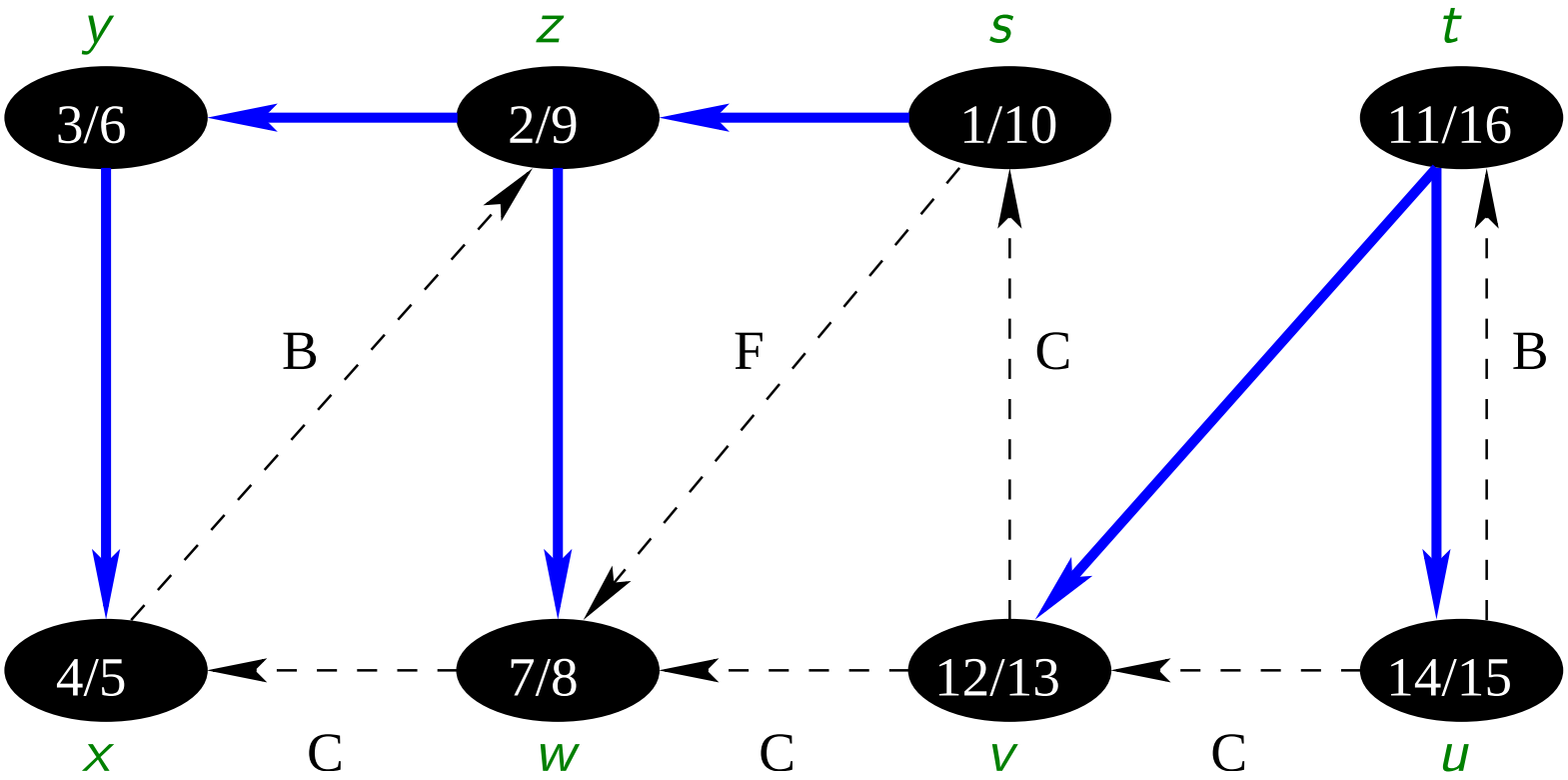
Classificação de arestas

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

Ela classifica as arestas em quatro tipos:

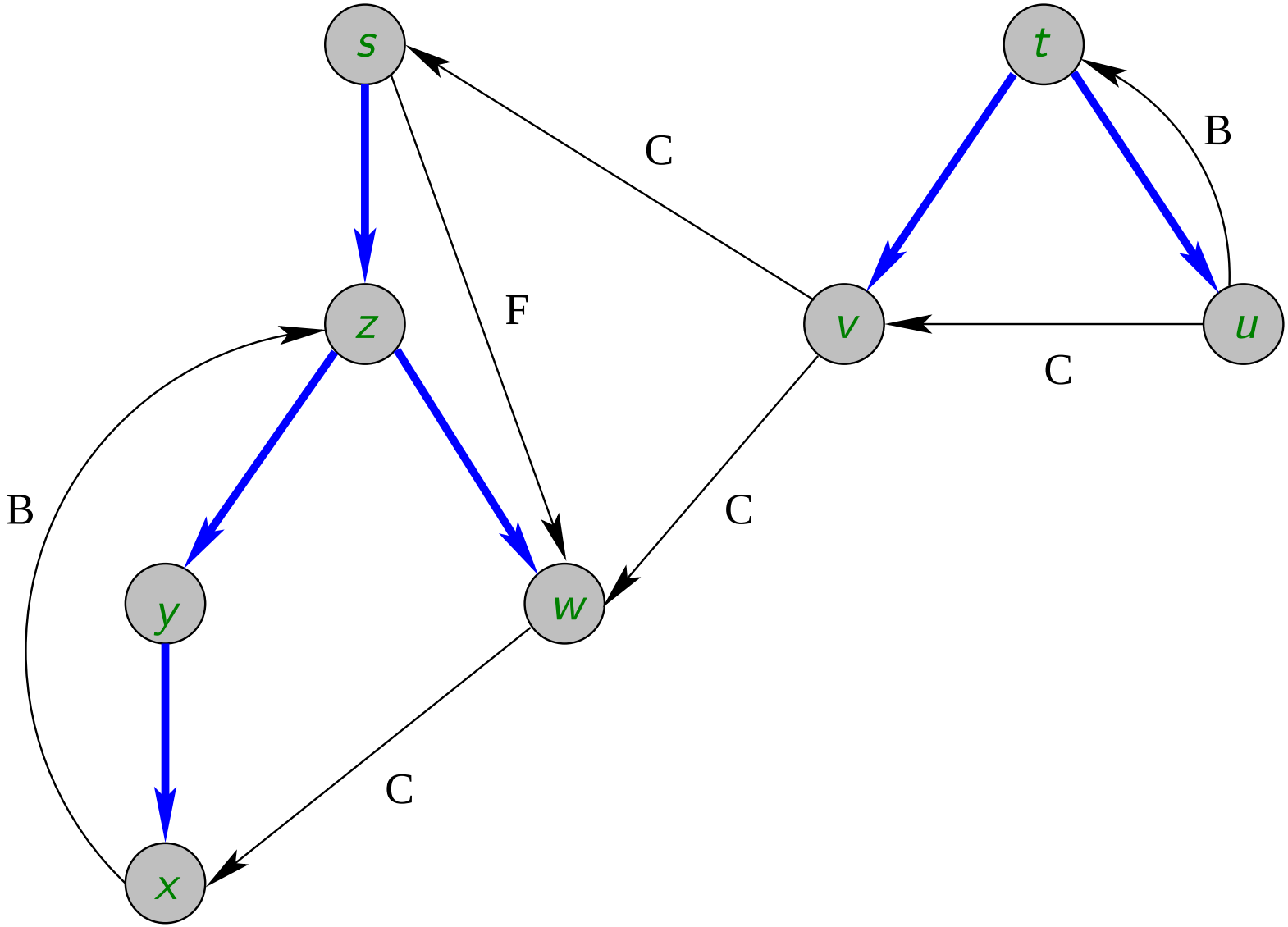
- Arestas da árvore (tree edges): arestas que pertencem à Floresta de BP.
- Arestas de retorno (backward edges): arestas (u, v) ligando um vértice u a um ancestral v na Árvore de BP.
- Arestas de avanço (forward edges): arestas (u, v) ligando um vértice u a um descendente próprio v na Árvore de BP.
-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

Classificação de arestas



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

Classificação de arestas



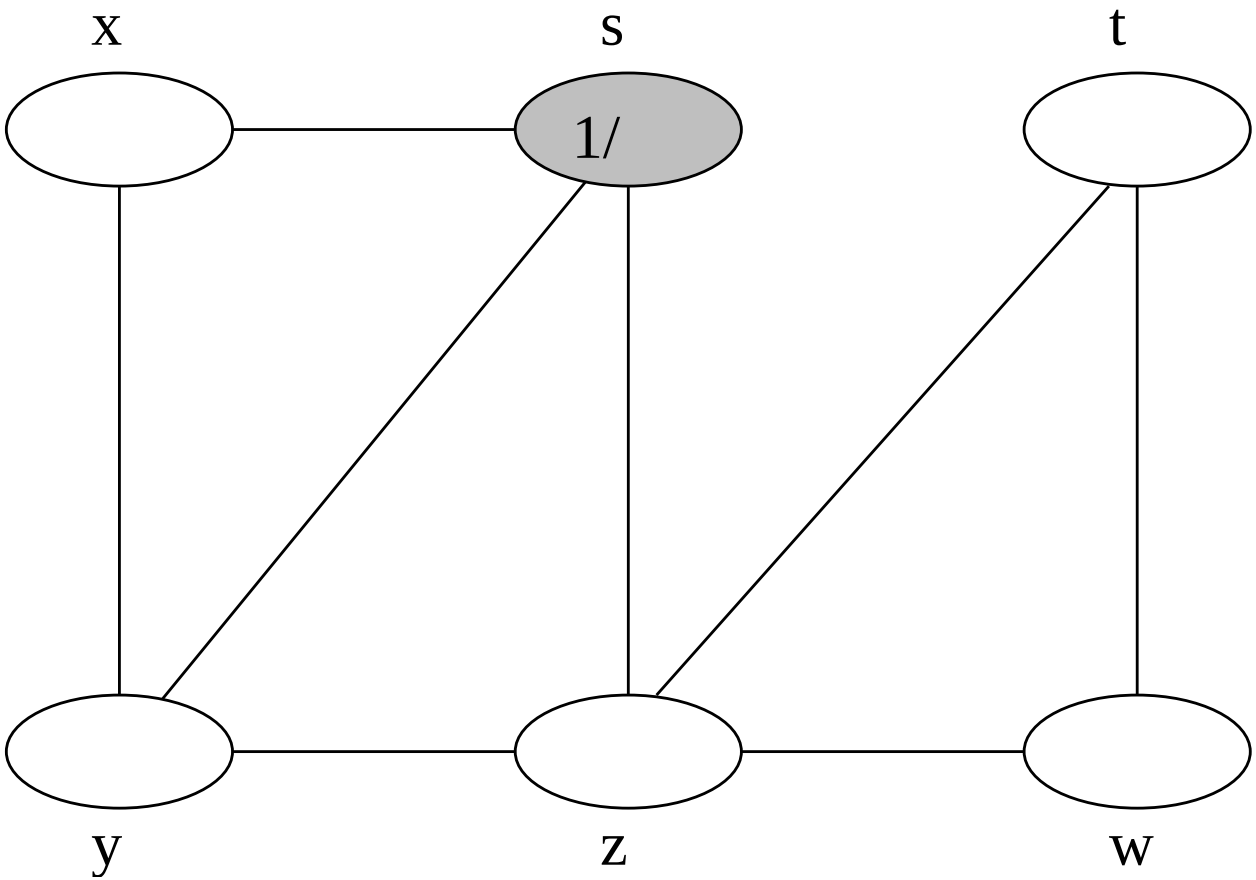
Grafos não-orientados

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

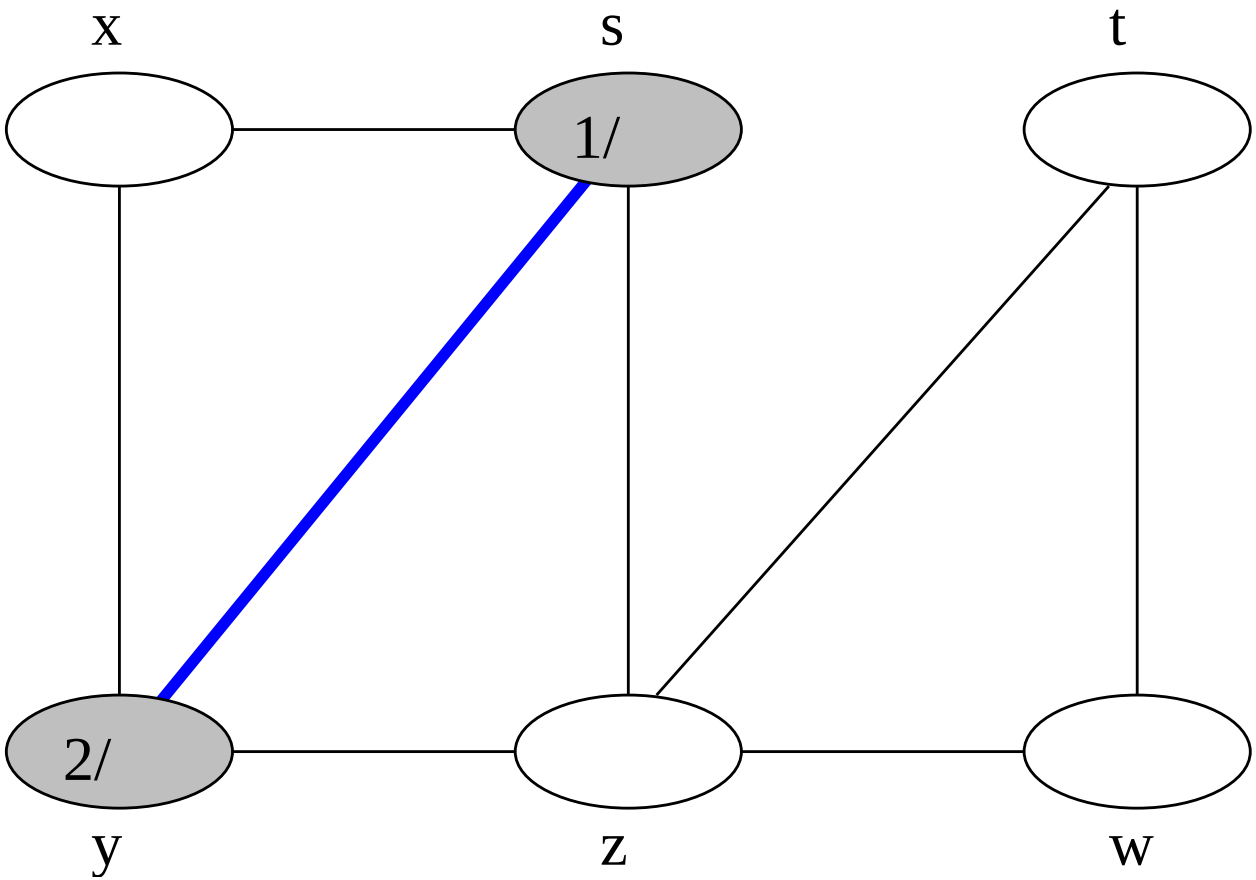
Para grafos não-orientados, existem apenas dois tipos de arestas.

