Projeto e Análise de Algoritmos

Buscas em grafos

Lehilton Pedrosa

Primeiro Semestre de 2020

Buscas em grafo

Buscas em grafos

Como percorrer os vértices de um grafo?

- mais complicado que lista, vetor, árvore binária
- podem ser direcionados ou não direcionados
- queremos descobrir informações sobre sua estrutura
- podemos pensar em cada componente separadamente
- **objetivo:** encontrar uma árvore geradora

Dois algoritmos

1. Busca em largura
   - diremos apenas BFS
   - do inglês breadth-first search
2. Busca em profundidade
   - diremos apenas DFS
   - do inglês depth-first search

Representação de árvores

Como representar uma árvore de busca?

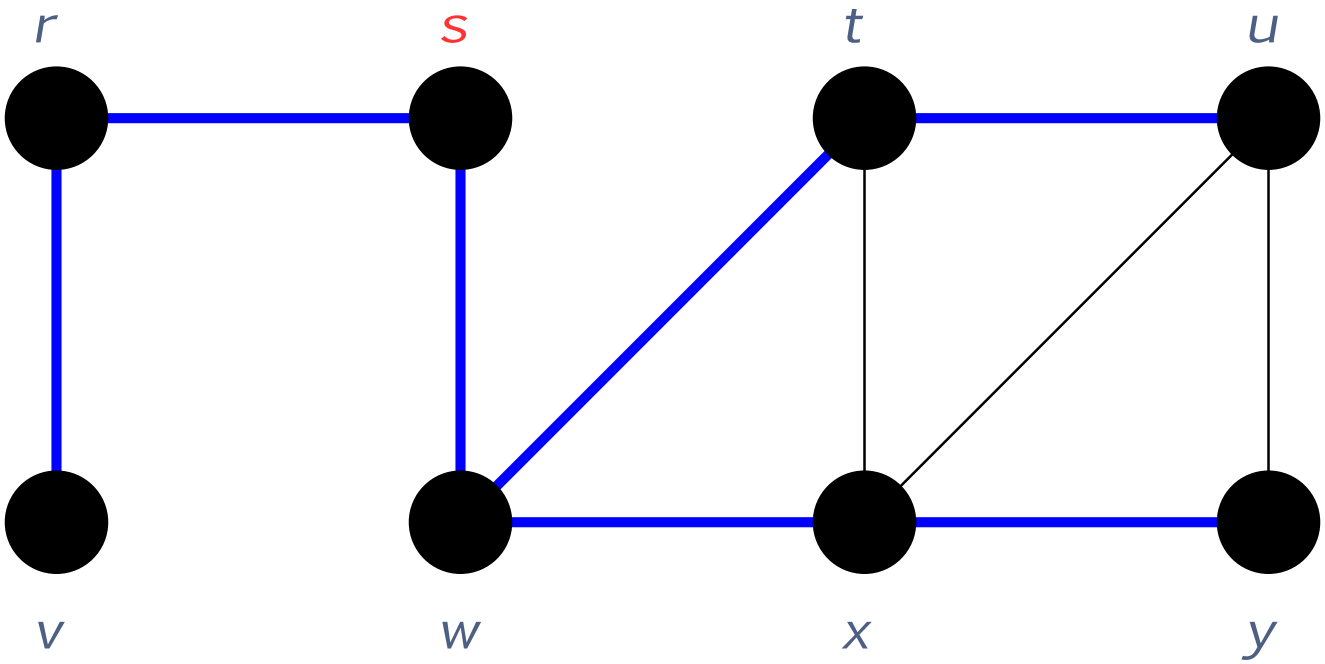
- vamos enraizá-la em um vértice de origem s
- representar a árvore com um vetor π de pais
- o pai de um vértice v é $\pi[v]$
- convencionamos que $\pi[s] = \text{NIL}$

Algumas propriedades

- existe aresta de $\pi[v]$ até v
- o caminho de s a v na árvore é

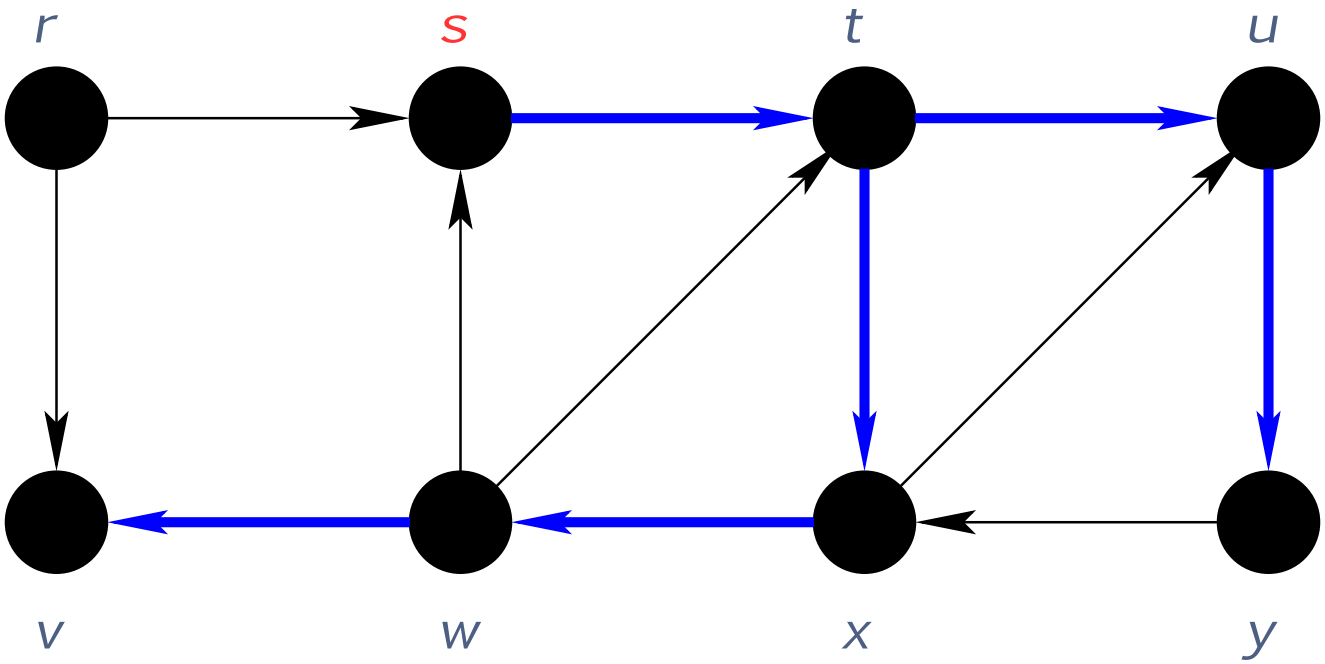
$$s \to \cdots \to \pi[\pi[\pi[v]]] \to \pi[\pi[v]] \to \pi[v] \to v$$