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

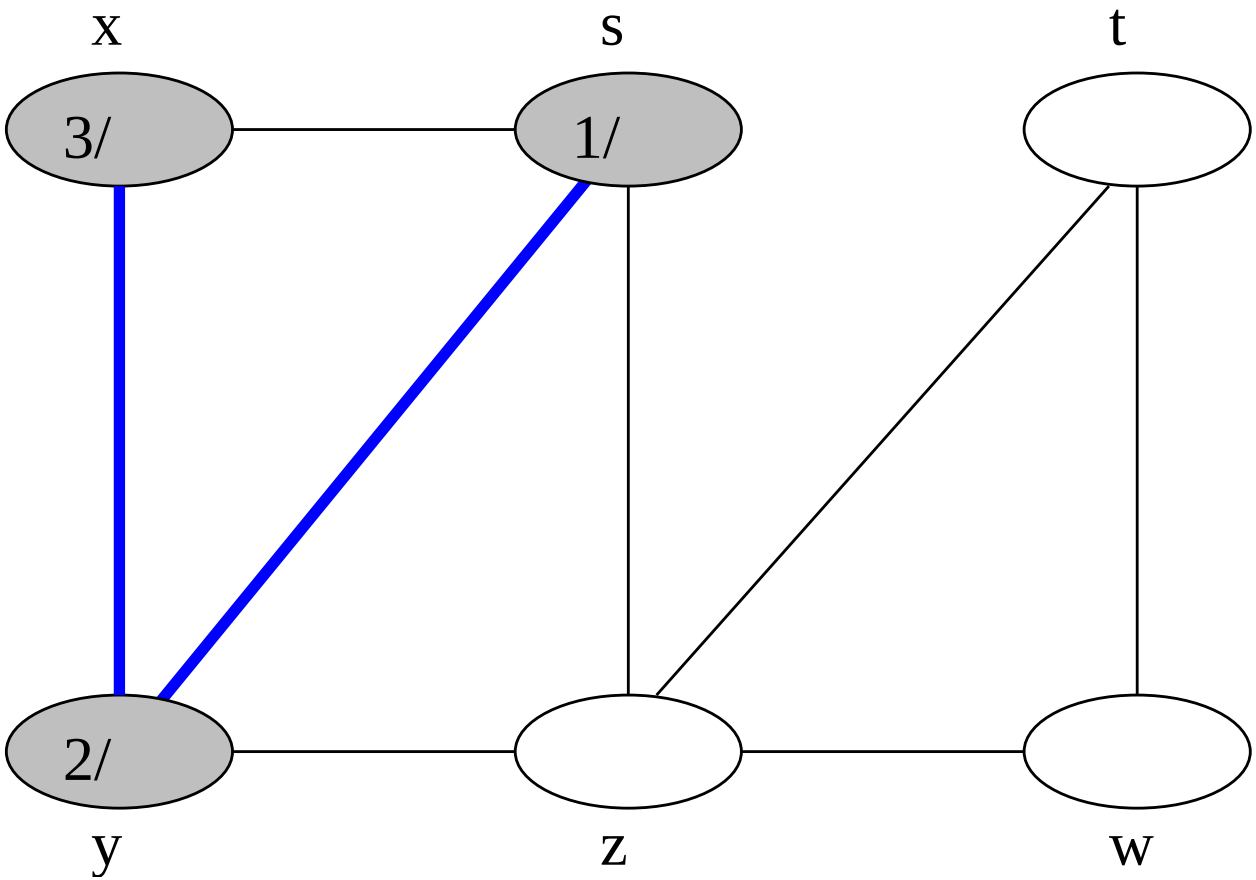
Exemplo



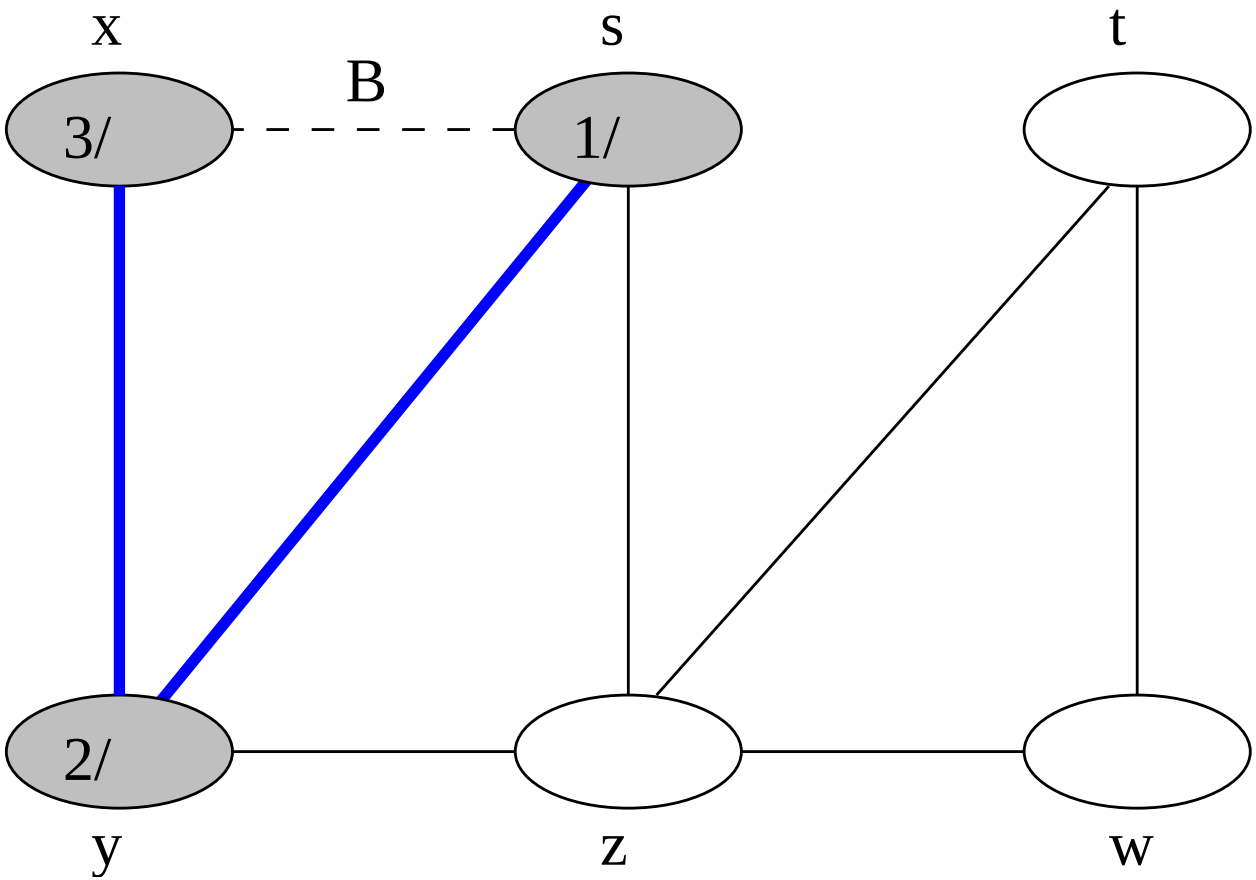
Exemplo



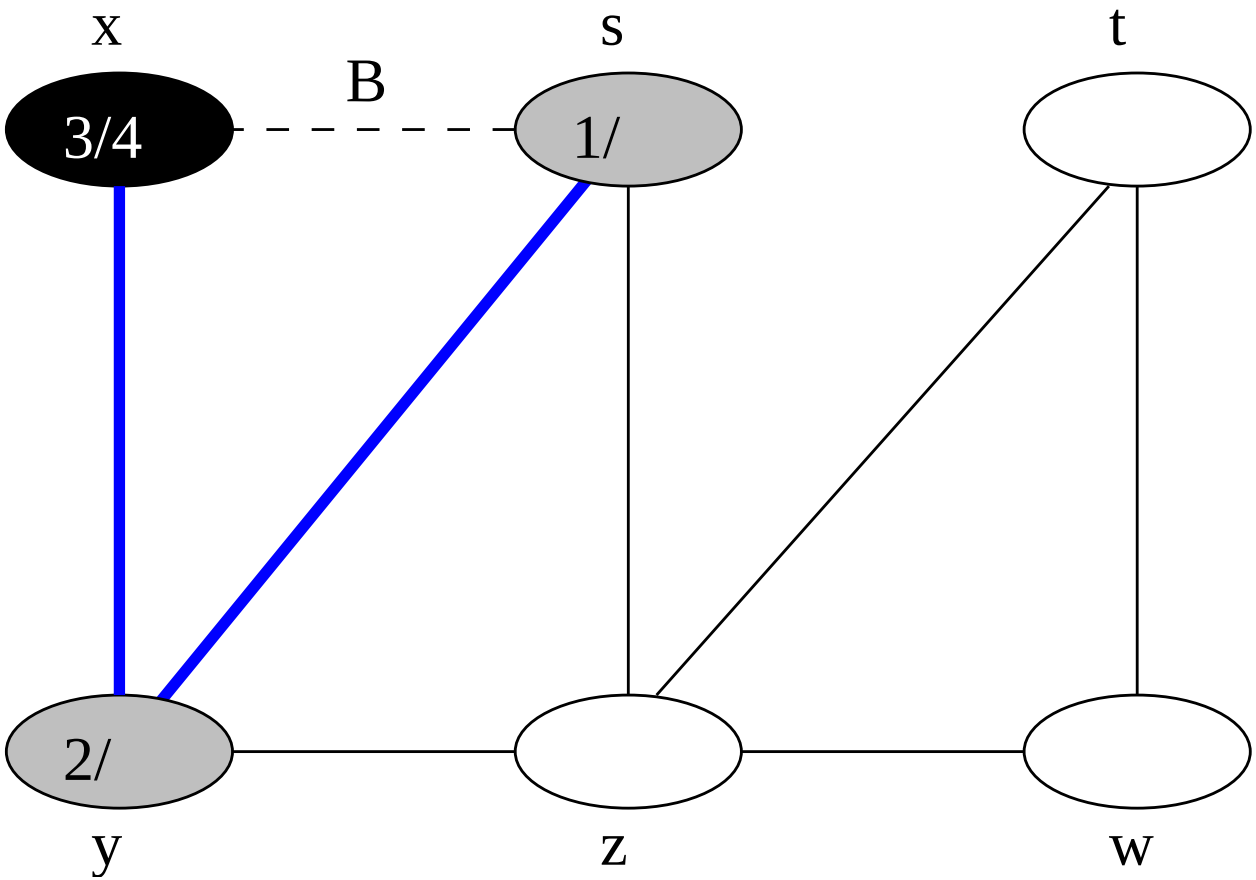
Exemplo



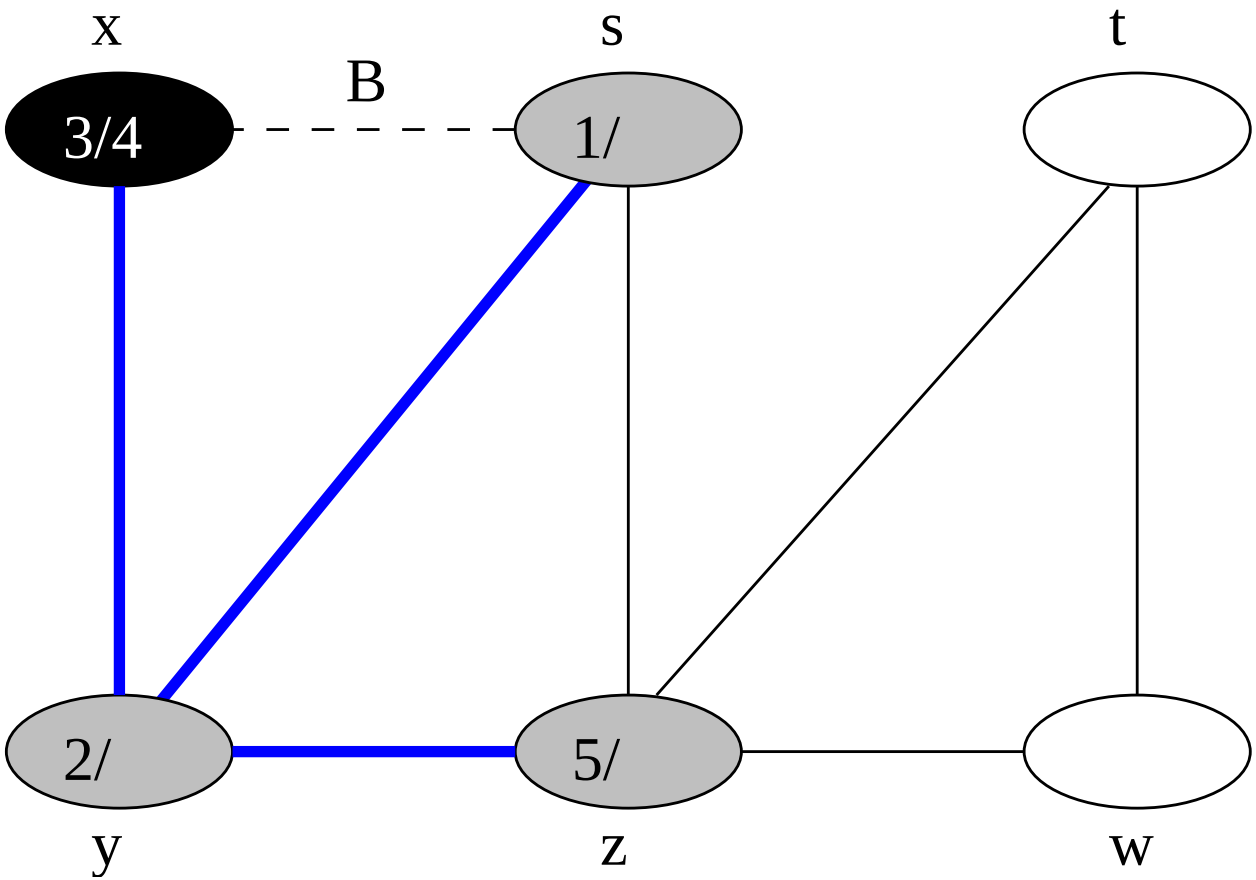
Exemplo



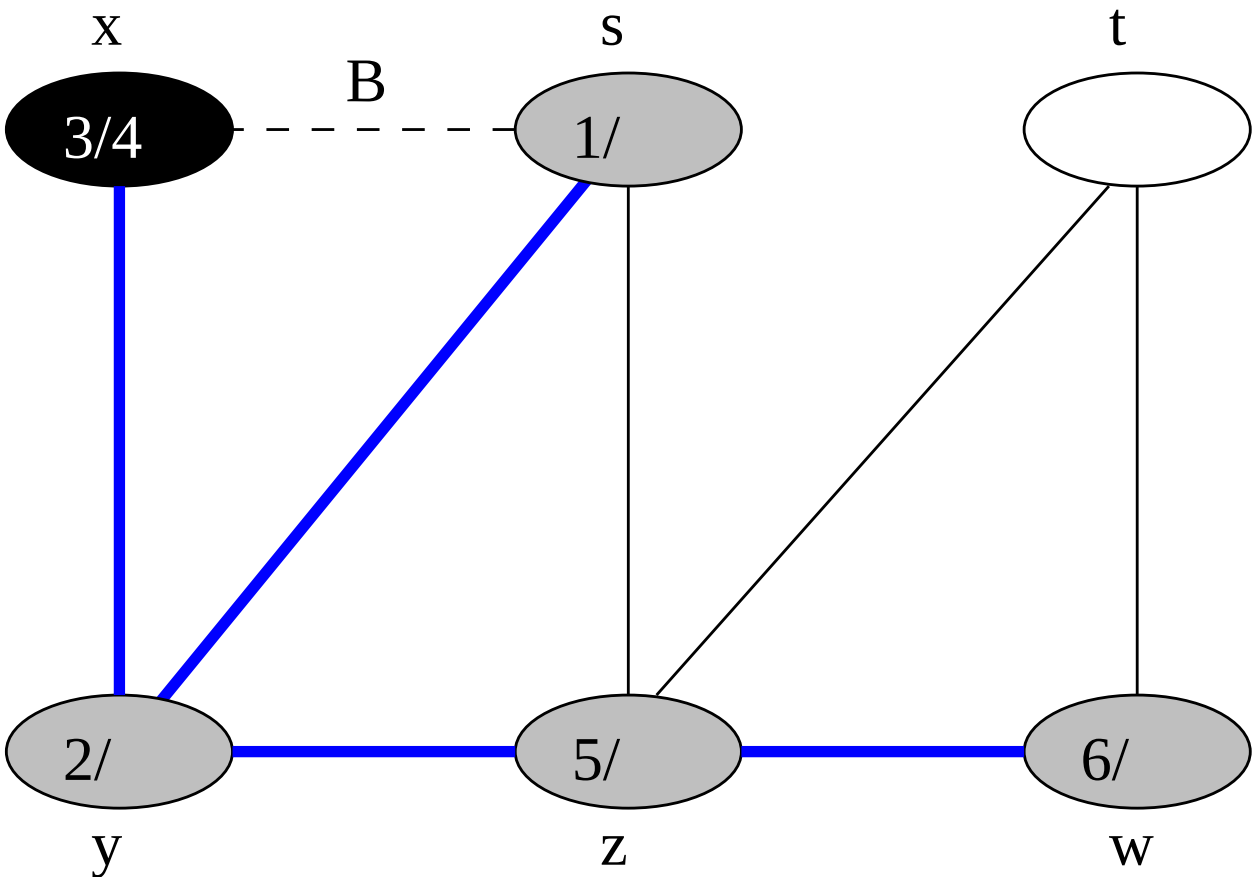
Exemplo



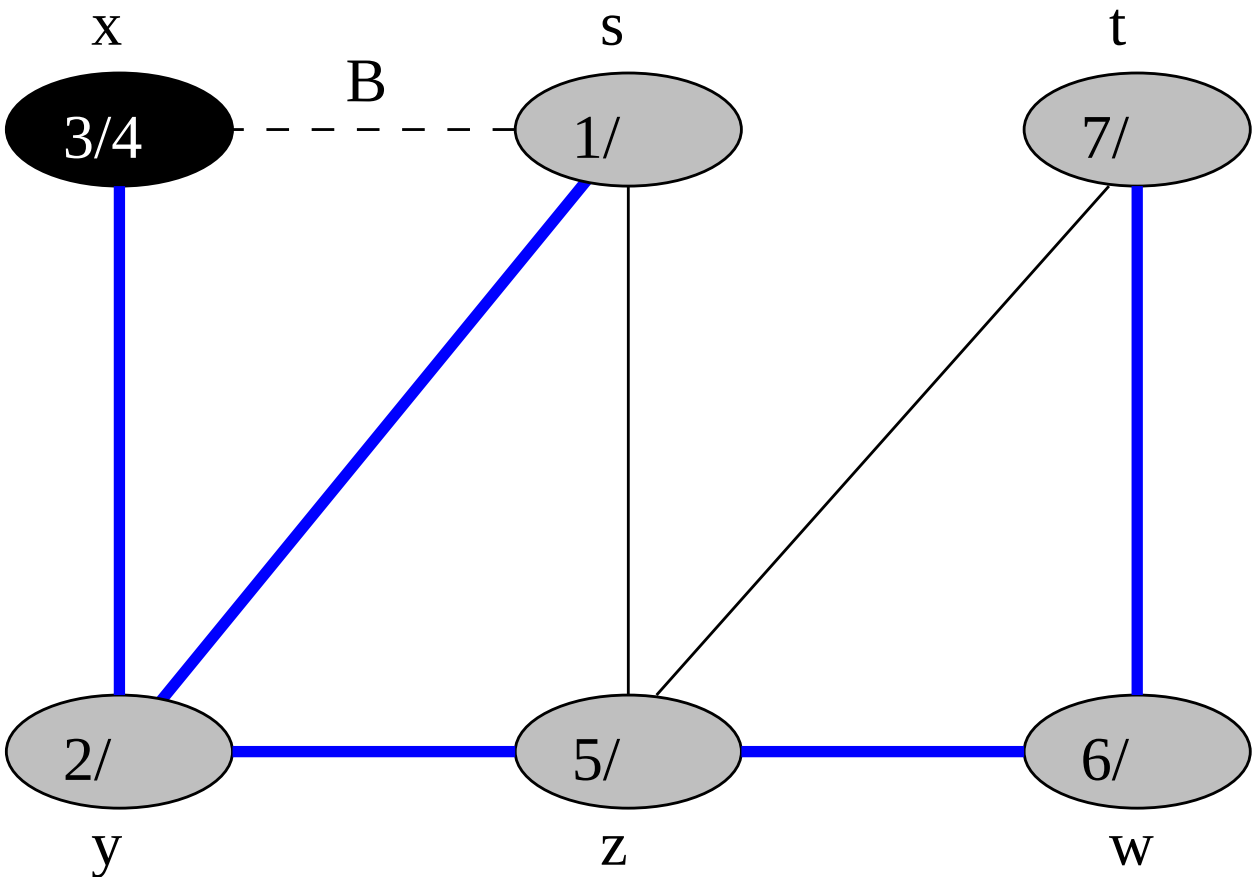
Exemplo



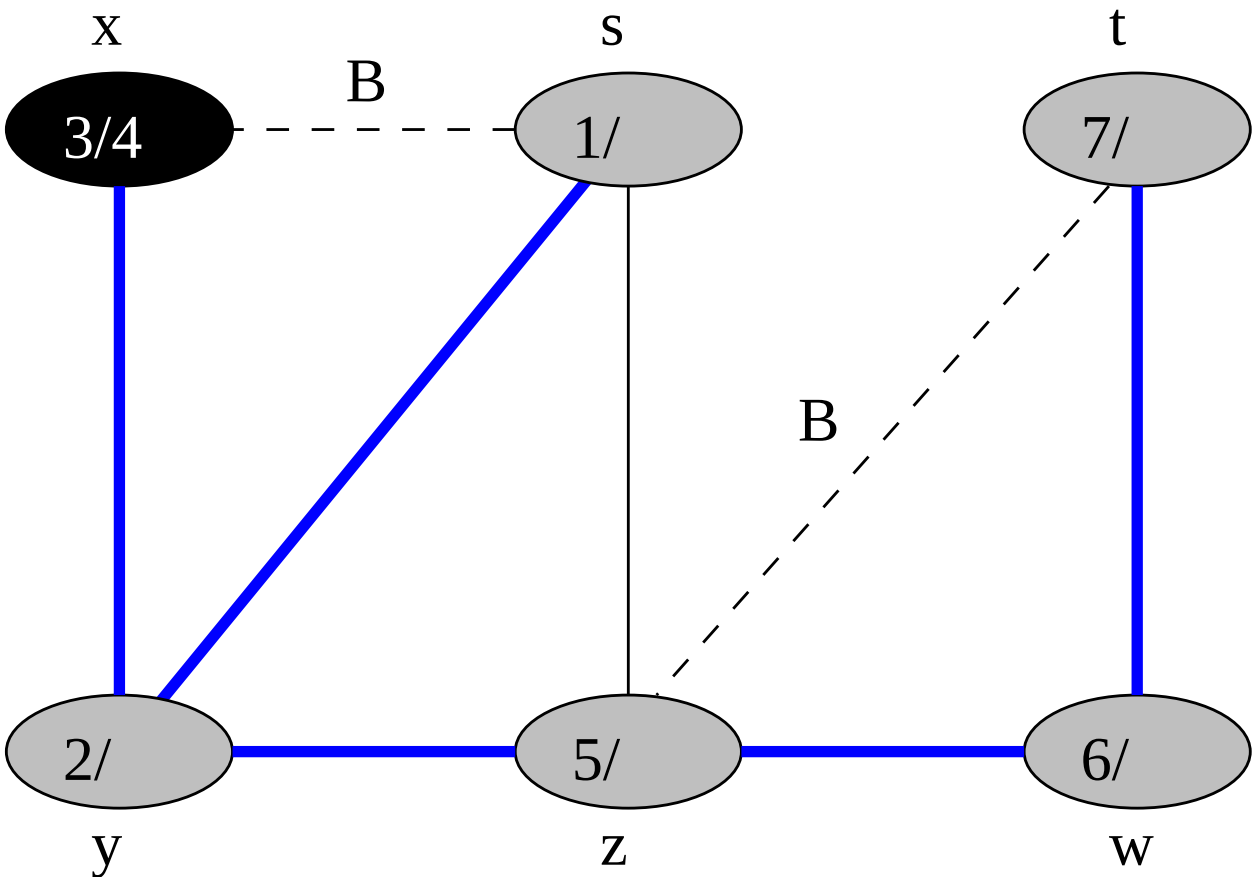
Exemplo



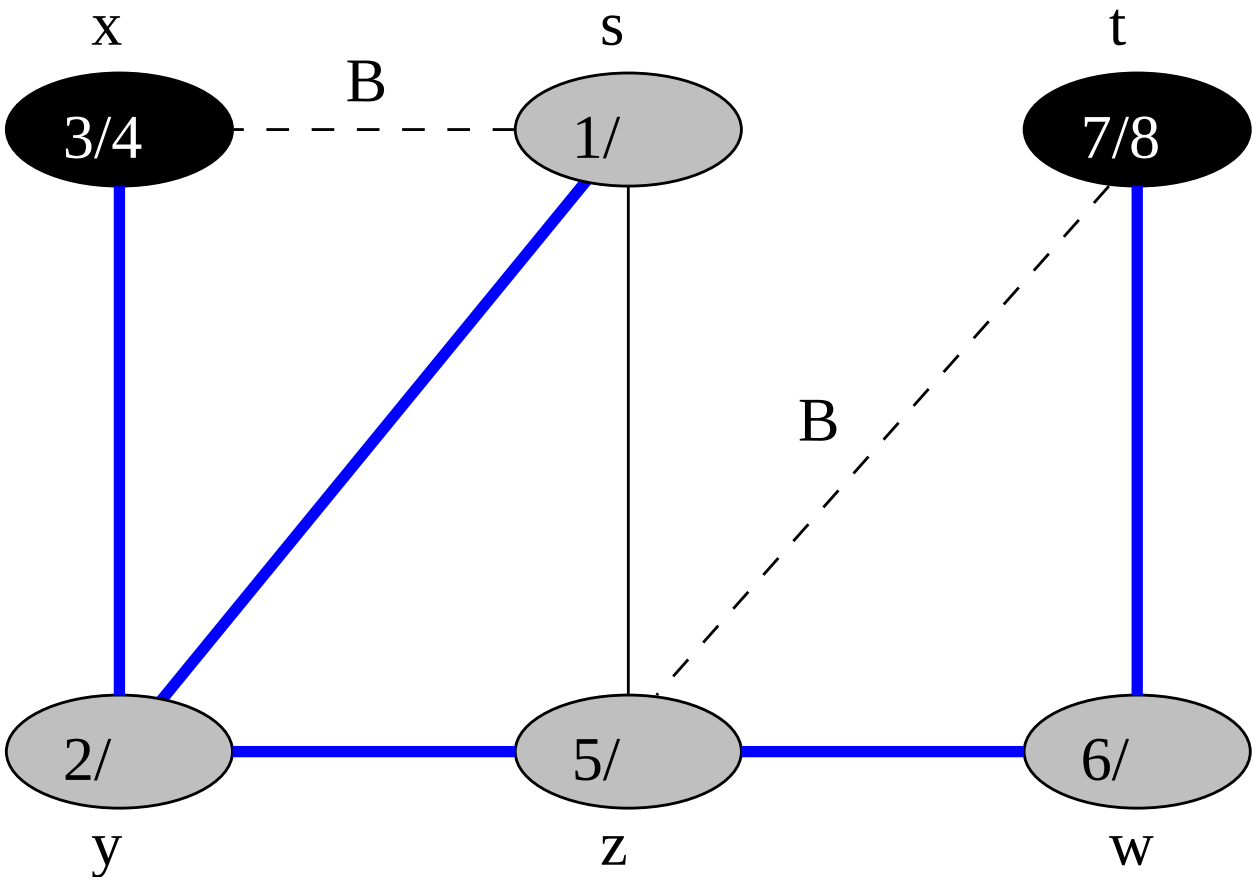
Exemplo



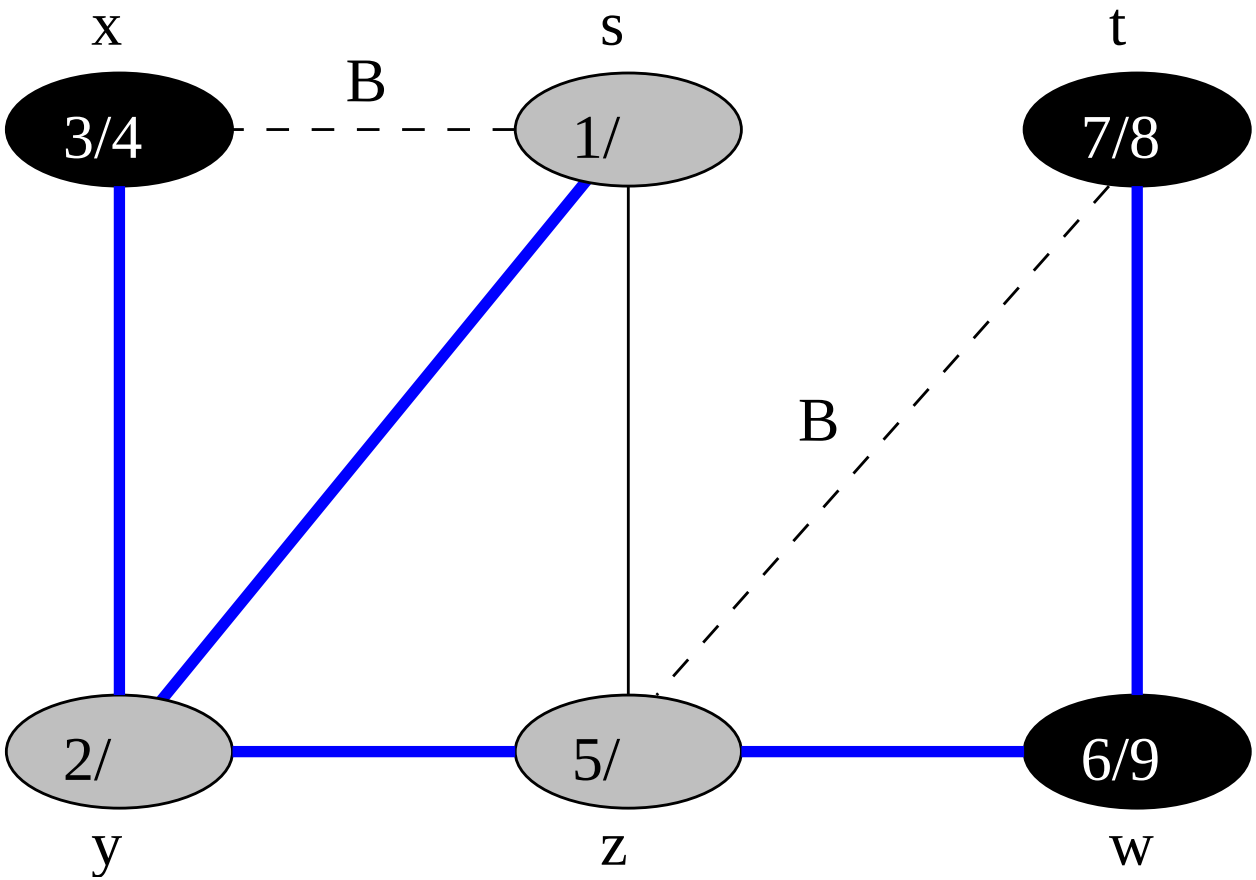
Exemplo



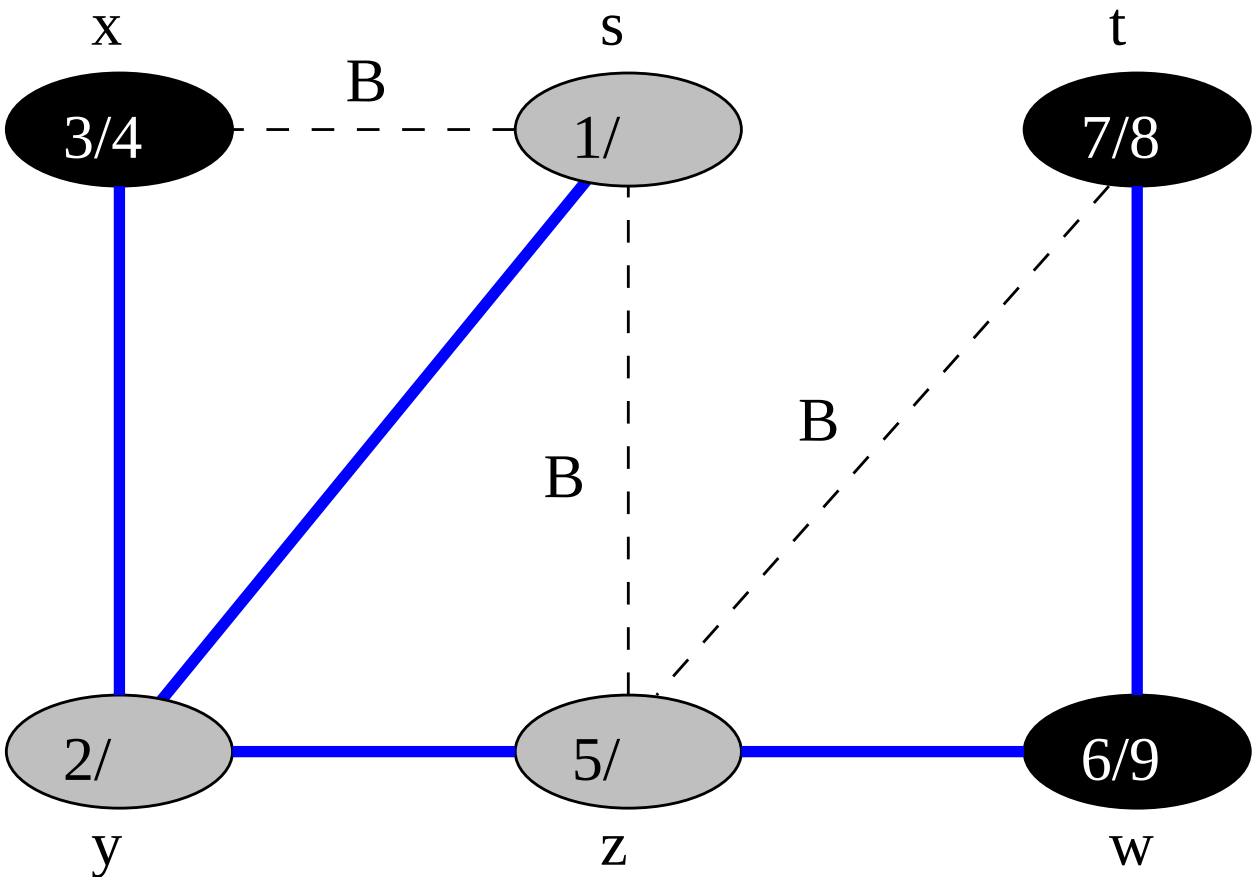
Exemplo



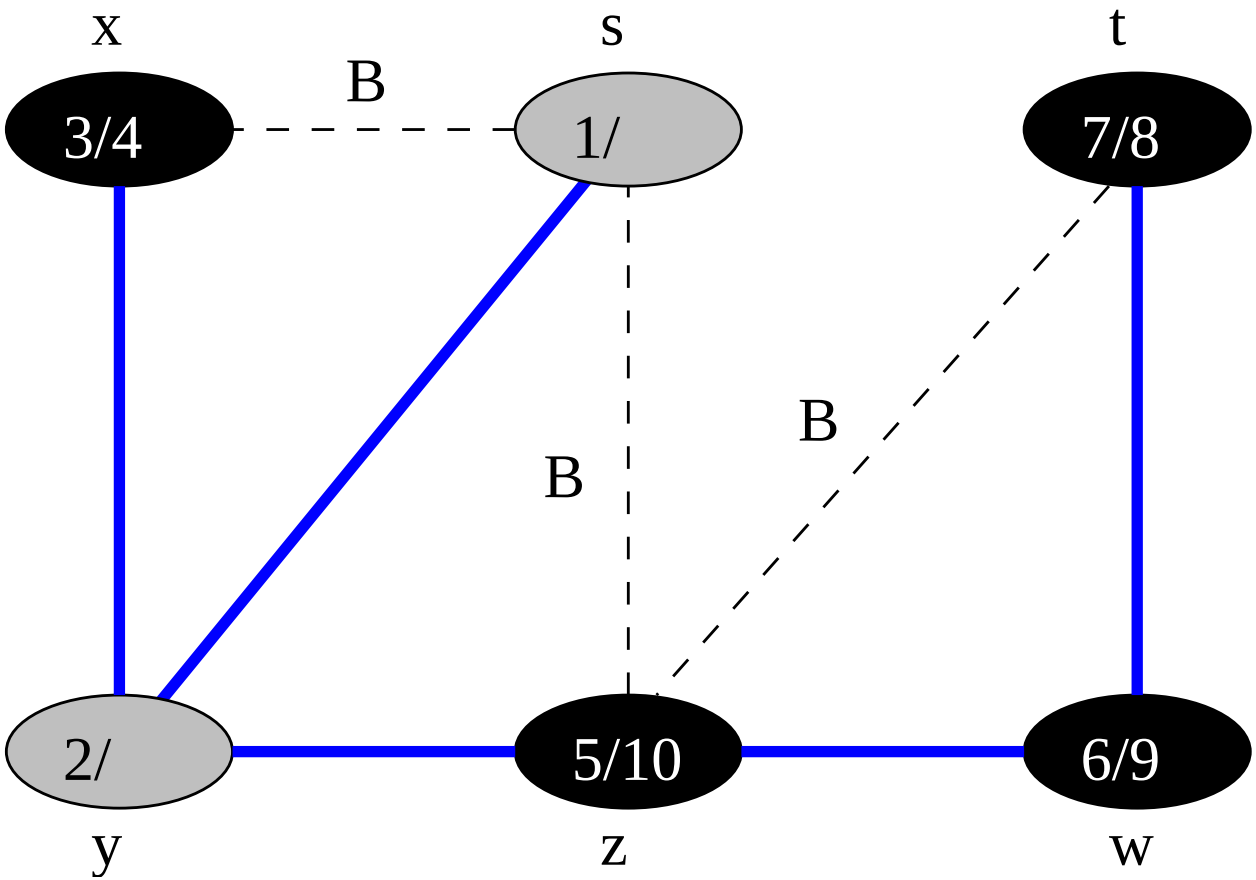
Exemplo



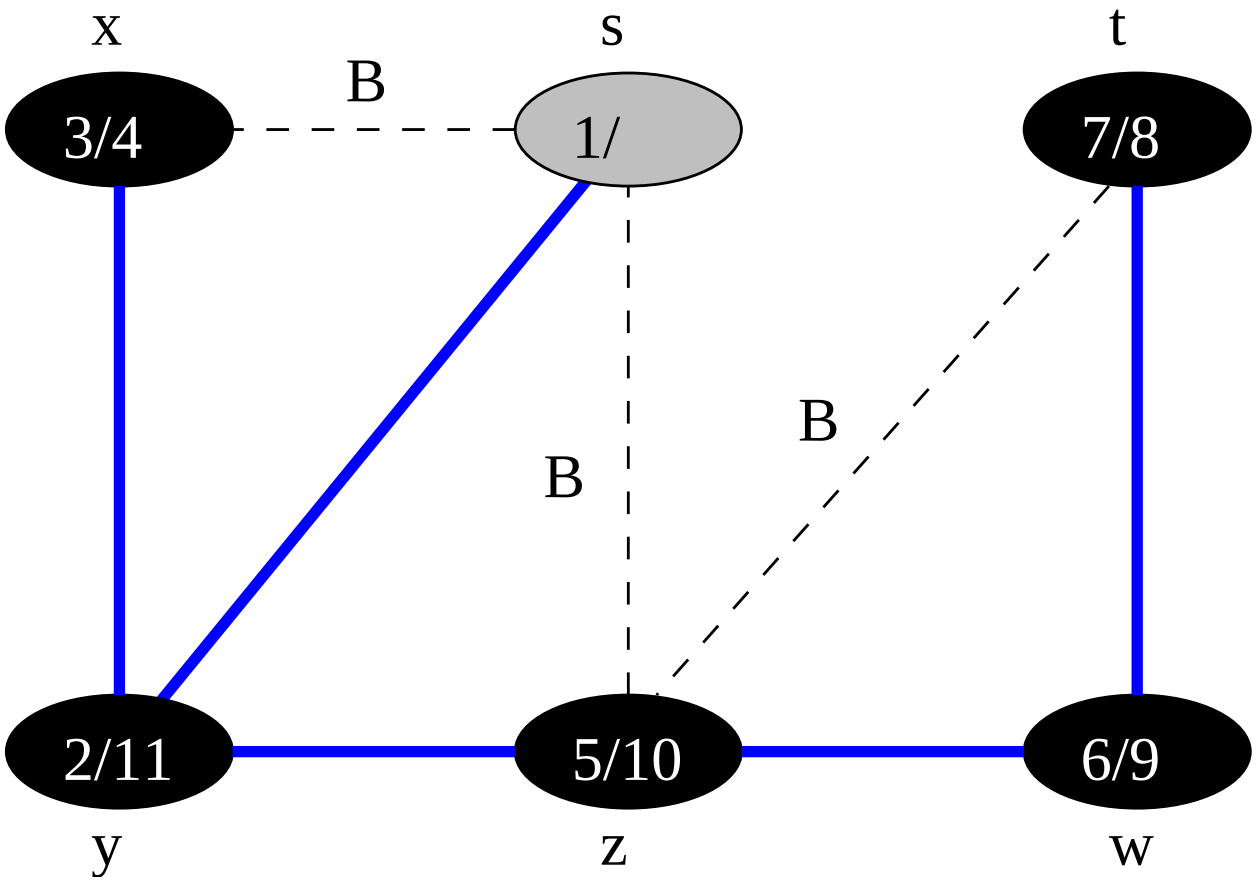
Exemplo



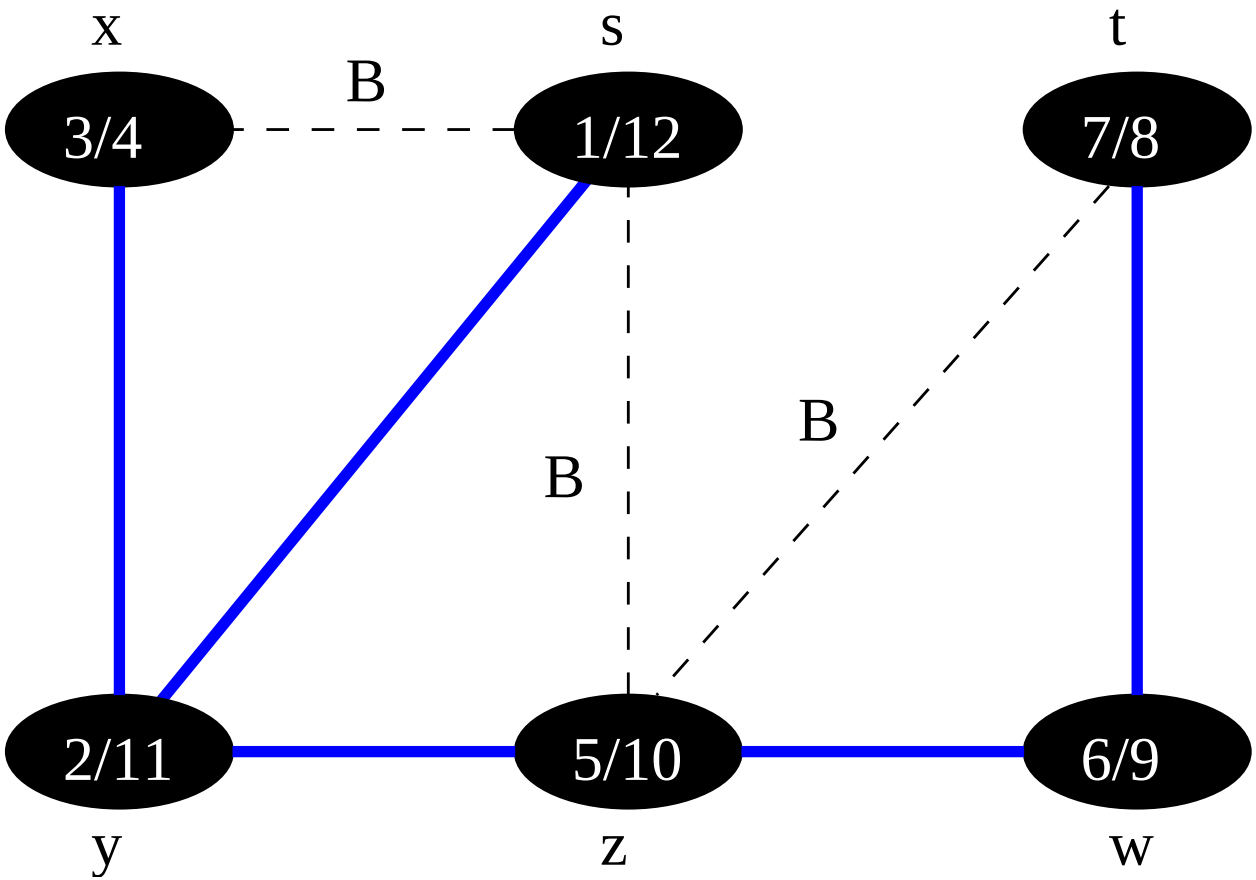
Exemplo



Exemplo



Exemplo



Exercício

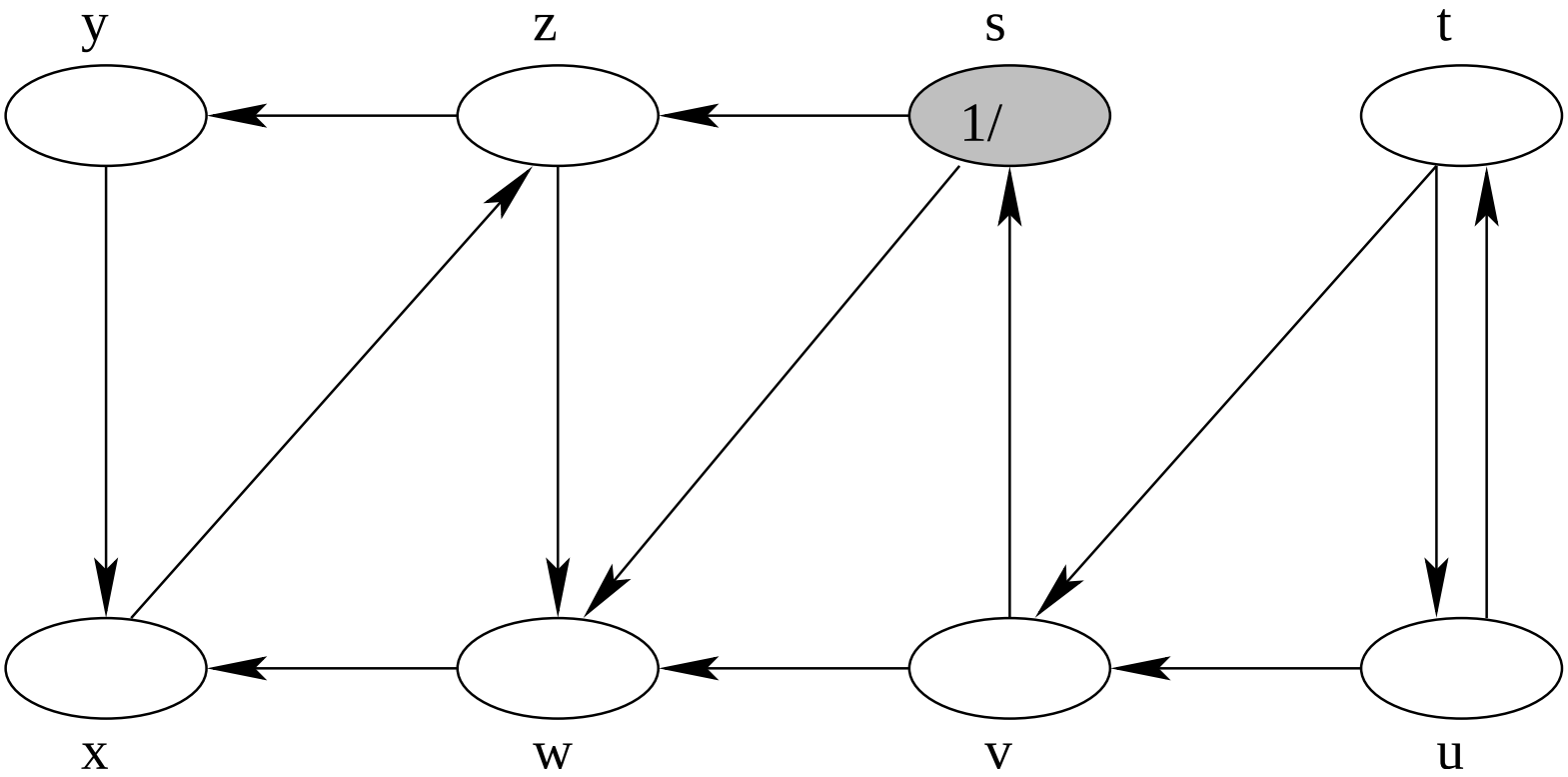
Já vimos o seguinte resultado.