Exemplo com grafo não direcionado



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

Exemplo com grafo direcionado



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

Caminho da árvore

```
PRINT-PATH(G, s, v)
1  se v = s então
2      imprima s
3  senão 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
```

- imprime o caminho de s a v na árvore de raiz s
- gasta tempo linear no tamanho desse caminho

Busca em largura

Distância entre vértices

Vértices alcançáveis

- alcançamos v a partir de s se há caminho de s a v
- pode haver diversos caminhos entre s a v
- queremos algum com o menor comprimento

A distância de s a v é o comprimento de um caminho mais curto de s a v

- denotamos este valor por $\text{dist}(s, v)$
- se v não for alcançável, definimos $\text{dist}(s, v) = \infty$

Busca em largura

Buscando os vértices alcançáveis em largura

- primeiro o vértice de origem
- depois os vizinhos do vértice de origem
- depois os vizinhos dos vizinhos do vértice de origem
- etc.

Descobrindo a distância

- um produto da busca são as distâncias à origem
- á árvore de busca fornece um caminho mais curto

Construindo uma árvore de busca

Ideia do algoritmo

- percorremos os vértices usando uma fila Q
- começamos com o vértice de origem s
- para cada vizinho v do vértice atual u
  - adicionamos uma aresta (u, v) à árvore de busca
  - inserimos v na fila de processamento
- repetimos com o primeiro vértice da fila

Cores dos vértices

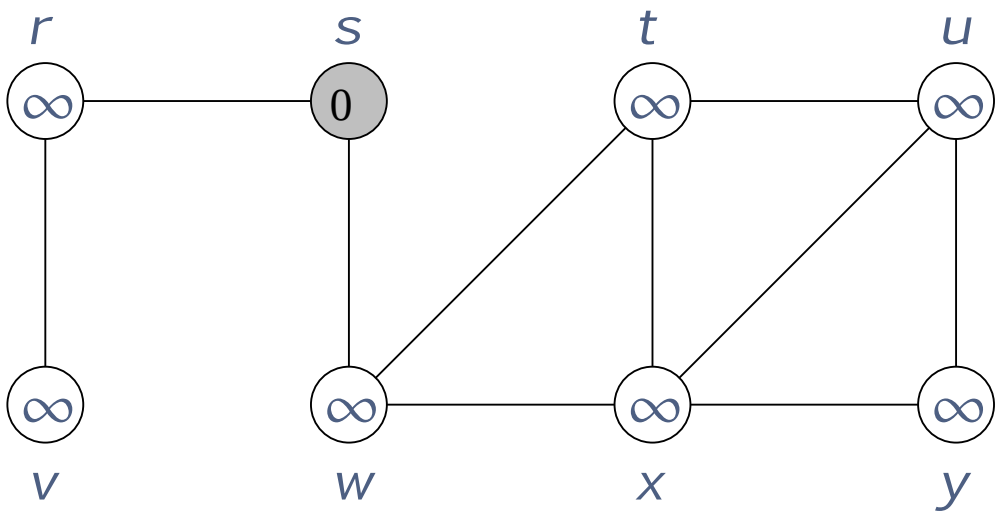
Vamos pintar o grafo durante a busca

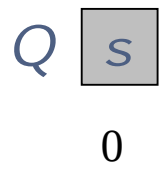
1. $cor[v] =$ branco se não descobrimos v ainda
2. $cor[v] =$ cinza se já descobrimos, mas não finalizamos v
3. $cor[v] =$ preto se já descobrimos e já finalizamos v