Teorema. Um grafo G é bipartido se, e somente se, não contém um ciclo ímpar.

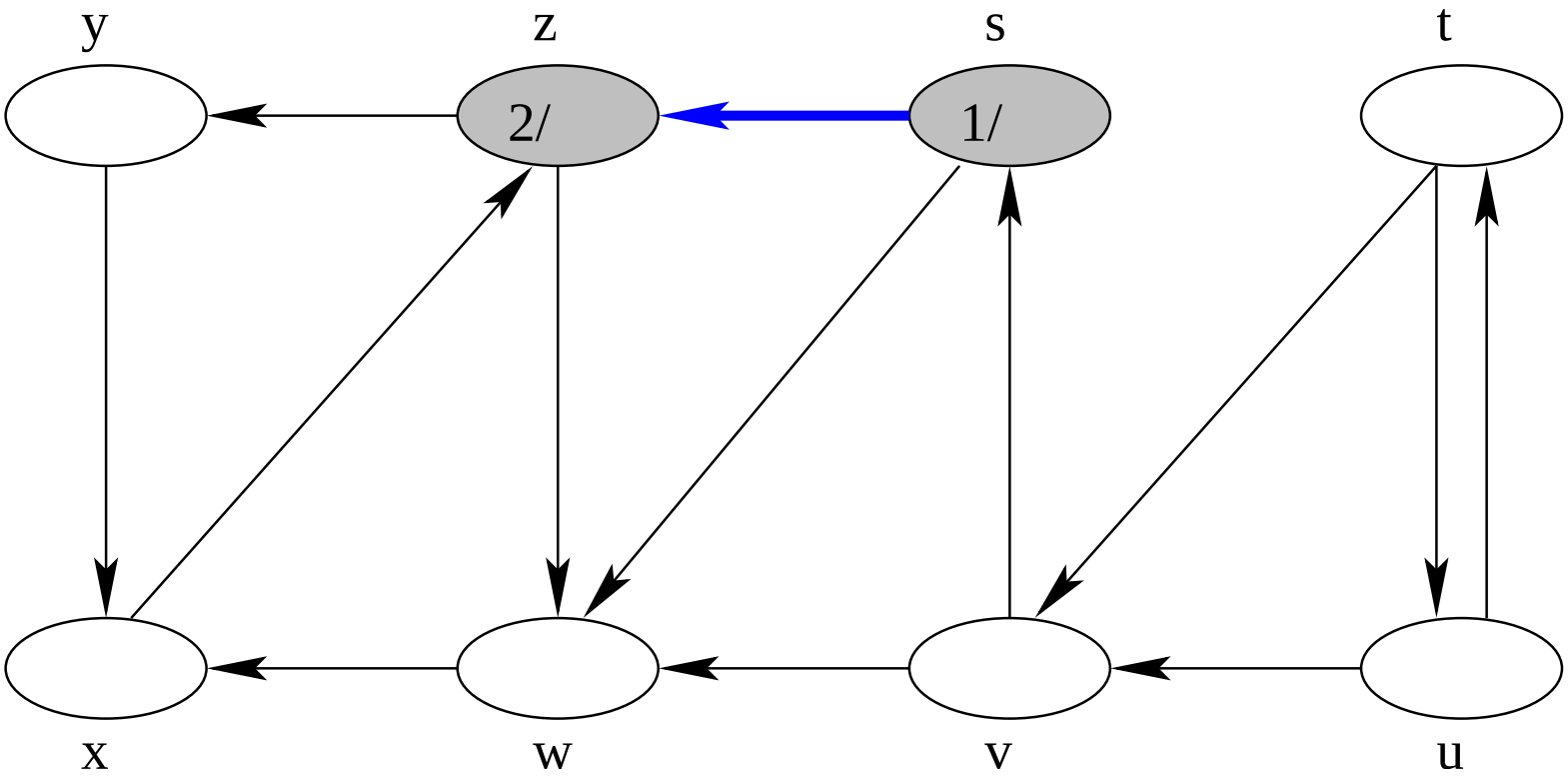
Exercício. Projete um algoritmo linear baseado em DFS que dado um grafo G representado por listas de adjacências, devolve

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

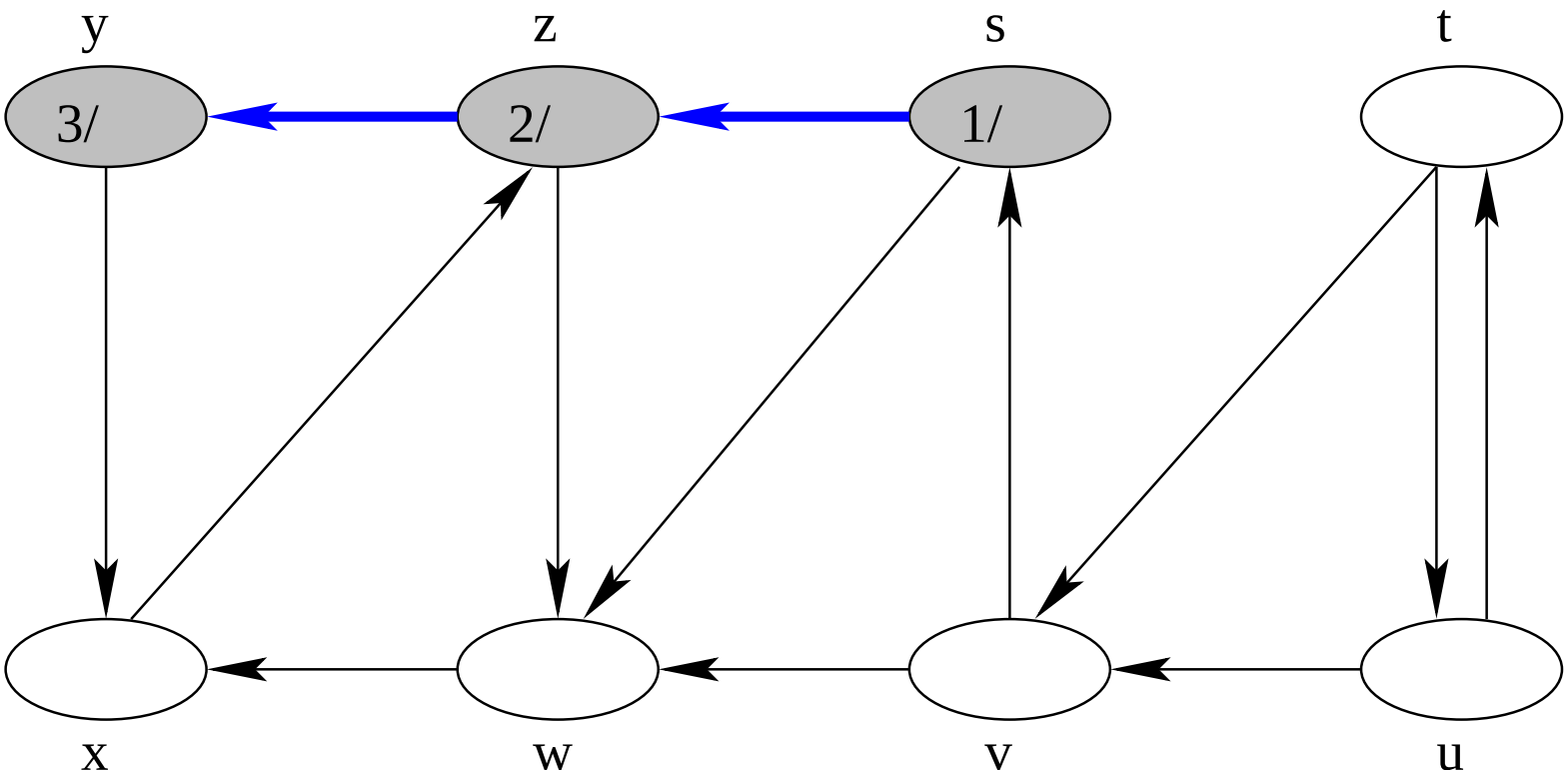
Exemplo DFS



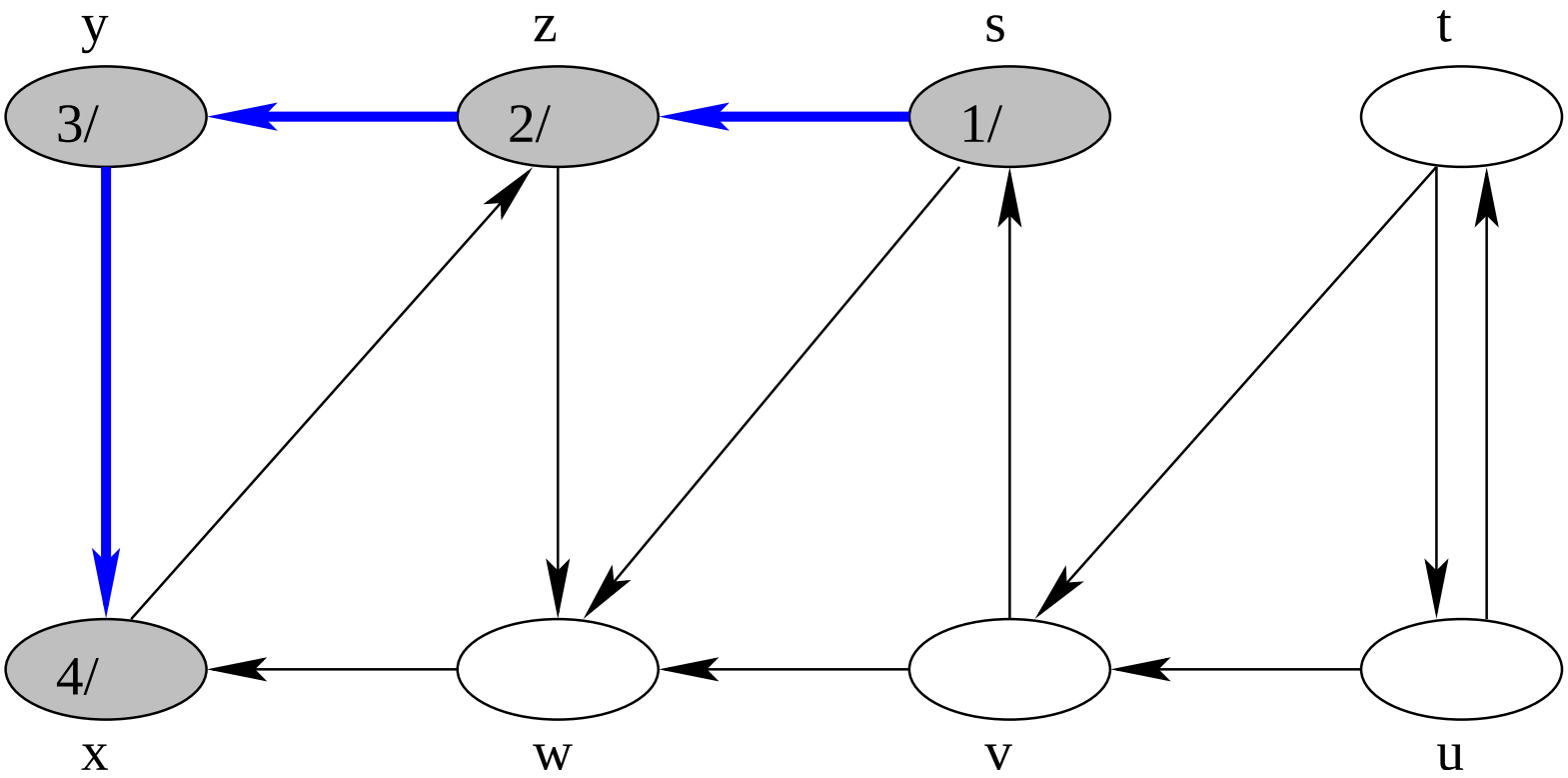
Exemplo DFS



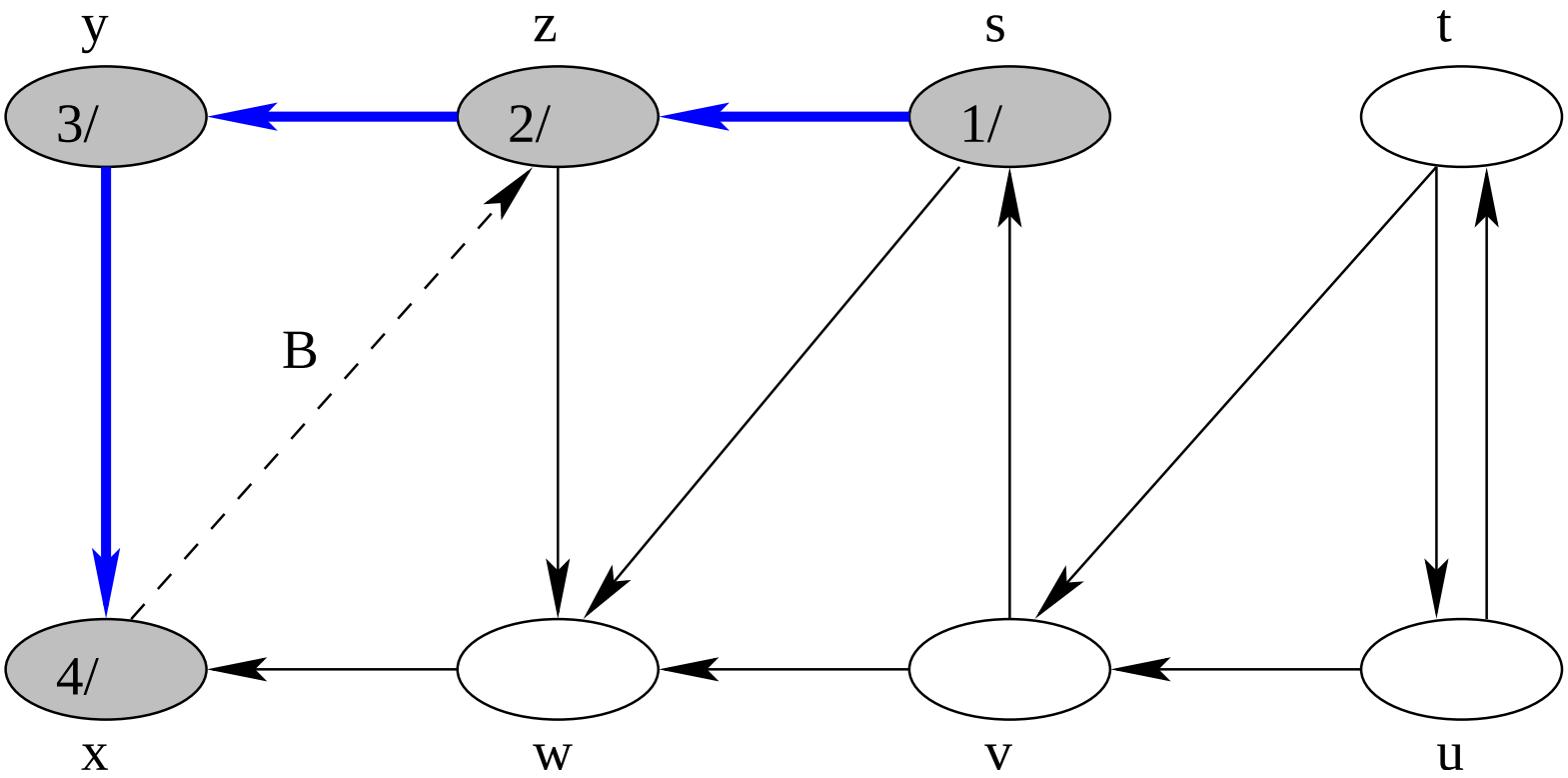
Exemplo DFS



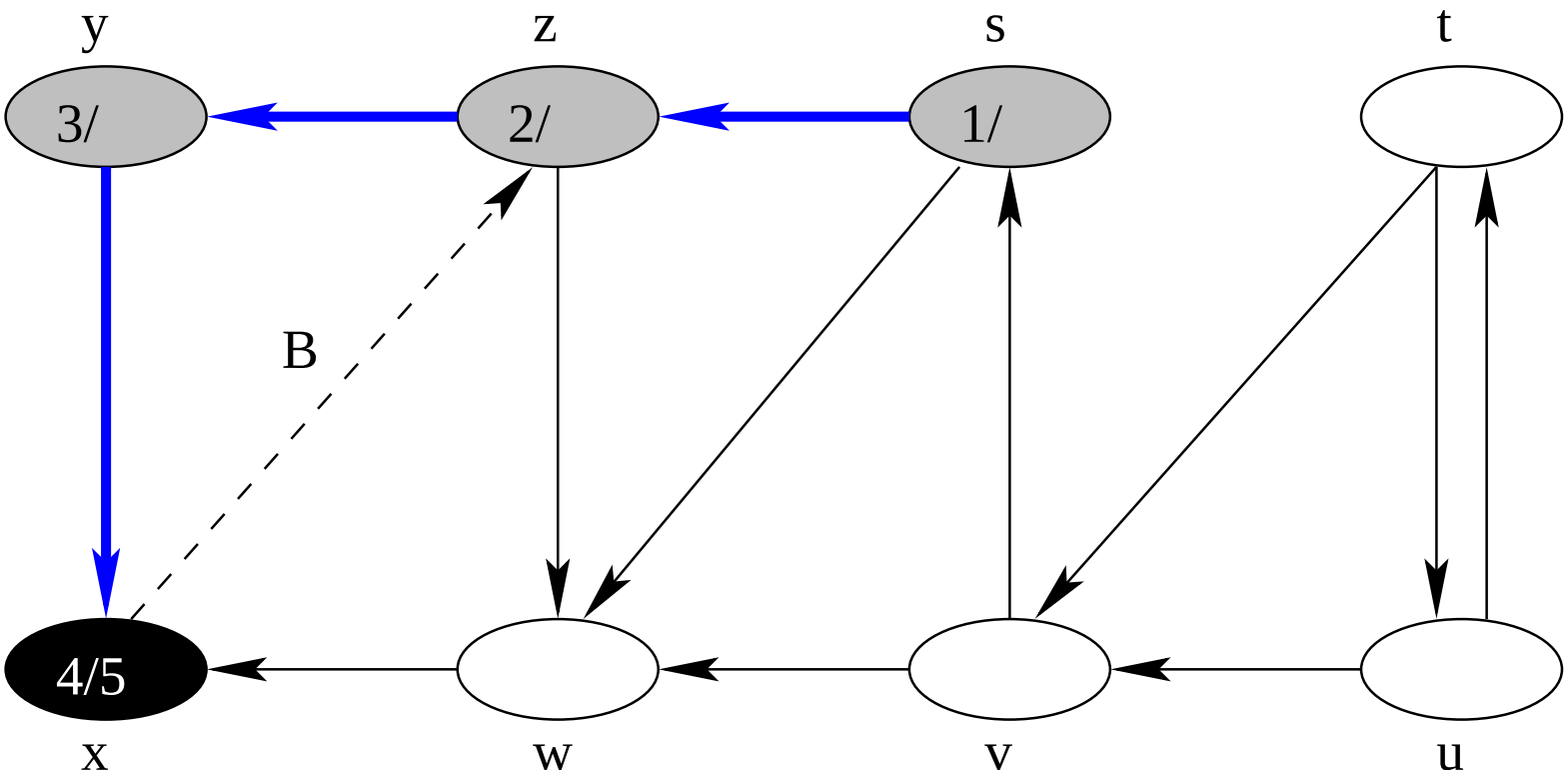
Exemplo DFS



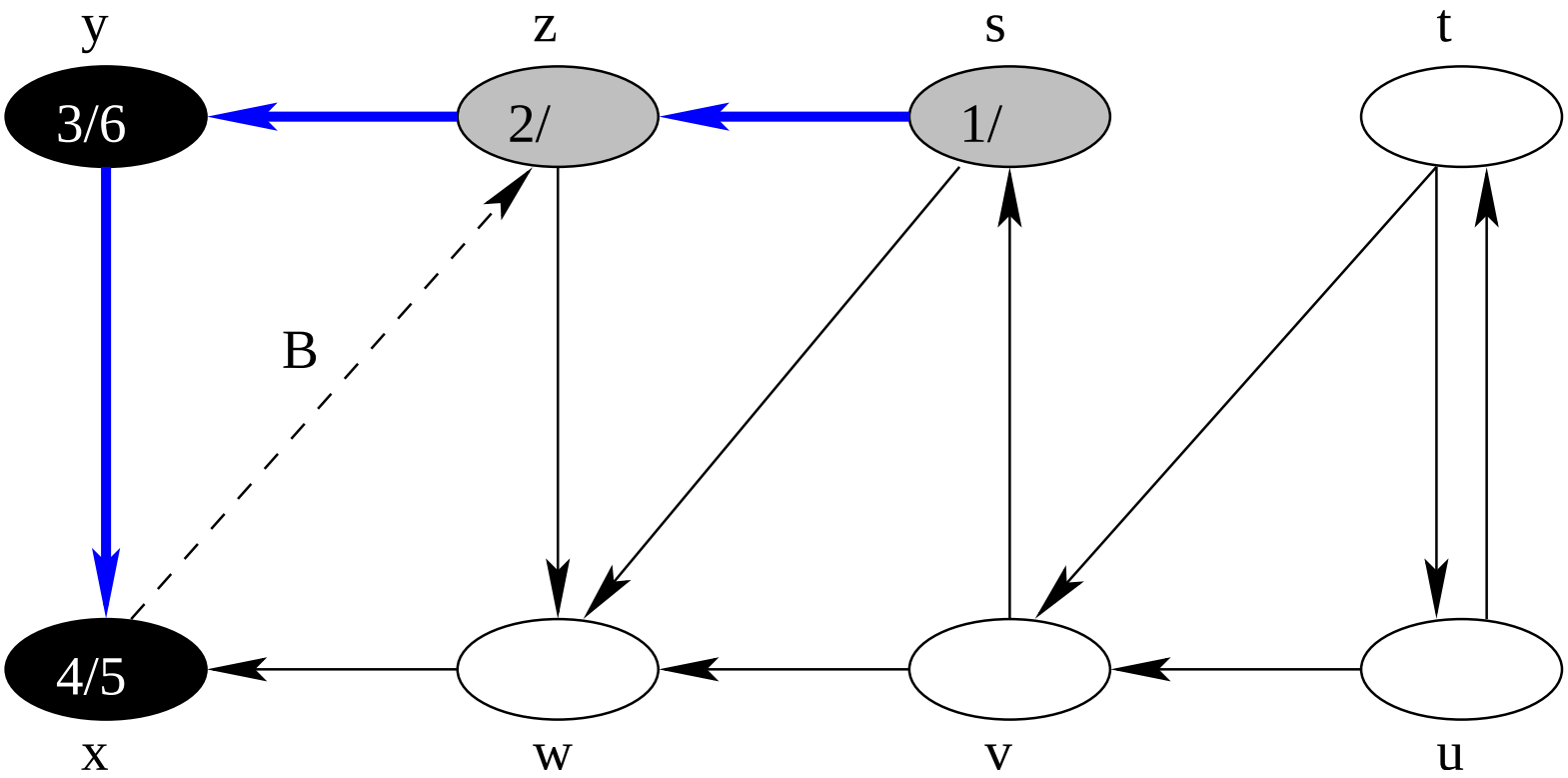
Exemplo DFS



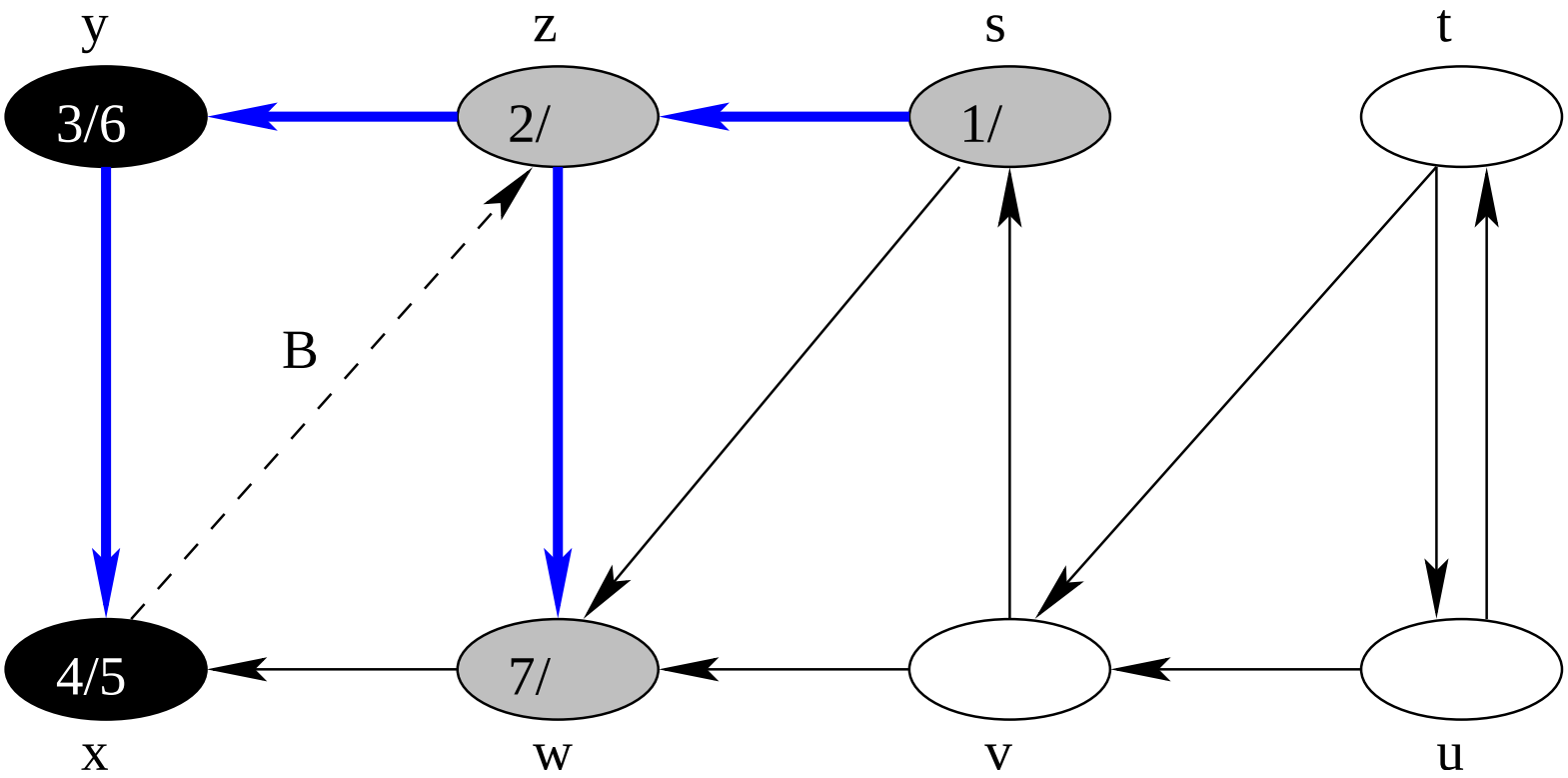
Exemplo DFS



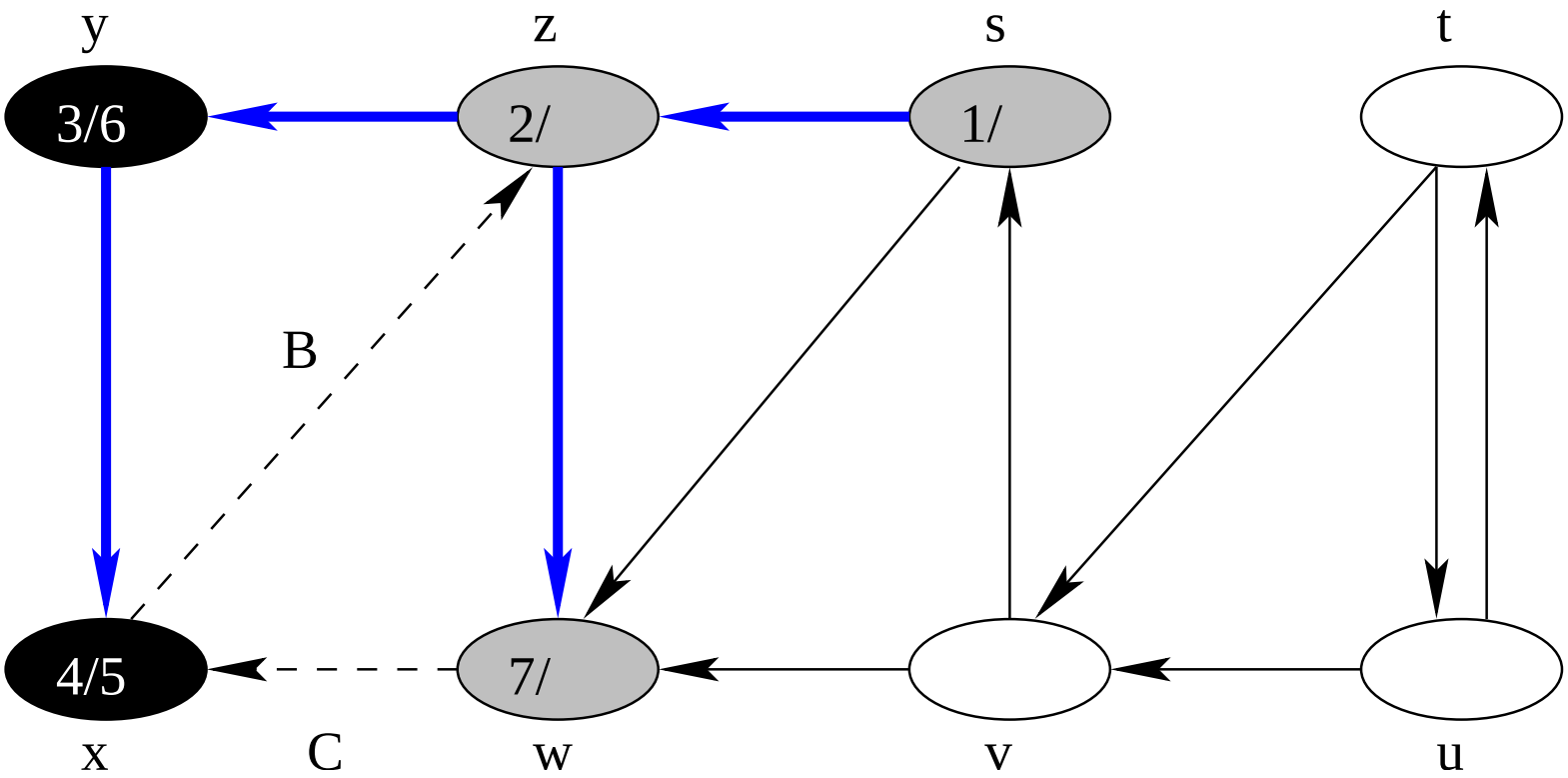
Exemplo DFS



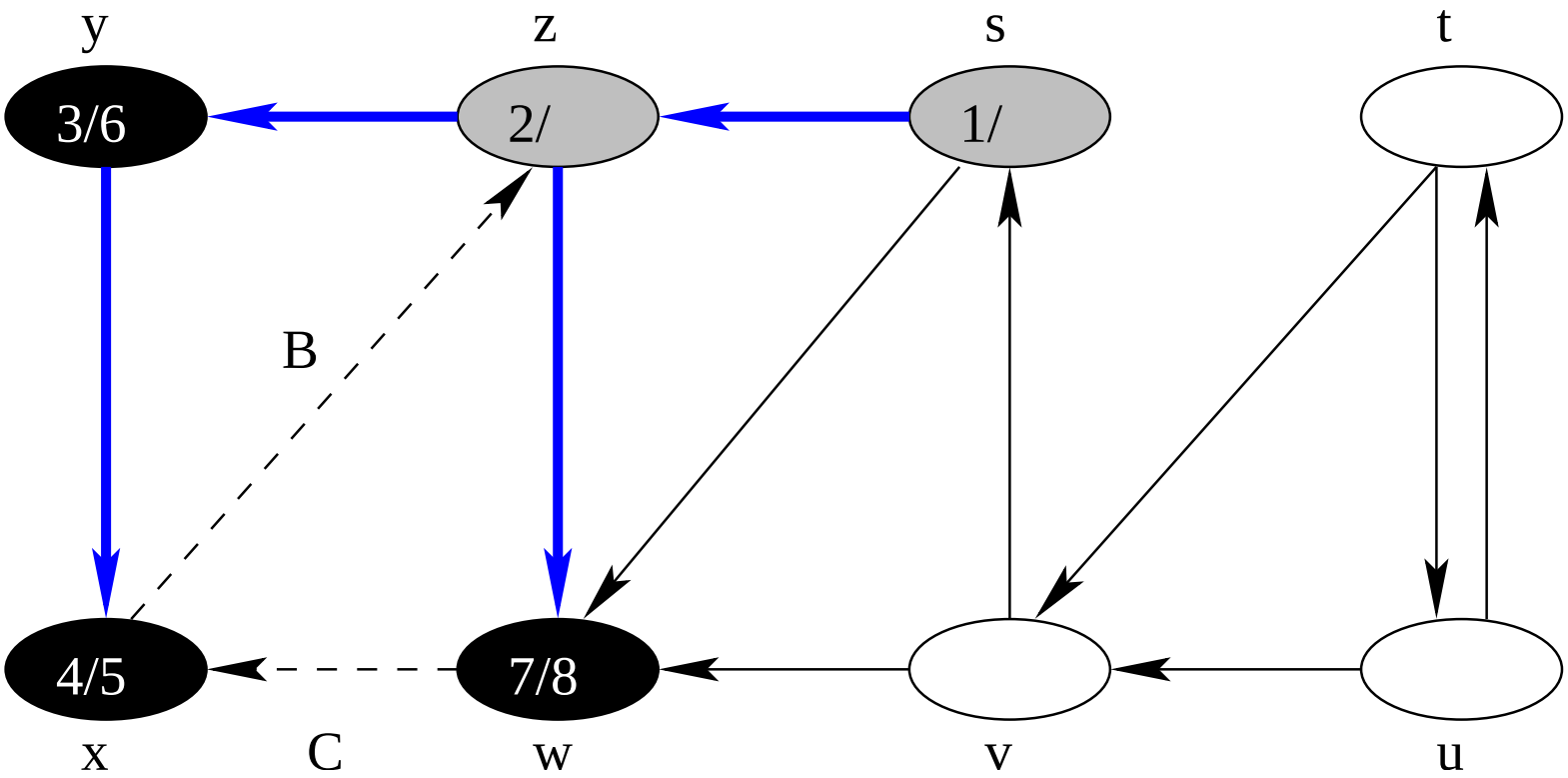
Exemplo DFS