Observações

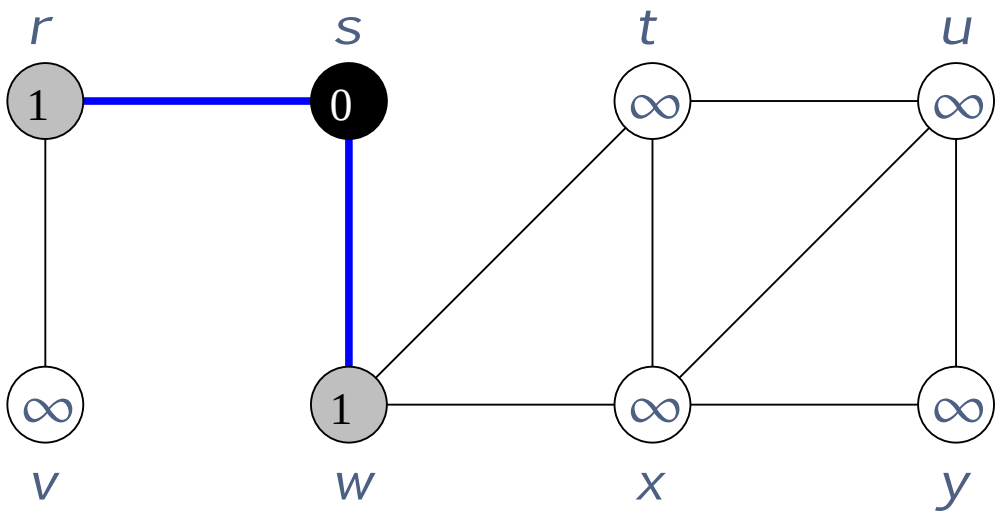
- não é necessário em uma implementação
- mas facilita o entendimento do algoritmo