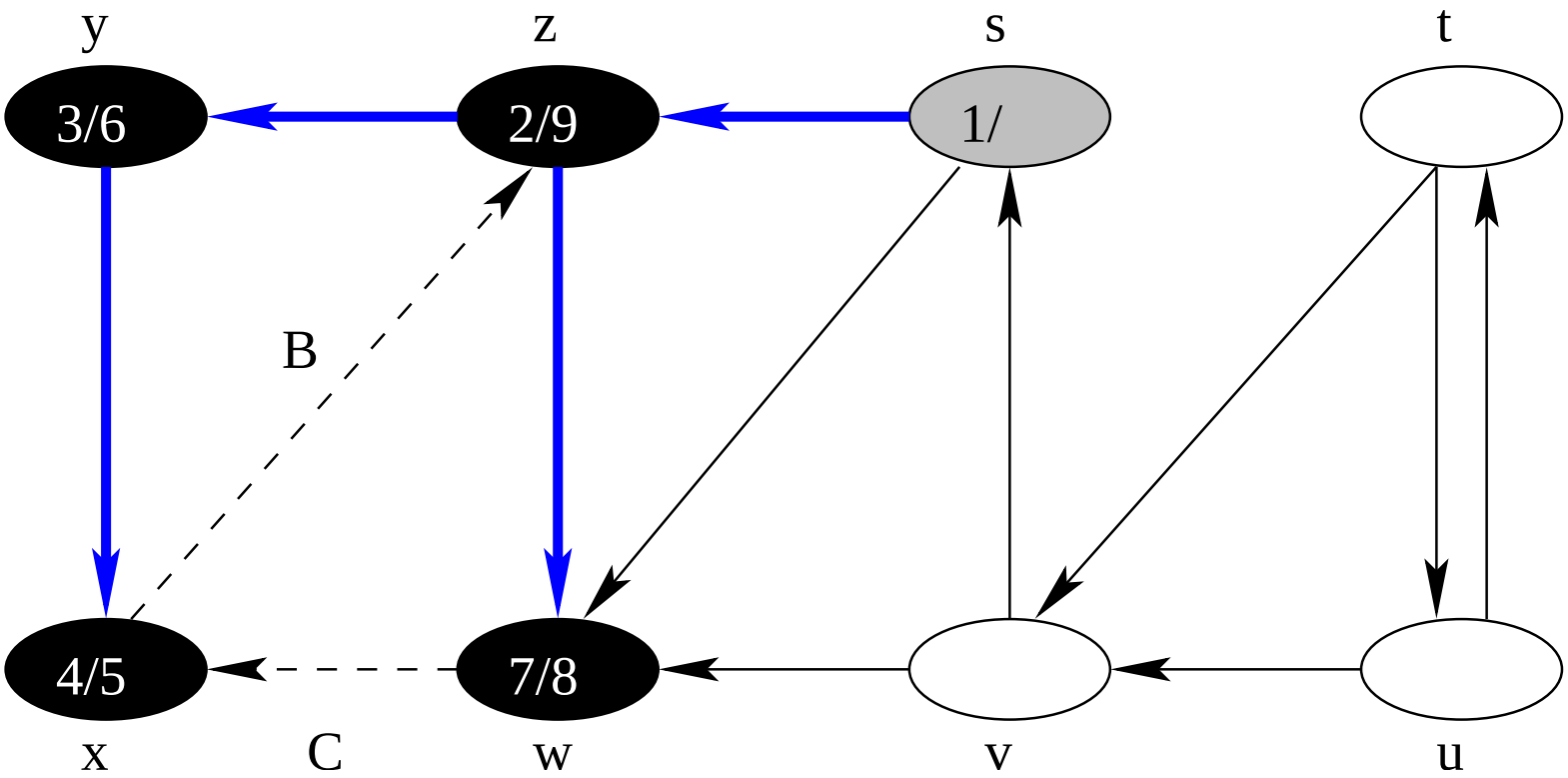
Exemplo DFS



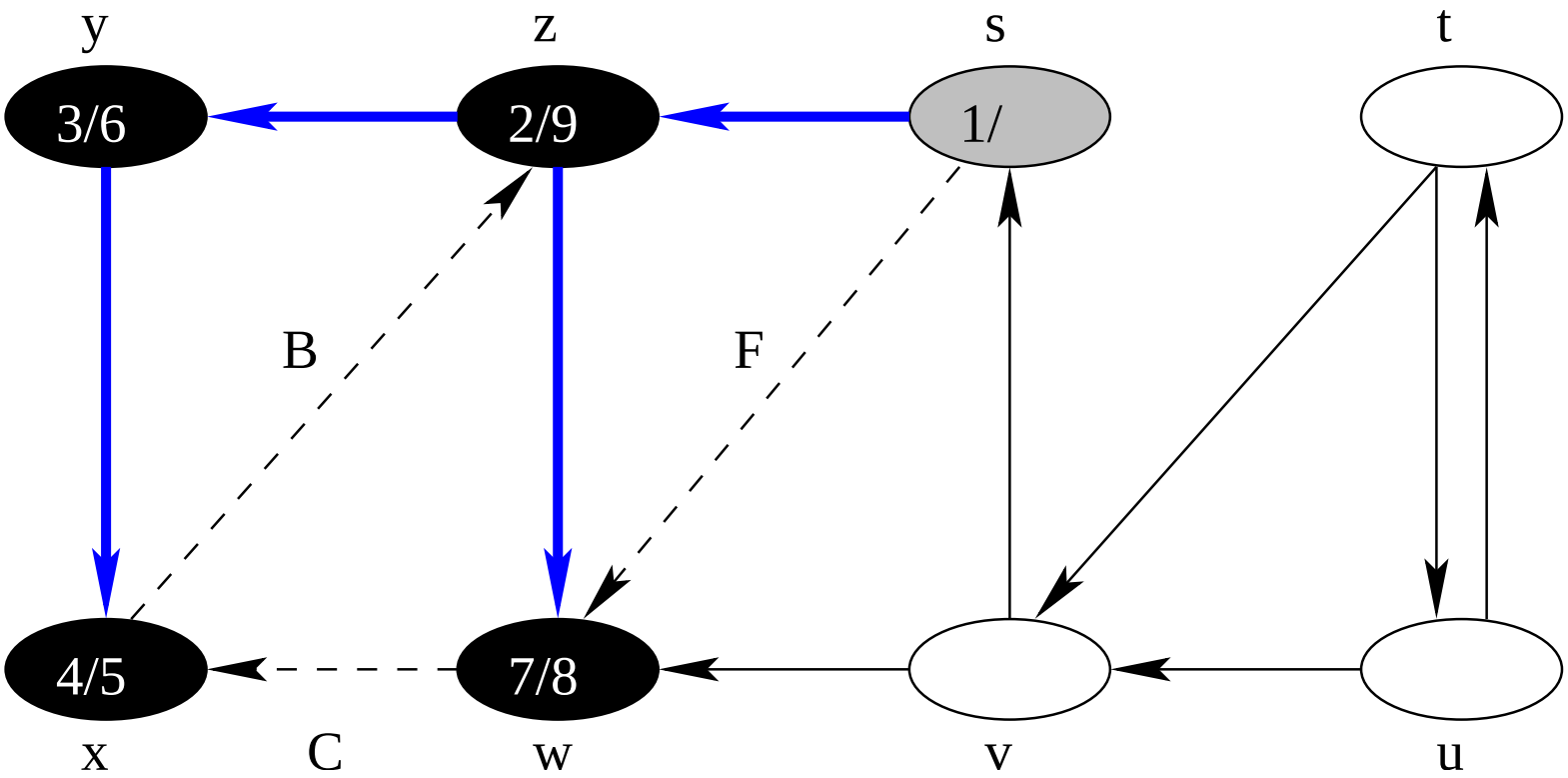
Exemplo DFS



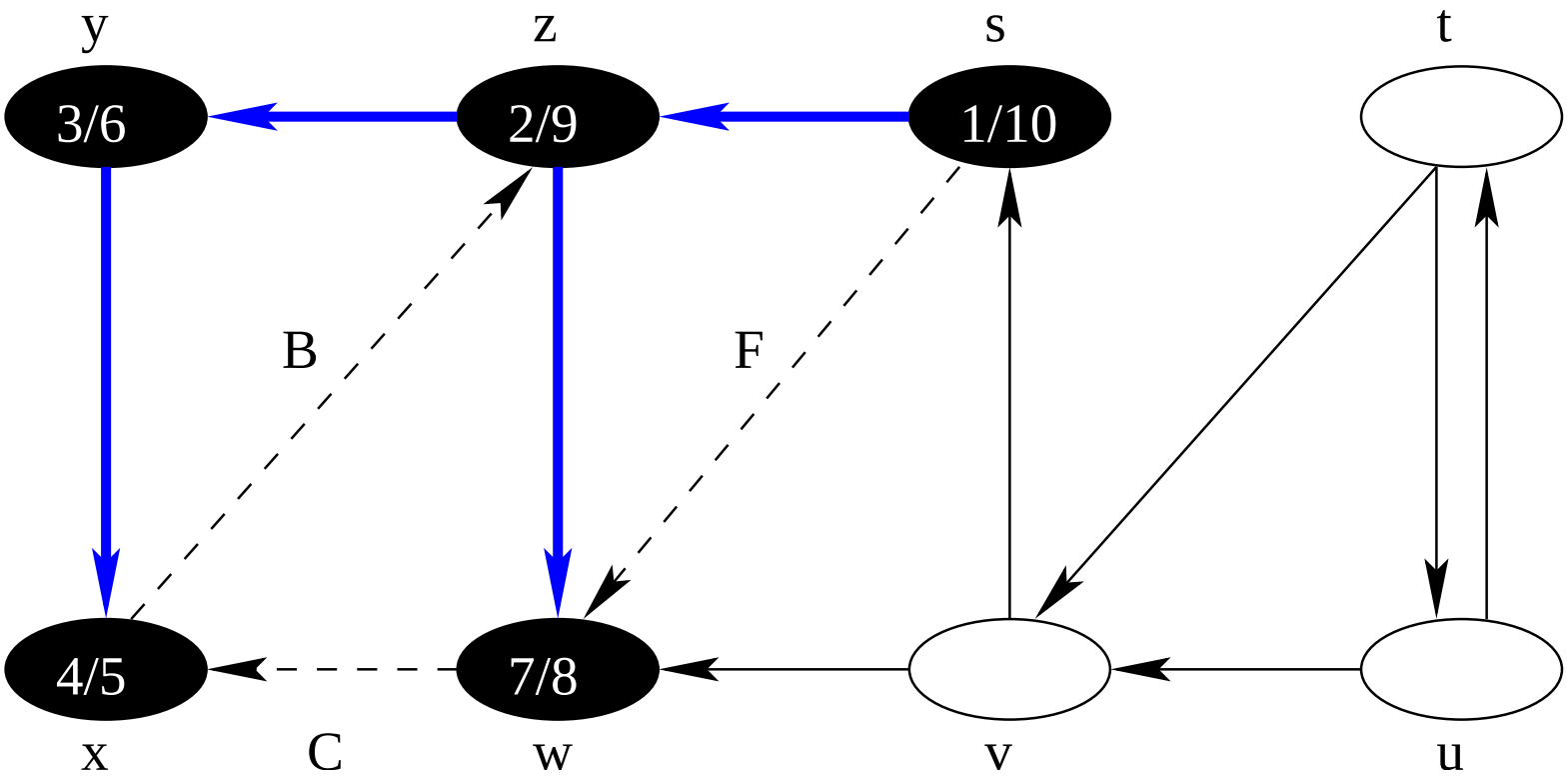
Exemplo DFS



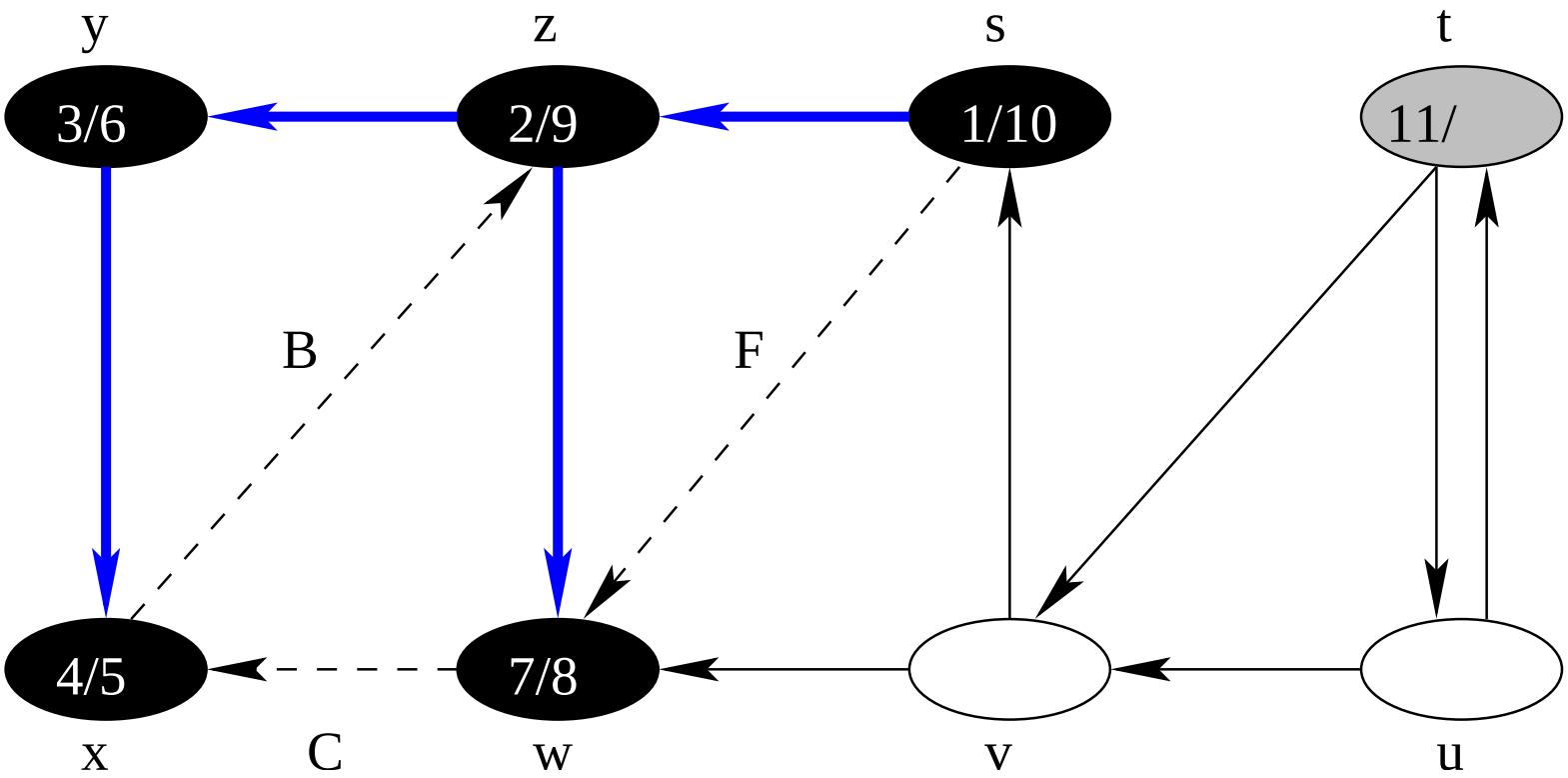
Exemplo DFS



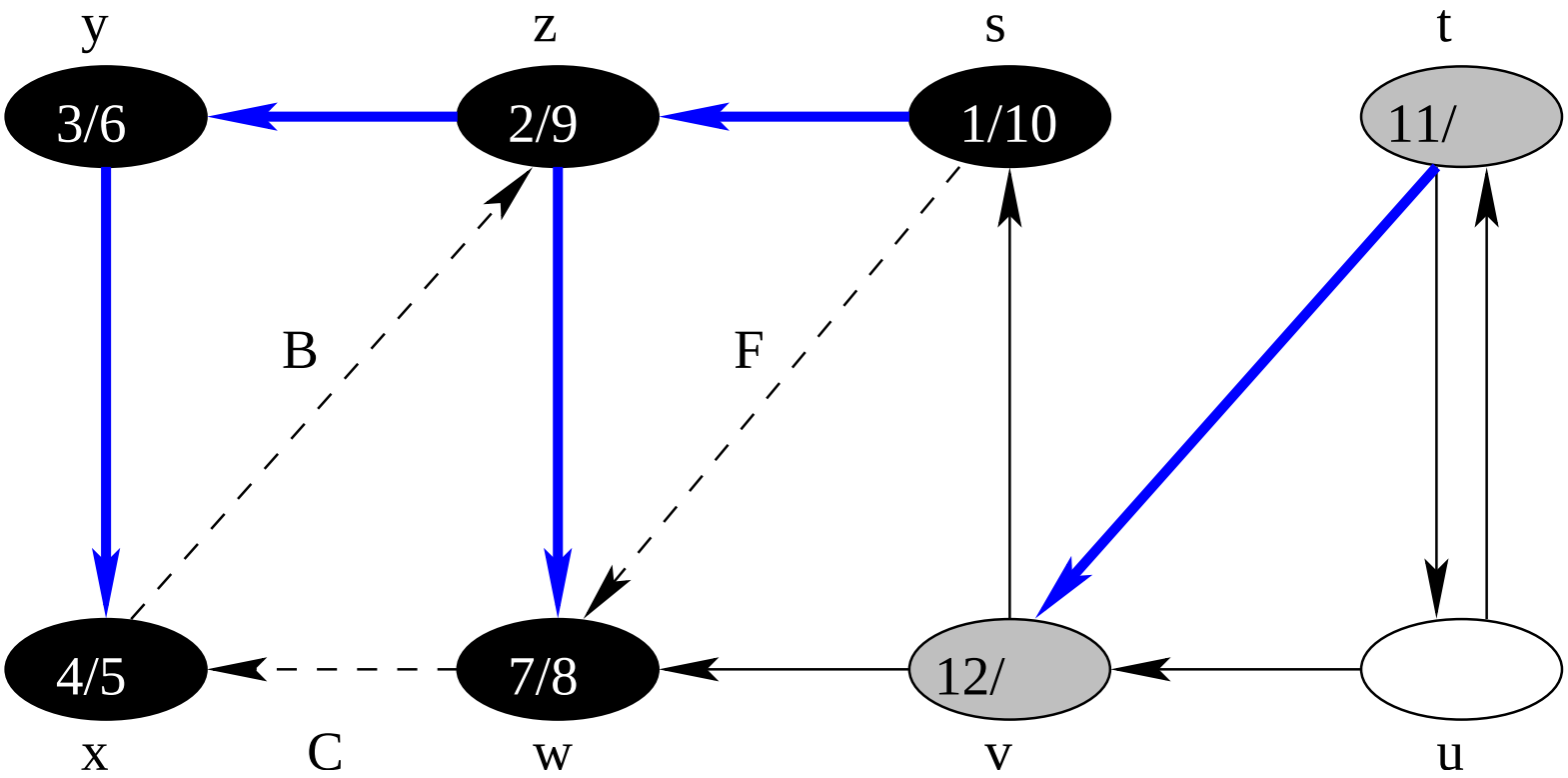
Exemplo DFS



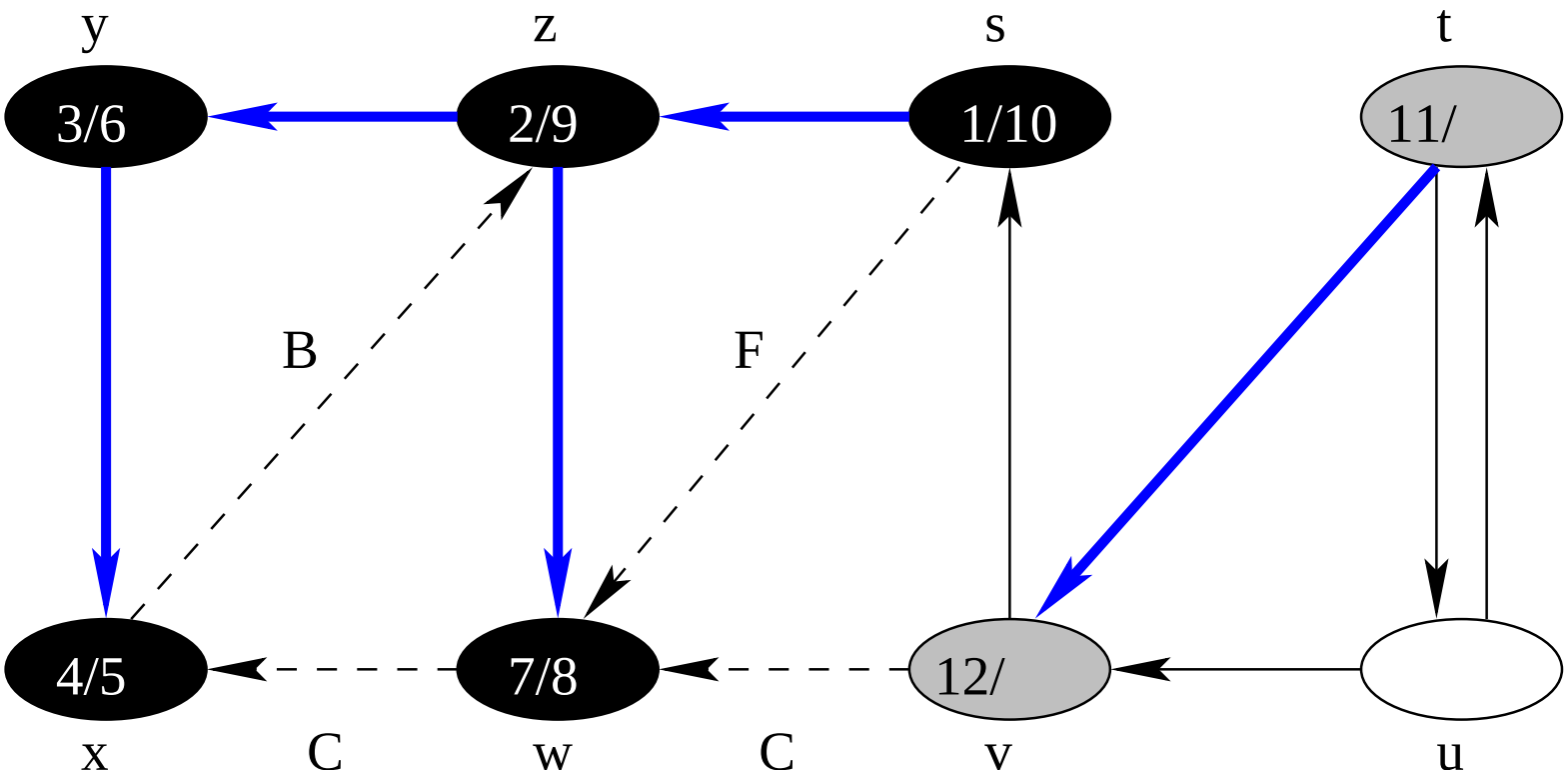
Exemplo DFS



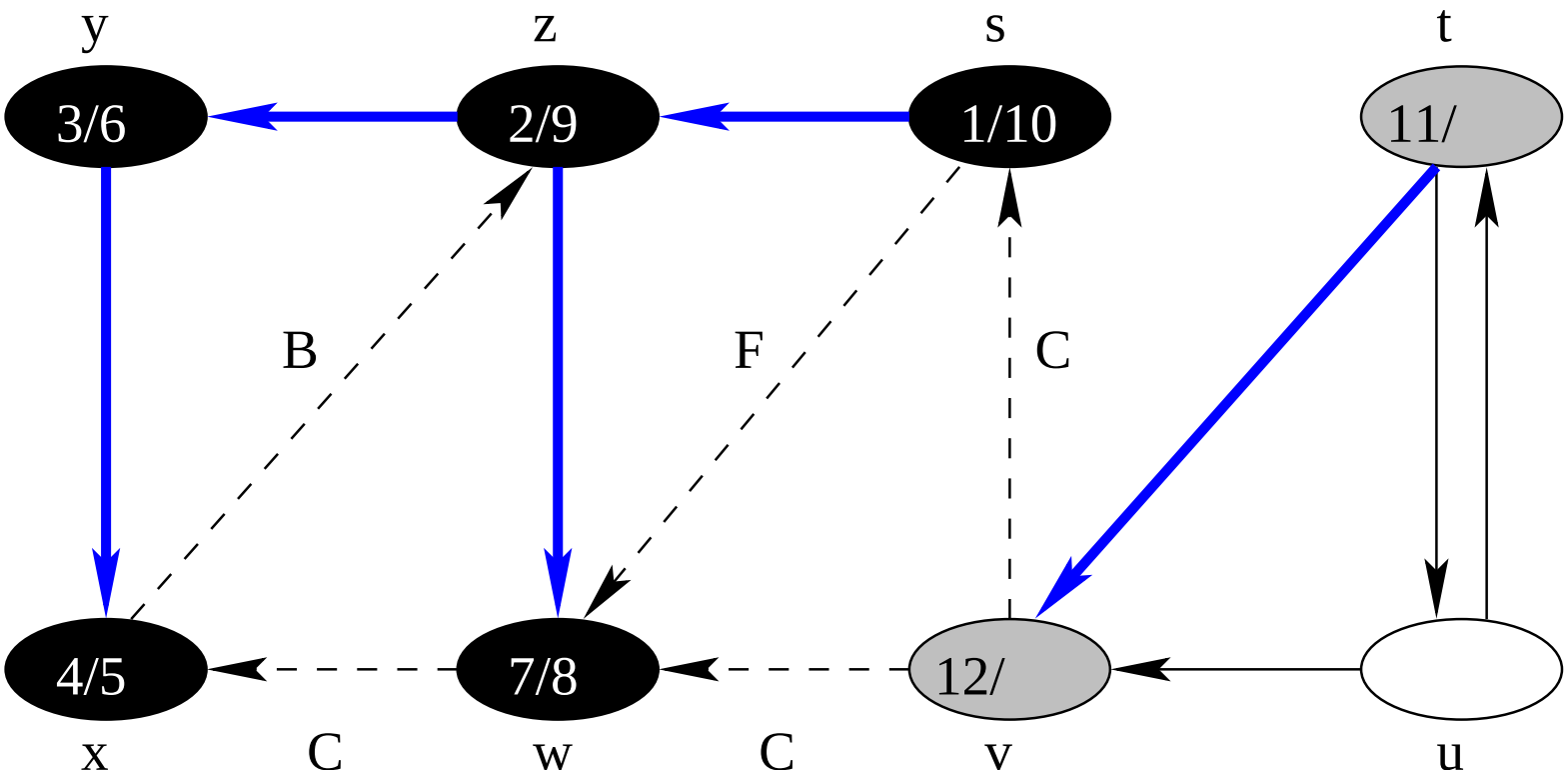
Exemplo DFS



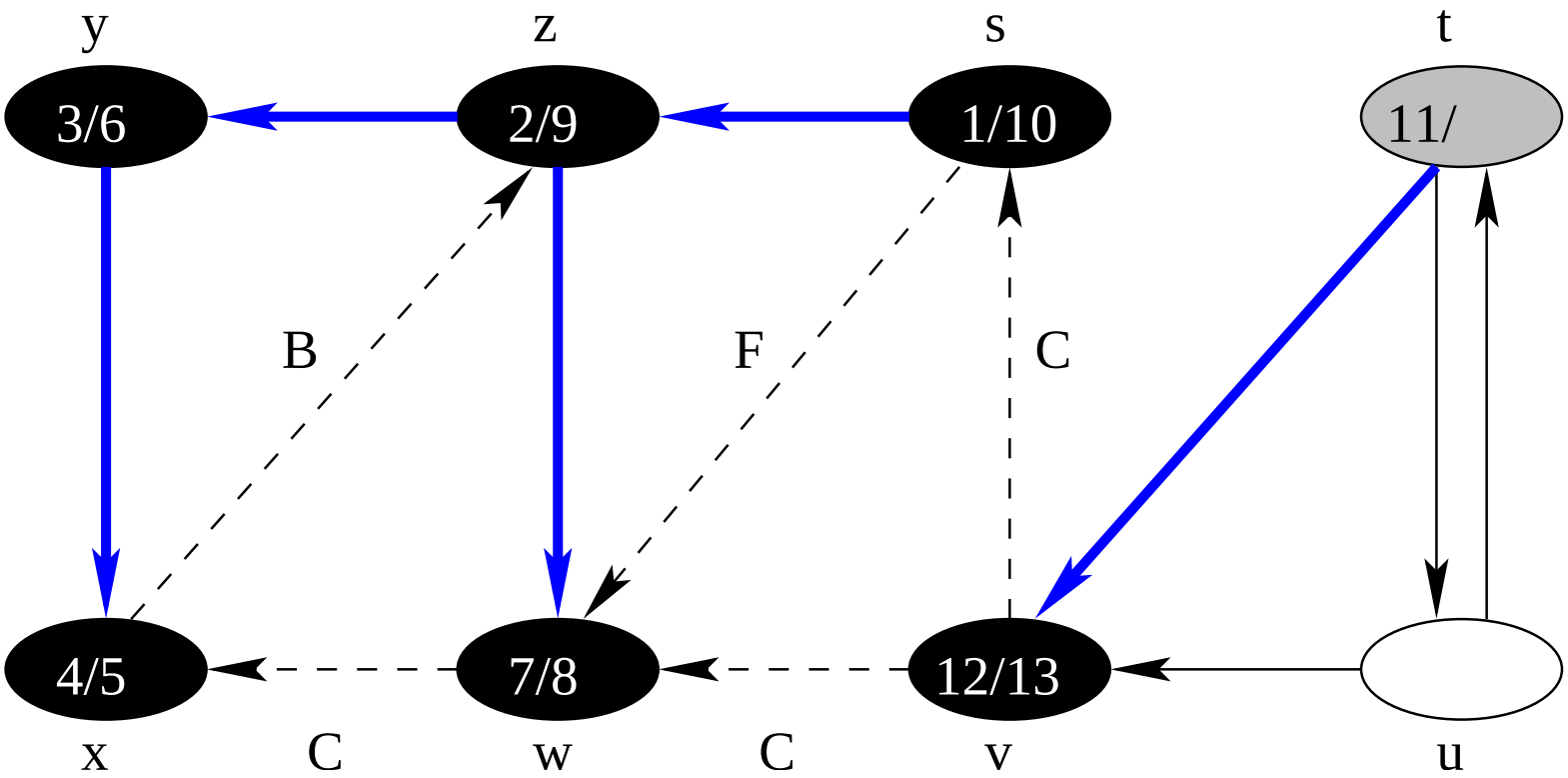
Exemplo DFS



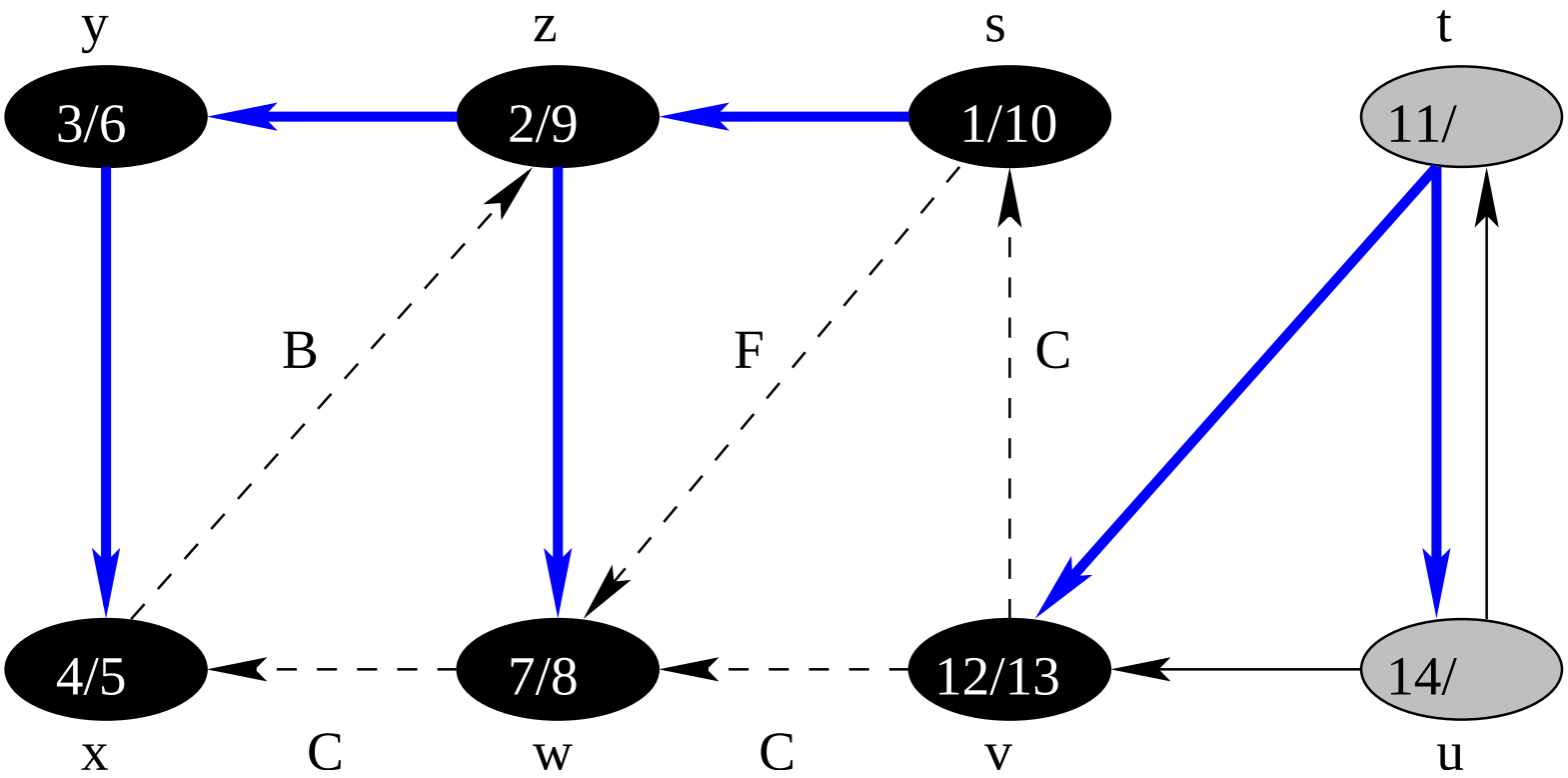