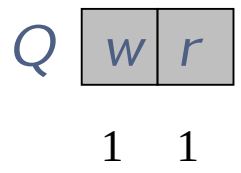
Exemplo de busca em largura



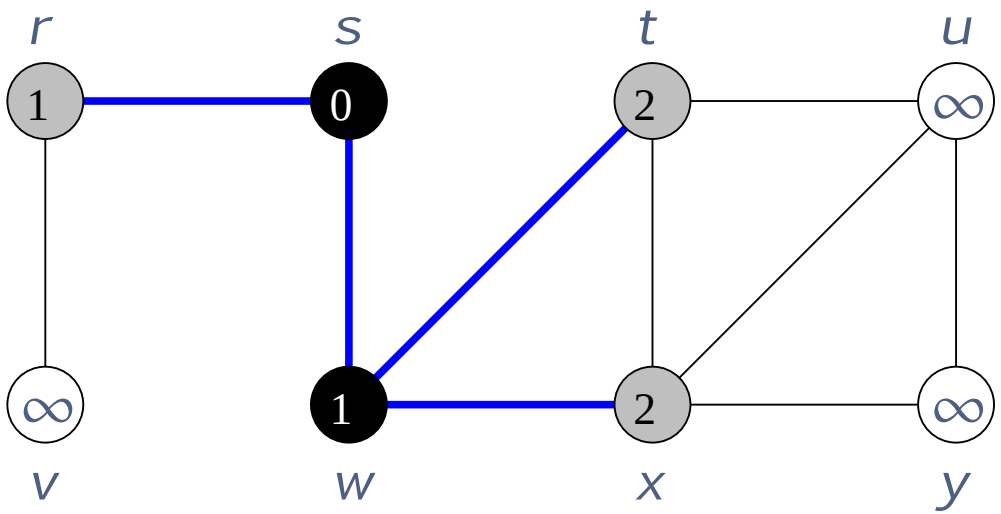


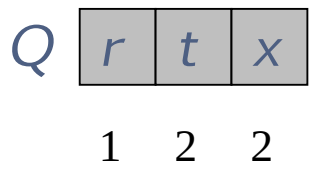
Exemplo de busca em largura



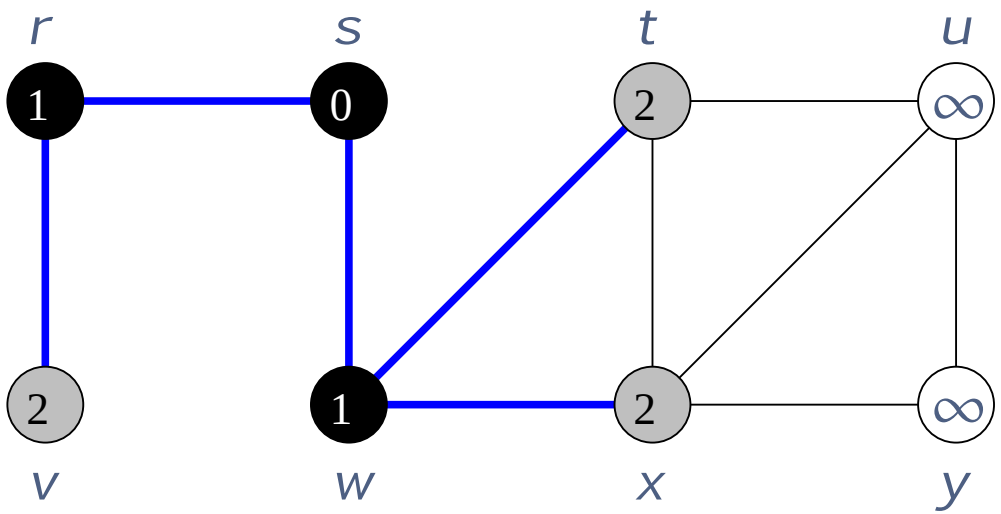


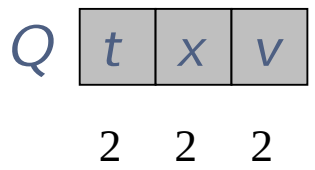
Exemplo de busca em largura



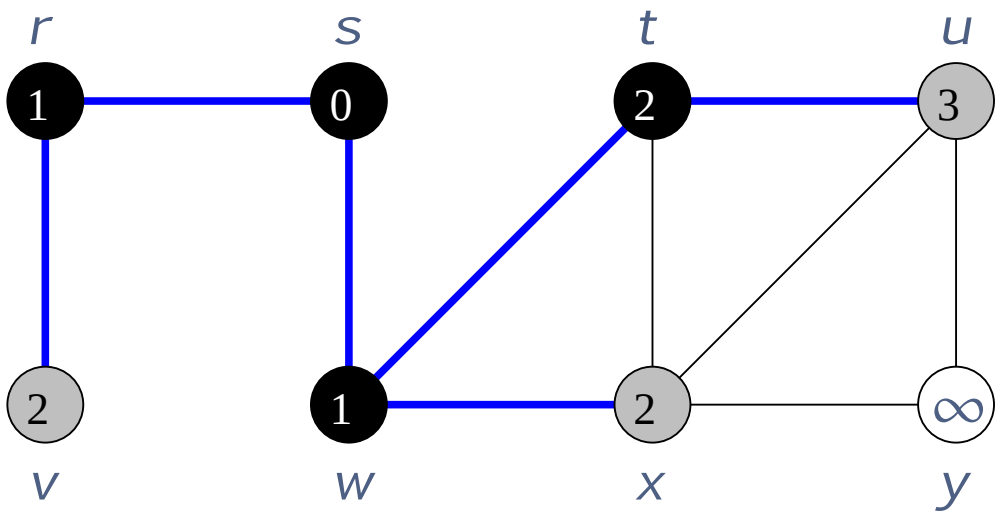


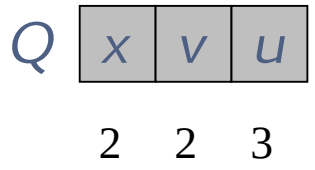
Exemplo de busca em largura



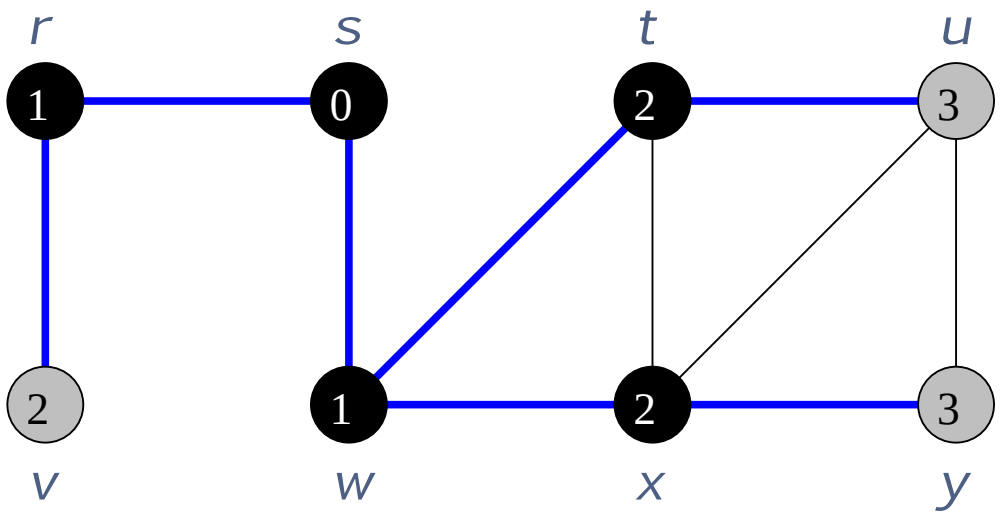


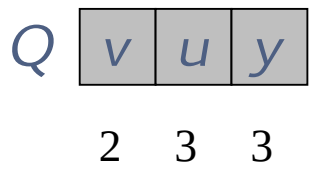
Exemplo de busca em largura



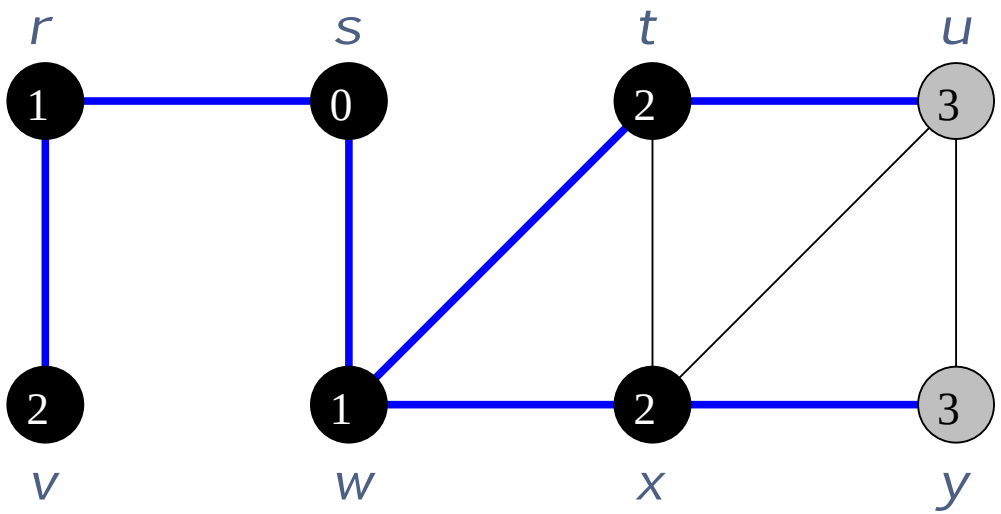


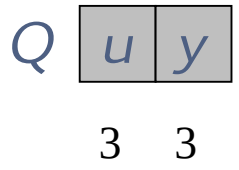
Exemplo de busca em largura



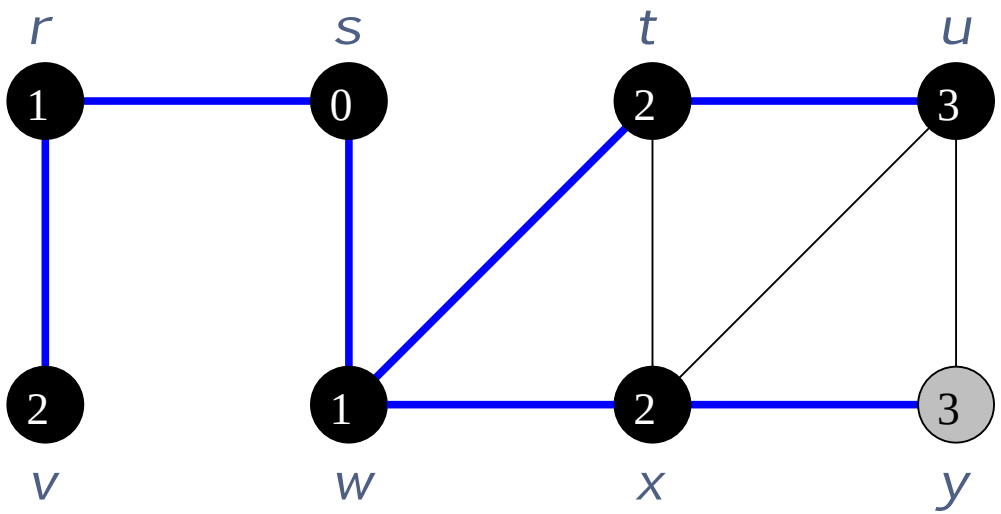


Exemplo de busca em largura





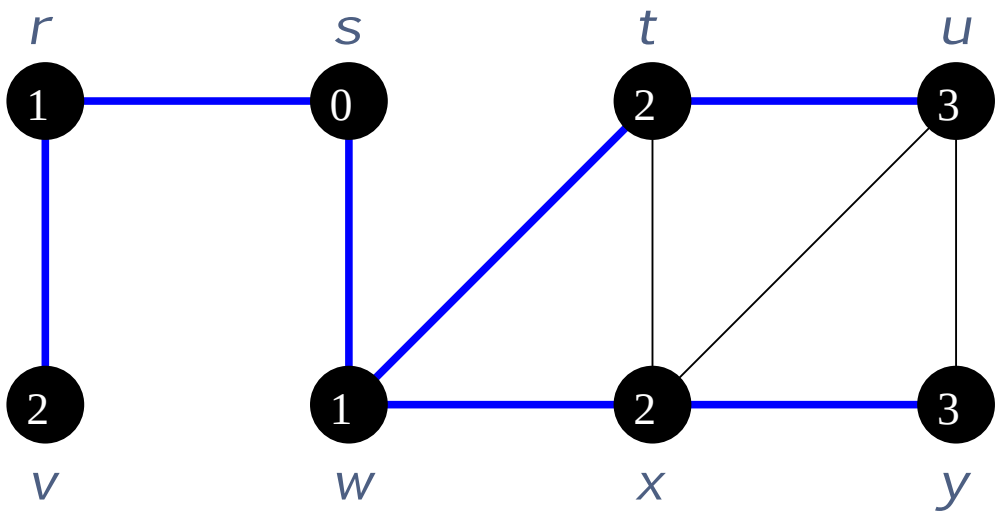
Exemplo de busca em largura



Q | y

3

Exemplo de busca em largura



Q $\emptyset$

Algoritmo BFS

```
BFS(G, s)
 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
```

- representamos G com listas de adjacências
- a árvore de busca em largura é representada por π
- calcula a distância $d[v]$ de s a v

Algoritmo BFS (cont)

```
 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
```

Análise de complexidade

Analisamos de forma agregada

1. o tempo de inicialização é $O(V)$
2. um vértice não volta a ser branco
   - enfileiramos cada vértice no máximo uma vez
   - desenfileiramos cada vértice no máximo uma vez
   - cada operação na fila leva tempo $O(1)$
   - o tempo gasto com a fila é $O(V)$
3. processamos cada vértice uma vez
   - cada lista de adjacências é percorrida uma vez
   - no pior caso, percorremos todas as listas
   - o tempo gasto percorrendo adjacências é $O(E)$

A complexidade da busca em largura é $O(V + E)$

Correção do algoritmo

Teorema

Seja $G = (V, E)$ um grafo e s um vértice de G. Então, depois de executar $\mathrm{BFS}(G, s)$, temos

1. π define uma árvore enraizada em s,
2. $d[v] = \mathrm{dist}(s, v)$ para todo $v \in V[G]$.

Precisamos de dois lemas

- Lema 1: o caminho de s a v na árvore tem tamanho $d[v]$
- Lema 2: a fila Q respeita a ordem de $d[v]$

Lema 1

Lema 1

Seja T a árvore induzida por π. Se $d[v] < \infty$, então

1. v é um vértice de T,
2. o caminho de s a v em T tem comprimento $d[v]$.

Demonstração:

- por indução no número de vezes que executamos ENQUEUE
- depois que executamos ENQUEUE pela primeira vez
  - T continha apenas s e valia $d[s] = 0$
  - como $d[s]$ nunca mais muda, isso completa a base

Prova do lema

Considere o instante em que enfileiramos v

- então v foi descoberto percorrendo os vizinhos de u
- mas u já havia sido enfileirado antes desse instante
- pela hipótese de indução
  1. existe um caminho de s a u em T
  2. esse caminho tem comprimento $d[u]$
- mais isso implica que
  1. há caminho de s a v em T, pois $\pi[v] = u$
  2. e esse caminho tem comprimento $d[v]$, pois $d[v] = d[u] + 1$
- e completamos a indução

Corolário 1

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

Lema 2

Lema 2

Suponha que $\langle v_1, v_2, \ldots, v_r \rangle$ seja a disposição da fila Q em alguma iteração do algoritmo. Então