Exemplo DFS



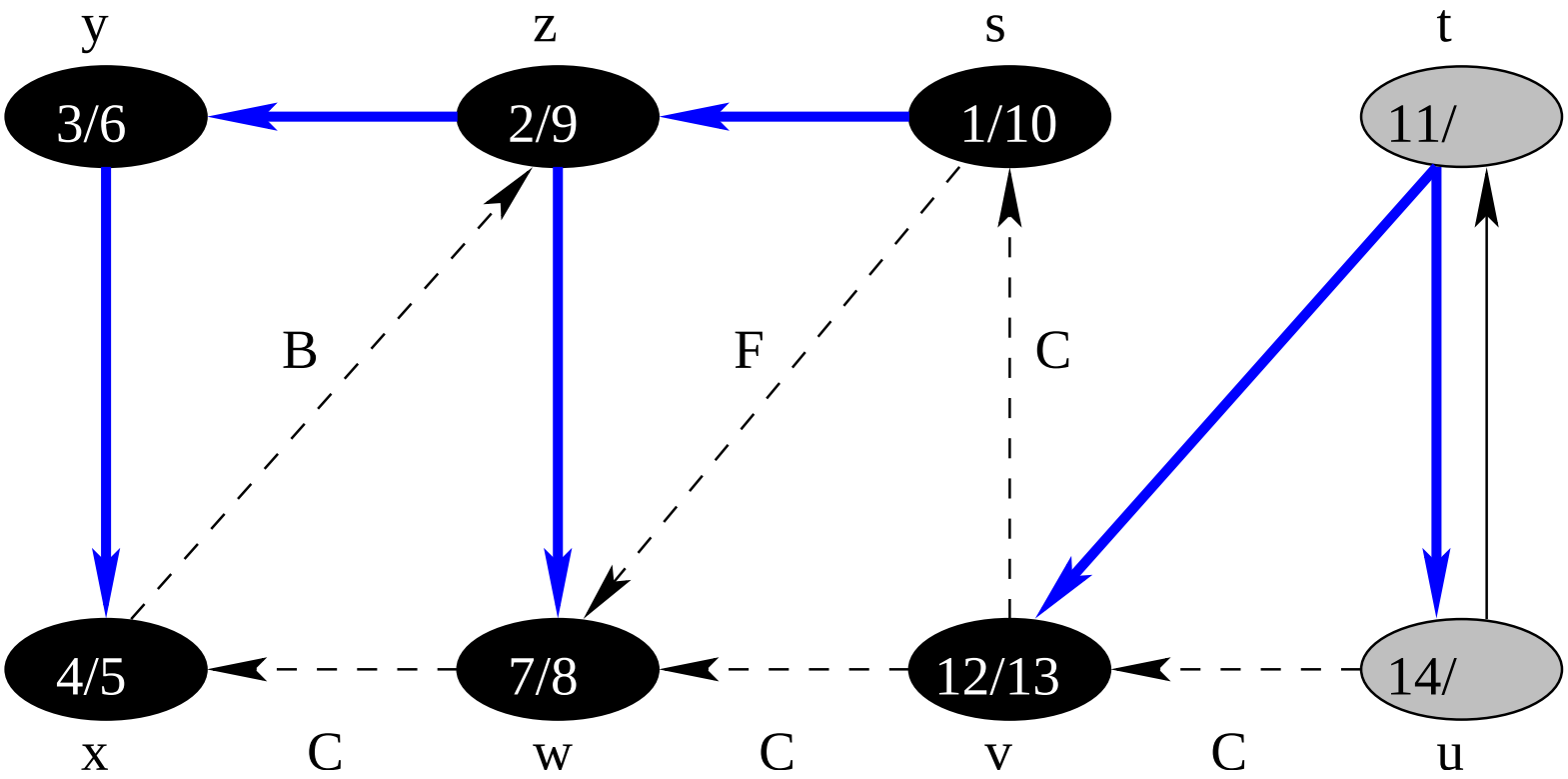
Exemplo DFS



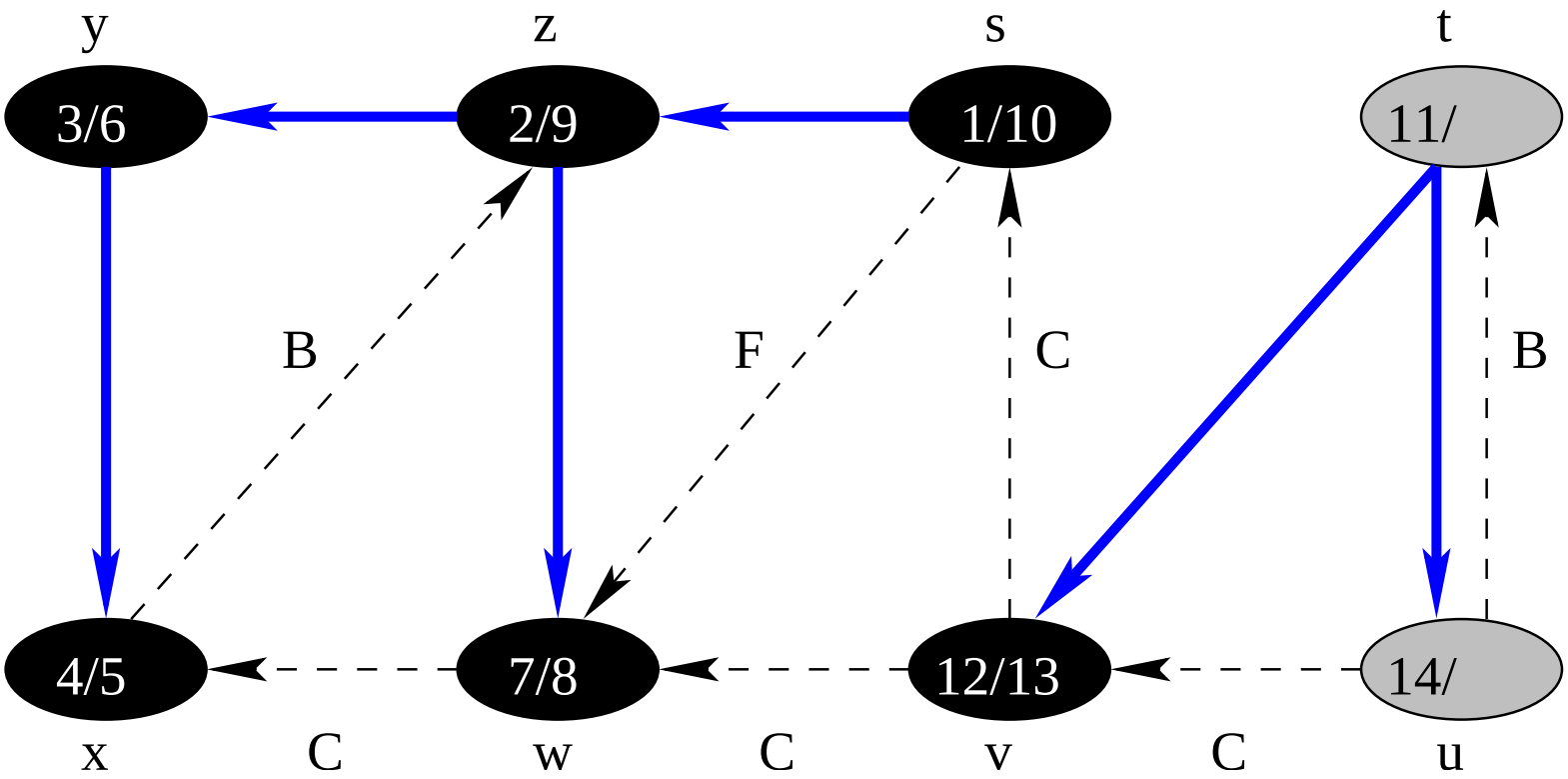
Exemplo DFS



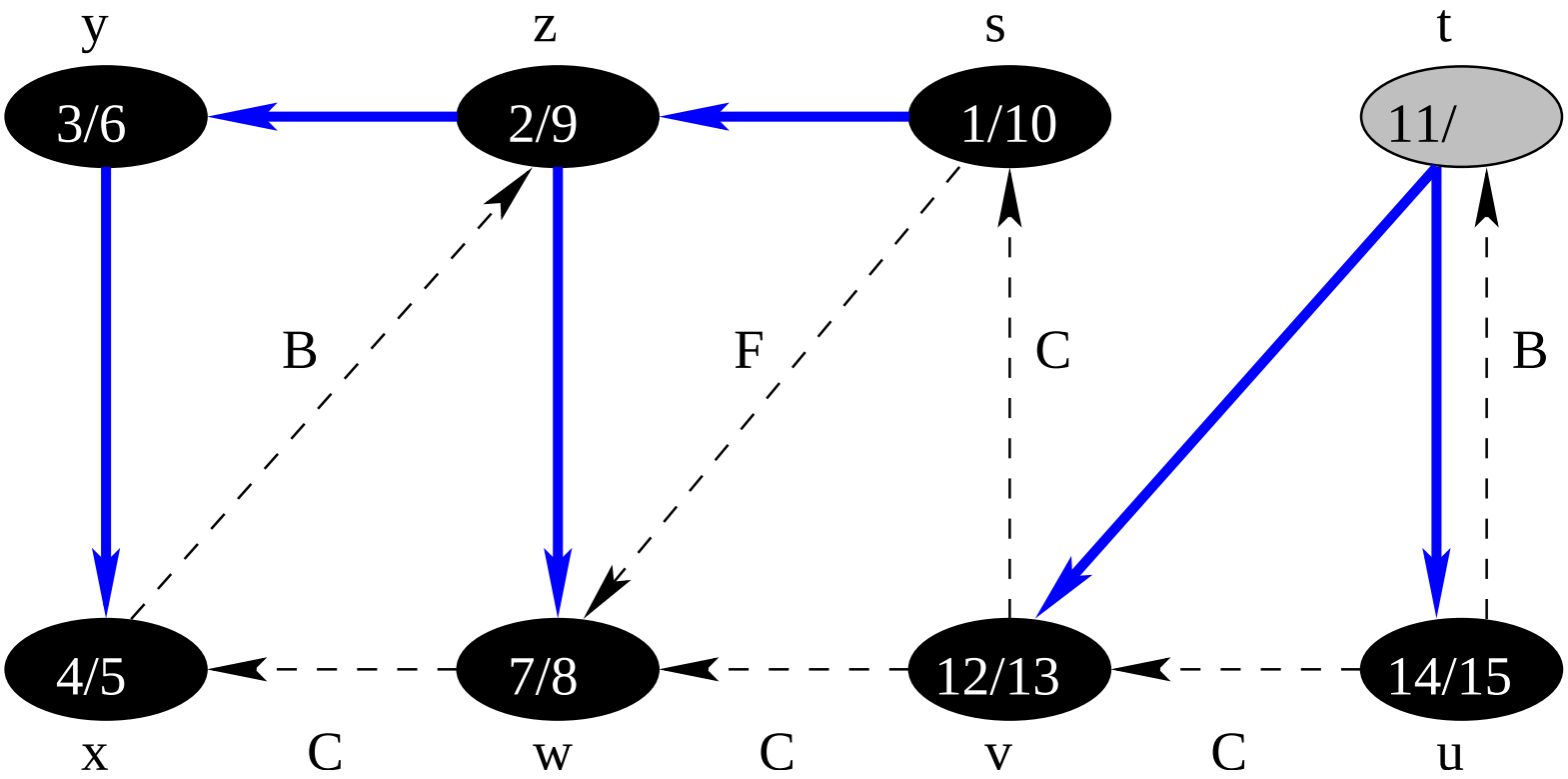
Exemplo DFS



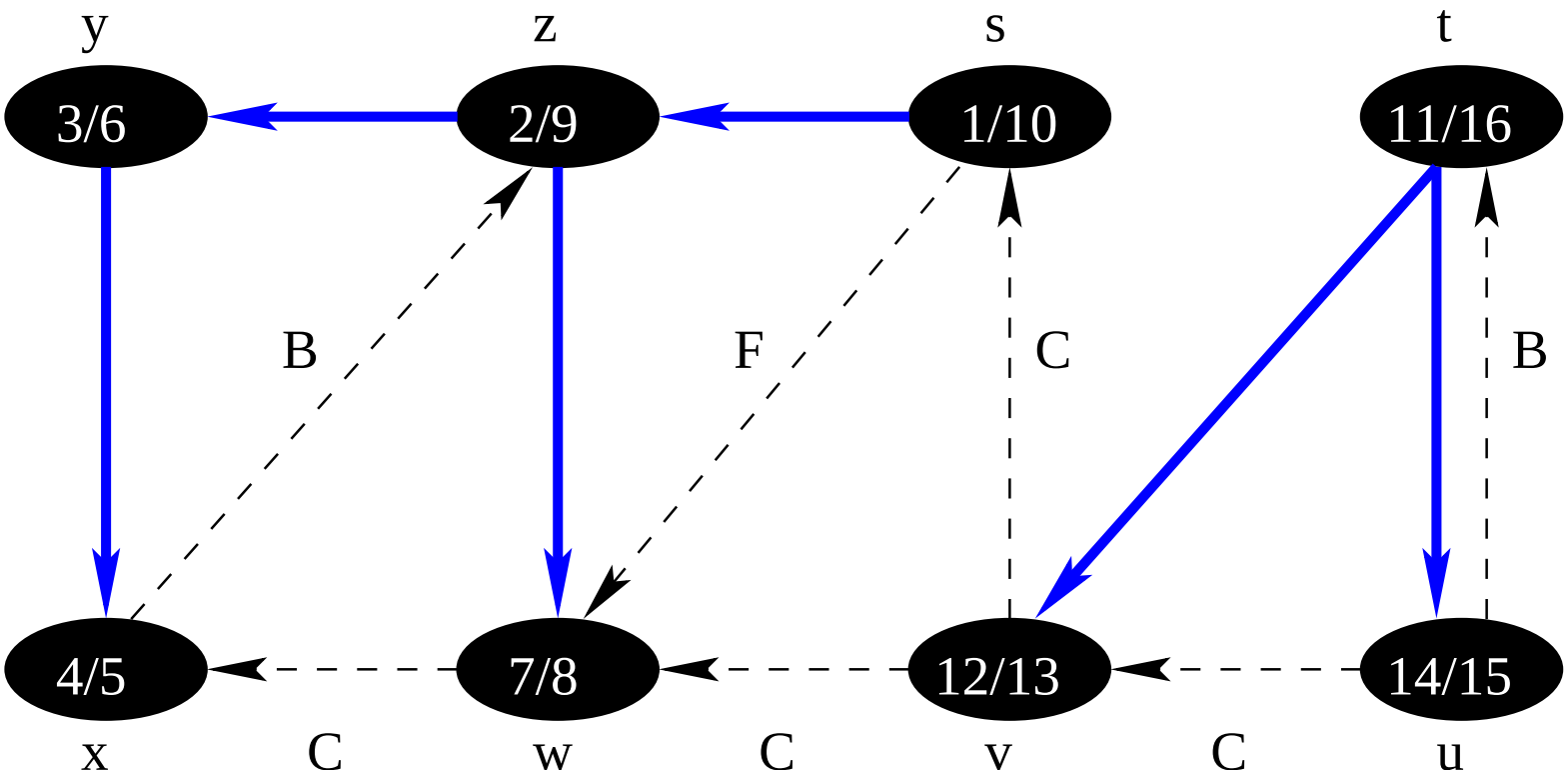
Exemplo DFS



Exemplo DFS



Exemplo DFS



Estrutura de parênteses

Corolário. (Intervalos encaixantes para descendentes)

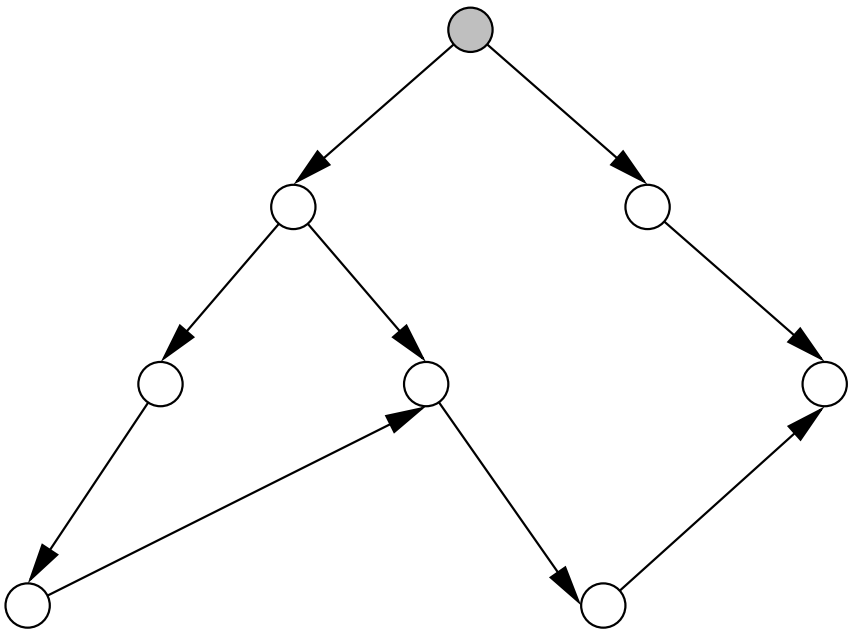
Um vértice v é um descendente próprio de u na Floresta de BP se e somente se $d[u] < d[v] < f[v] < f[u]$.