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

Demonstração:

- por indução no número de iterações
- antes da primeira iteração, $Q = \langle s \rangle$ e o lema vale

Prova do lema

Considere a execução do laço

- no início da iteração a fila era $\langle v_1, v_2, \ldots, v_r \rangle$
- na iteração, removemos v_1 e inserimos $v_{r+1}, \ldots, v_{r+t}$
- no final da iteração a fila será $\langle v_2, \ldots, v_r, v_{r+1}, \ldots, v_{r+t} \rangle$

Inserimos vizinhos de v_1

- se v_j é um vértice inserido, então $d[v_j] = d[v_1] + 1$
- pela hipótese de indução

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

- portanto

$$d[v_2] \leq \cdots \leq d[v_r] \leq d[v_{r+1}] \leq \cdots \leq d[v_{r+t}] \leq d[v_2] + 1$$

Prova do teorema

Teorema

Seja $G = (V, E)$ um grafo e s um vértice de G. Então, depois de executar $\mathrm{BFS}(G, s)$, temos

1. π define uma árvore enraizada em s,
2. $d[v] = \mathrm{dist}(s, v)$ para todo $v \in V[G]$.

Demonstração

- note que π define uma árvore enraizada em s (por quê?)
- também, se $\mathrm{dist}(s, v) = \infty$, então $d[v] = \infty$ pelo Corolário 1
- resta provar que se $\mathrm{dist}(s, v) < \infty$, então $d[v] = \mathrm{dist}(s, v)$

Prova do teorema (cont)

Considere um vértice v com $\text{dist}(s, v) = k$

- iremos provar que $d[v] = k$ por indução em k
- se $k = 0$, devemos ter $v = s$ e a afirmação vale
- considere então o caso em que $k \geq 1$

Suponha que $d[u] = \text{dist}(s, u)$ para todo u com $\text{dist}(s, u) < k$

- considere um caminho de s a v de comprimento k
- chame de u o vértice que antecede v nesse caminho
- daí $\text{dist}(s, u) = k - 1$ e portanto $d[u] = k - 1$

Prova do teorema (cont)

Considere o instante em que u foi removido de Q

- suponha por contradição que v seja preto
- daí v foi removido de Q antes de u
- então o Lema 2 implica que $d[v] \leq d[u] < k$
- mas o Corolário 1 implica que $k = \text{dist}(s, v) \leq d[v]$
- isso é uma contradição, então v não pode ser preto

Portanto, nesse instante, v era branco ou cinza

- se v era branco
  - v será inserido na fila nessa iteração
  - e teremos $d[v] = d[u] + 1 = k$
- se v era cinza
  - v já estava na fila nesse instante
  - então o Lema 2 implica $d[v] \leq d[u] + 1 = k$
  - como $k \leq d[v]$, temos $d[v] = k$
- em qualquer caso, concluímos a indução

Busca em profundidade

Busca em profundidade

Buscando os vértices alcançáveis em profundidade

- começamos com o vértice de origem
- depois, todos os alcançáveis pelo primeiro vizinho
- depois, todos os alcançáveis pelo segundo vizinho
- etc.

É a estratégia de vários algoritmos

- identificar as componentes conexas
- encontrar uma ordenação topológica

Construindo uma árvore de busca

Ideia do algoritmo

- começamos com o vértice de origem s
- para cada vizinho não visitado v do vértice atual
  1. adicionamos uma aresta (u, v) à árvore de busca
  2. visitamos recursivamente a partir de v

Alternativa

Ideia alternativa

- percorremos os vértices usando uma pilha S
- começamos com o vértice de origem s
- para cada vizinho v do vértice atual u
  - adicionamos uma aresta (u, v) à árvore de busca
  - inserimos v na pilha de processamento
- repetimos com o vértice do topo da pilha

Observação

- pode levar a uma árvore de busca distinta
- compare com fila da a busca em largura

Floresta de busca

Visitando todos os vértices

- a árvore de busca contém só vértices alcançáveis de s
- algumas vezes queremos visitar todos os vértices
- repetimos o processo com os vértices não visitados
- obteremos uma floresta de busca

Representando uma floresta

- também utilizamos um vetor de pais π
- um vértice v com $\pi[v] = \text{NIL}$ é raiz de uma árvore de busca
- as arestas da floresta são

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

Cores dos vértices

De novo, vamos pintar o grafo durante a busca

1. $cor[v] = \text{branco}$ se não descobrimos v ainda
2. $cor[v] = \text{cinza}$ se já descobrimos, mas não finalizamos v
3. $cor[v] = \text{preto}$ se já descobrimos e já finalizamos v

Observações

- os vértices cinza têm suas chamadas recursivas ativas
- a pilha de chamadas induz um caminho na floresta

Tempo de descoberta e finalização

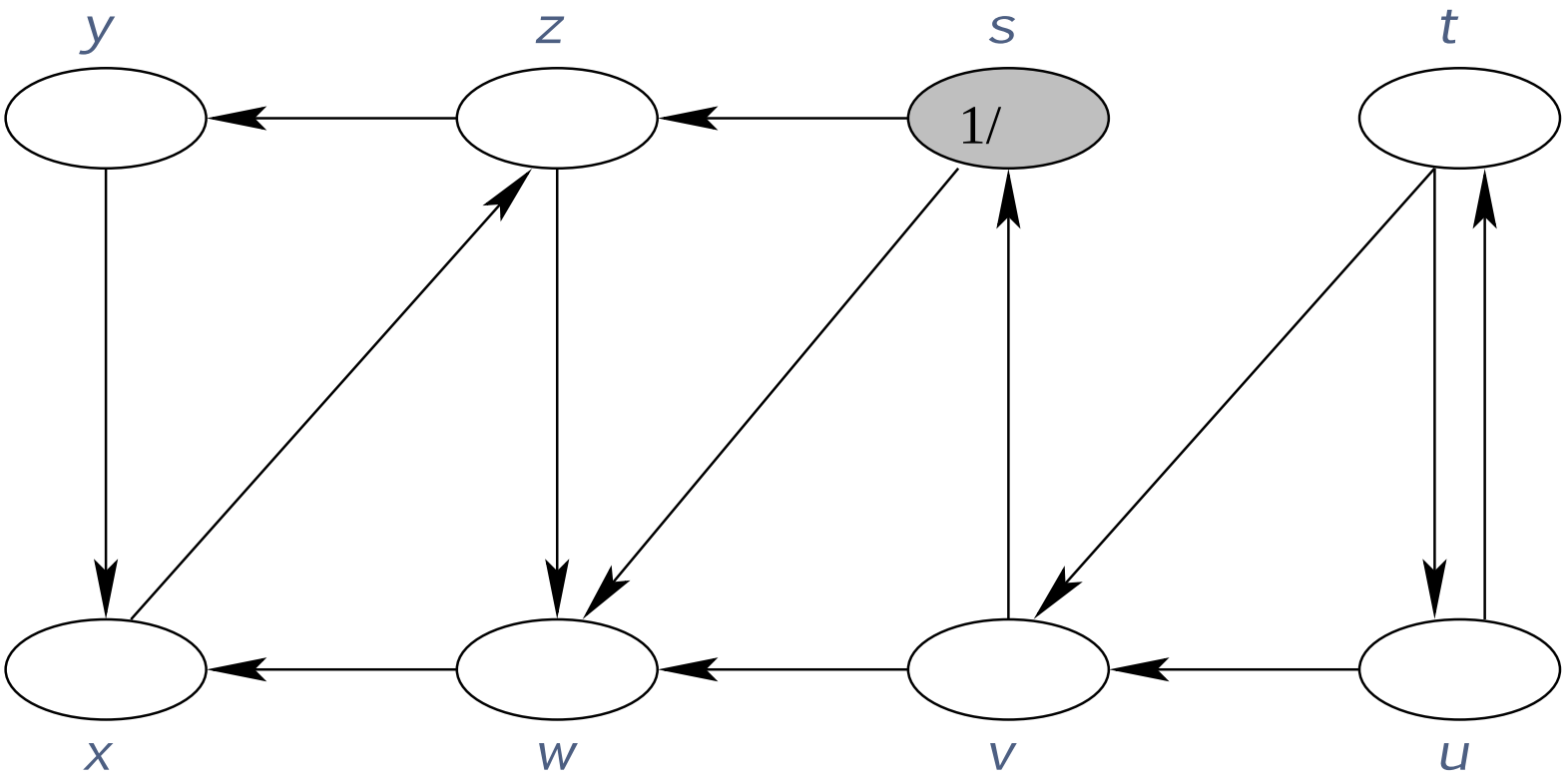
Vamos associar rótulos aos vértices

- $d[v]$ é instante de descoberta de v
- $f[v]$ é instante de finalização de v