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).

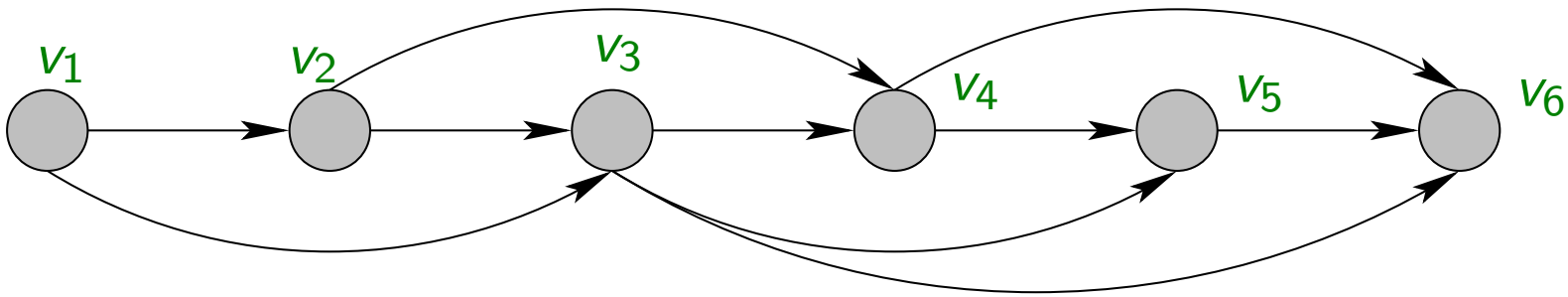


Ordenação Topológica

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

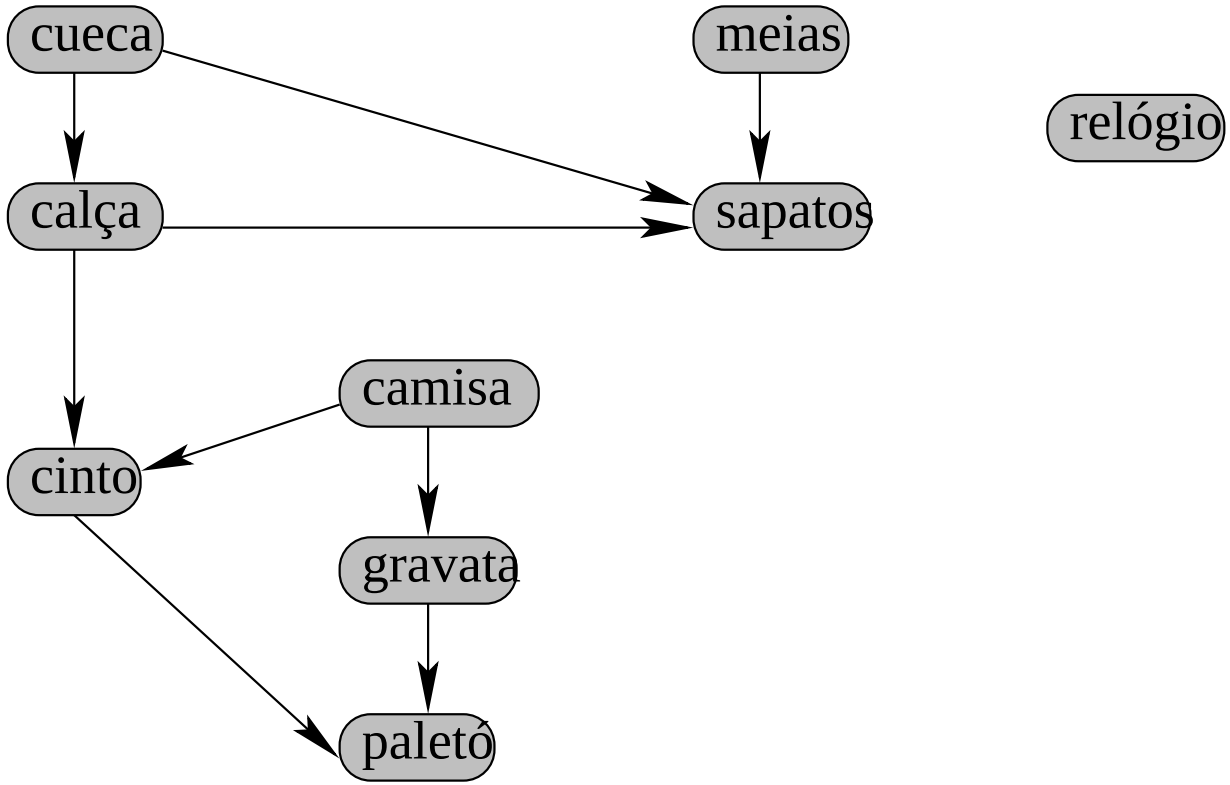
$$v_1, v_2, v_3, \ldots, v_{n-2}, v_{n-1}, v_n$$

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



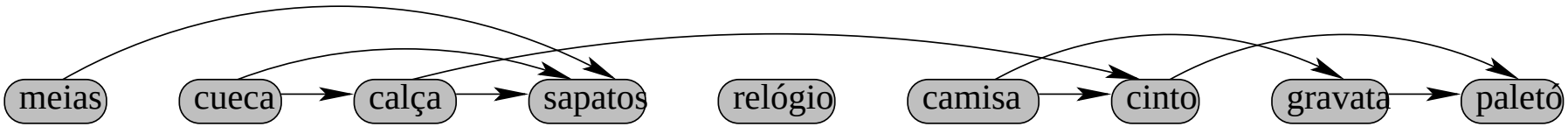
Ordenação Topológica

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



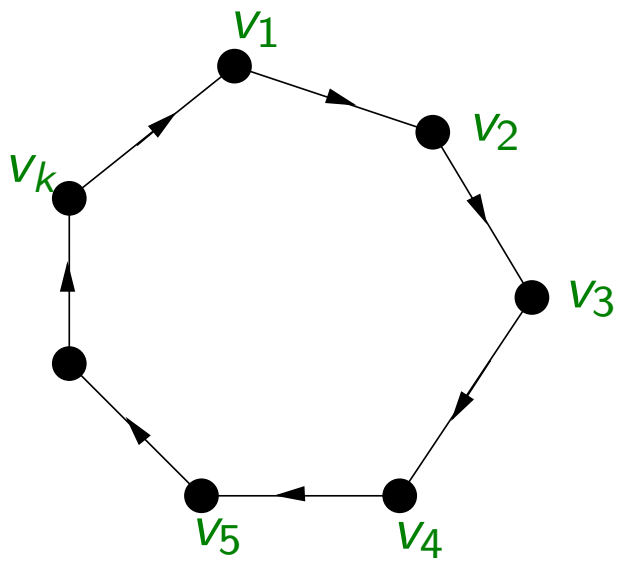
Ordenação Topológica

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



Ordenação Topológica

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



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

Grafo orientado acíclico

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

Ou seja,

- v é uma fonte se $g^-(v) = 0$ e
- v é um sorvedouro se $g^+(v) = 0$.

Grafo orientado acíclico

Lema 1. Seja G um grafo orientado tal que $g^+(v) \geq 1$ para todo $v \in V(G)$. Então G contém um ciclo.

Prova. Seja P um caminho mais longo em G e seja v seu último vértice. Então v é adjacente a algum vértice de P e portanto, existe um ciclo. ■

Lema 2. Seja G um grafo orientado tal que $g^-(v) \geq 1$ para todo $v \in V(G)$. Então G contém um ciclo. ■

Corolário. Todo grafo orientado acíclico G possui (pelo menos) um sorvedouro e uma fonte.

Ordenação topológica

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

Prova.

($\Leftarrow$) Obviamente, se G possui uma ordenação topológica então G é acíclico.

($\Rightarrow$) Usaremos indução em $n := |V|$ para mostrar que todo grafo acíclico G possui uma ordenação topológica $v_1, \ldots, v_n$.

Ordenação topológica

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 $n - 1$ vértices possui uma ordenação topológica $v_1, \ldots, v_{n-1}$.

Passo de indução: pelo Corolário, G possui um sorvedouro v_n. Então $G - v_n$ 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. ■

Ordenação topológica

Baseado na prova anterior, pode-se projetar por indução um algoritmo para obter uma ordenação topológica de um grafo orientado 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 $O(V^2)$.

Ordenação topológica

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

- Encontre um sorvedouro v_n de G. (em tempo médio $O(1)$!)
- Recursivamente encontre uma ordenação topológica $v_1, \ldots, v_{n-1}$ de $G - v_n$.
- Devolva $v_1, v_2, \ldots, v_n$.

Observação: para se obter um algoritmo linear (para qualquer problema) é necessário que

- em cada iteração, o algoritmo execute $O(1)$ instruções, ou
- o custo amortizado (custo médio) por iteração seja $O(1)$. No caso, você tem que garantir que o custo total do primeiro passo seja $O(V + E)$ ao longo da execução.

Exemplo: custo amortizado

1 **para cada** $u \in V[G]$ **faça**
2 **para cada** $v \in \mathrm{Adj}[u]$ **faça**
3 imprime v ▷ ou qualquer operação de custo $O(1)$

A soma dos comprimentos de todas as listas de adjacências é $O(E)$.

O custo total do algoritmo é $O(V + E)$.

Assim, o custo amortizado das linhas 2-3 é $O(E/V)$.

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). Em uma busca em profundidade, todos os vértices cinzas estão em um caminho e portanto v é um ancestral de u.

Mas isto significa que existe um ciclo orientado formado pelo caminho na árvore de v a u e pela aresta (u, v), o que é uma contradição. Logo, u 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

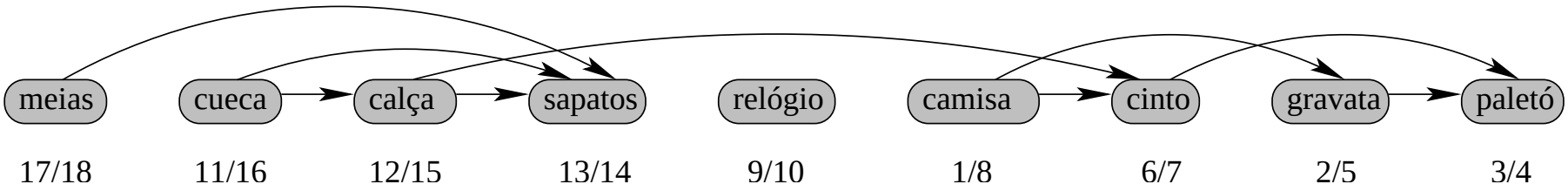
Recebe um grafo orientado acíclico G e
devolve uma ordenação topológica de G.

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.

Corretude

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

Prova:

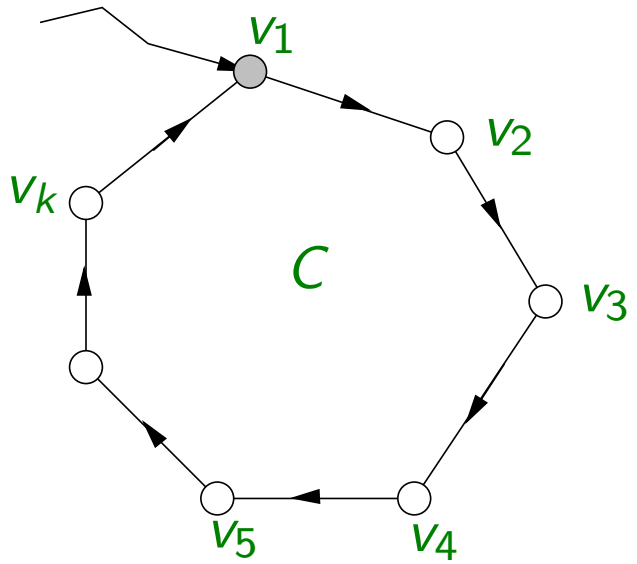
($\Leftarrow$) Suponha que (u, v) é uma aresta de retorno.

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

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

Corretude

($\Rightarrow$) Agora suponha que G contém um ciclo orientado C.



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

Corretude

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

Para mostrar que o algoritmo funciona, basta 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, (a cor de) v é branco ou preto.

Corretude

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

Portanto, TOPOLOGICAL-SORT funciona corretamente.

Contagem de caminhos

Exercício (CLRS 22.4-2). Descreva um algoritmo linear que recebe um grafo orientado 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

Problema: calcular o número de caminhos de s a t.

Ideia: combinar ordenação topológica e programação dinâmica!

Seja G um grafo orientado acíclico. Seja

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

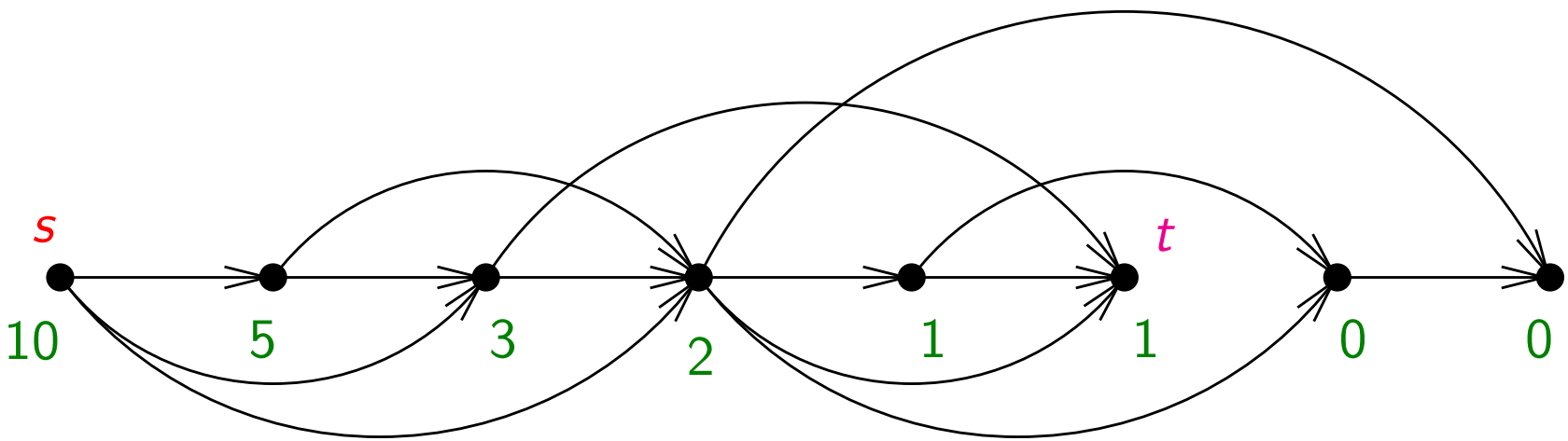
Queremos determinar $p(s)$.

Para isto, queremos achar uma recorrência para $p(v)$.

Contagem de caminhos

Seja G um grafo orientado acíclico. Seja

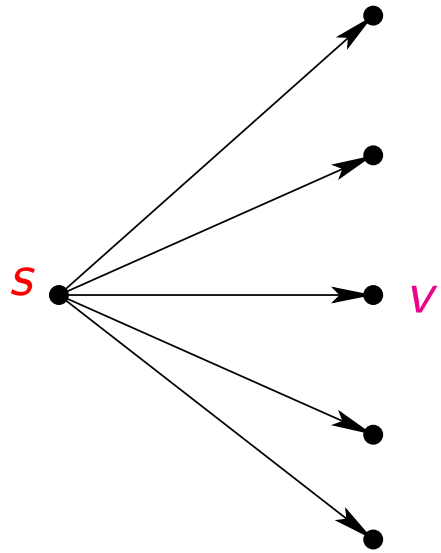
$$p(v) = \text{número de caminhos de } v \text{ a } t.$$



Queremos achar uma recorrência para $p(v)$.

Contagem de caminhos

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

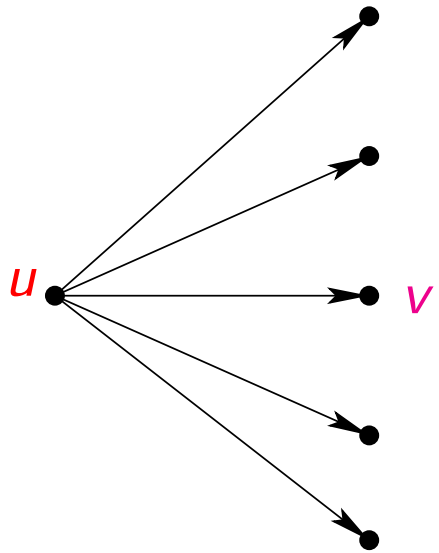


Qual é a relação entre $p(s)$ e estes valores?

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

Contagem de caminhos

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



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

Recorrência

Fixe uma ordenação topológica de G.

O valor $p(u)$ depende apenas de valores $p(v)$ onde v sucede u.

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

Note que se u é um sorvedouro distinto de t, então a terceira regra diz que $p(u) = 0$.

Algoritmo

CONTA-CAMINHOS(G, s, t) $\quad \triangleright$ G é acíclico

1. **para cada** $v \in V[G] - \{s\}$ **faça**
2. $\quad p[v] \leftarrow 0$
3. $p[t] \leftarrow 1$
4. compute uma ordenação topológica $v_1, v_2, \ldots, v_n$ de G
5. **para** $i \leftarrow n-1$ **até** 1 **faça** $\triangleright$ em ordem inversa
6. $\quad$ **para cada** $v \in \mathrm{Adj}[v_i]$ **faça**
7. $\quad\quad p[v_i] \leftarrow p[v_i] + p[v]$
8. **devolva** p

Complexidade: $O(V + E)$

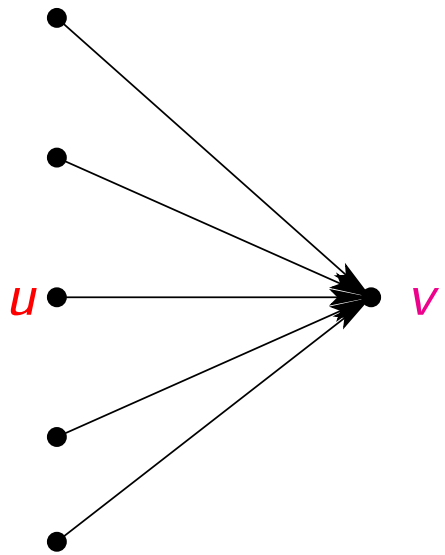
Para ver a corretude basta provar que no início da linha 5 vale o seguinte invariante: $p[v_i] = p(v_i)$.

Outra forma de contar os caminhos

Seja G um grafo orientado 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

CONTA-CAMINHOS(G, s, t) $\quad \triangleright\ G$ é acíclico

1. **para cada** $v \in V[G] - \{s\}$ **faça**
2. $\quad q[v] \leftarrow 0$
3. $q[s] \leftarrow 1$
4. compute uma ordenação topológica $v_1, v_2, \ldots, v_n$ de G
5. **para** $i \leftarrow 1$ **até** $n - 1$ **faça**
6. $\quad$ **para cada** $v \in \mathrm{Adj}[v_i]$ **faça**
7. $\quad\quad q[v] \leftarrow q[v] + q[v_i]$
8. **devolva** q

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

Grafos orientados acíclicos vs ordenação topológica

Tentar usar ordenação topológica quando o grafo orientado for acíclico!
Tenho que me lembrar disso. . .

Dito isso, há problemas envolvendo grafos orientados acíclicos para os quais não é suficiente usar (apenas) ordenação topológica.