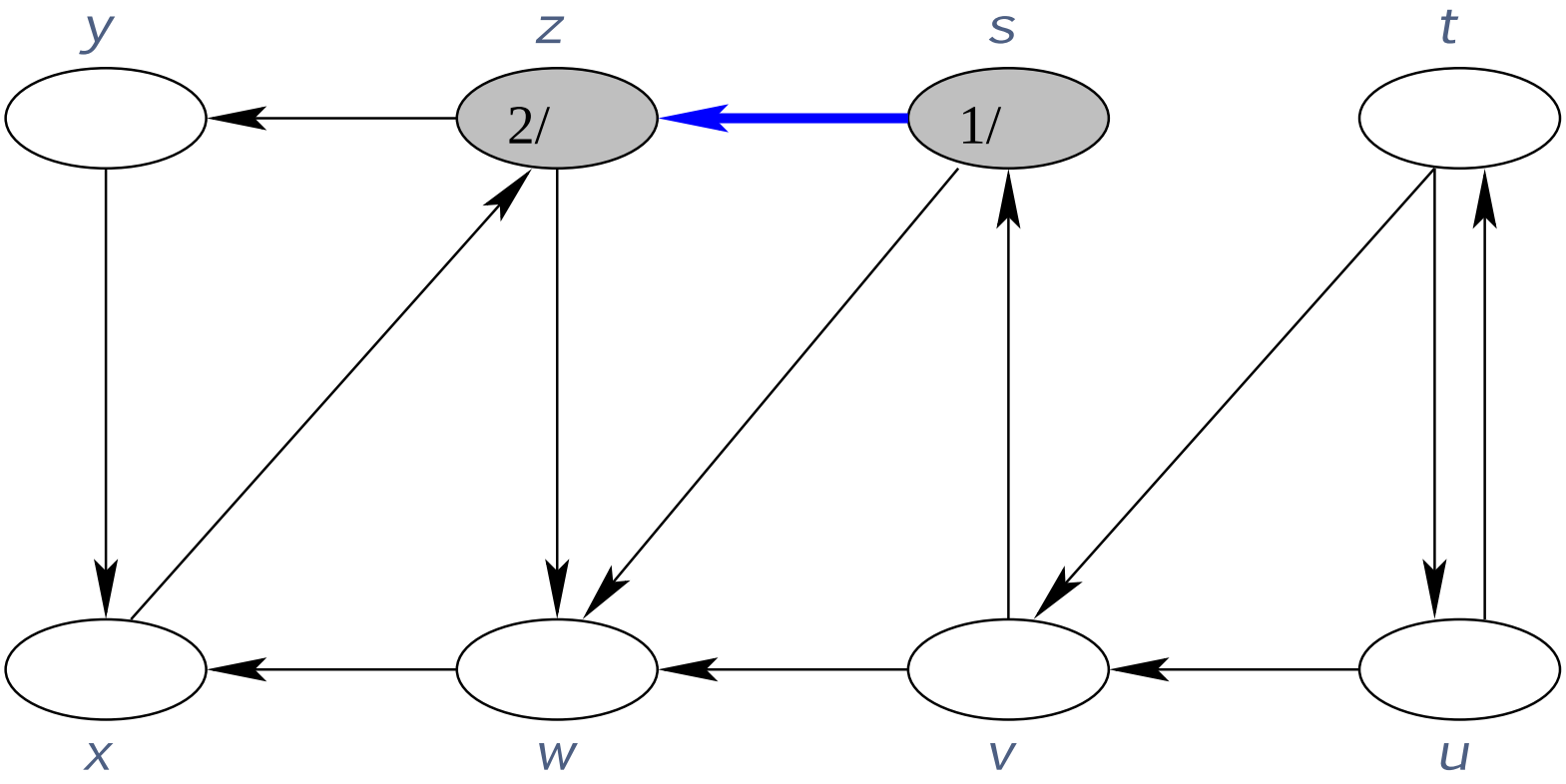
Observações

- os rótulos são inteiros distintos entre 1 e $2|V|$
- refletem os instantes em que v muda de cor

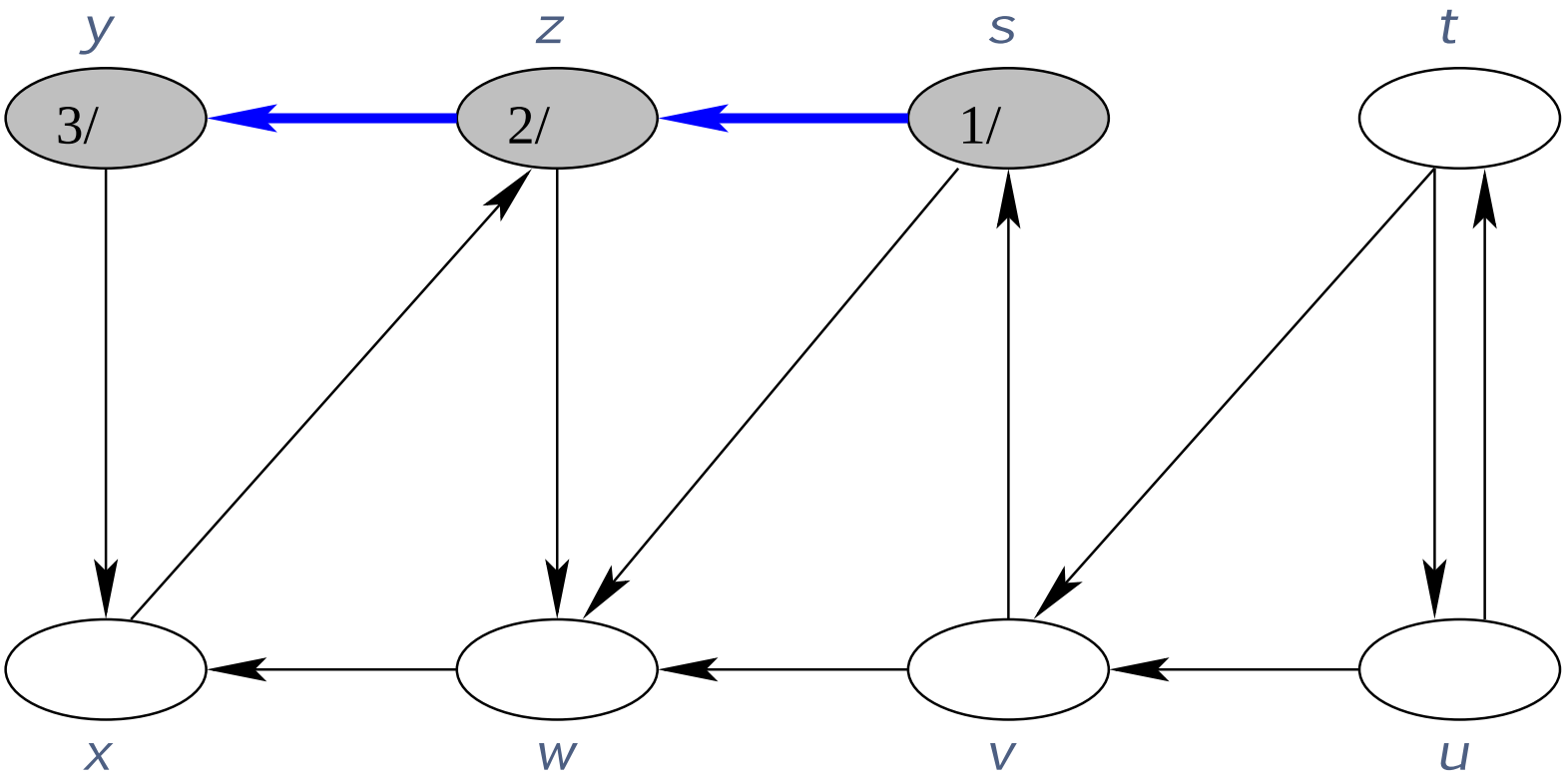
Exemplo de busca em profundidade



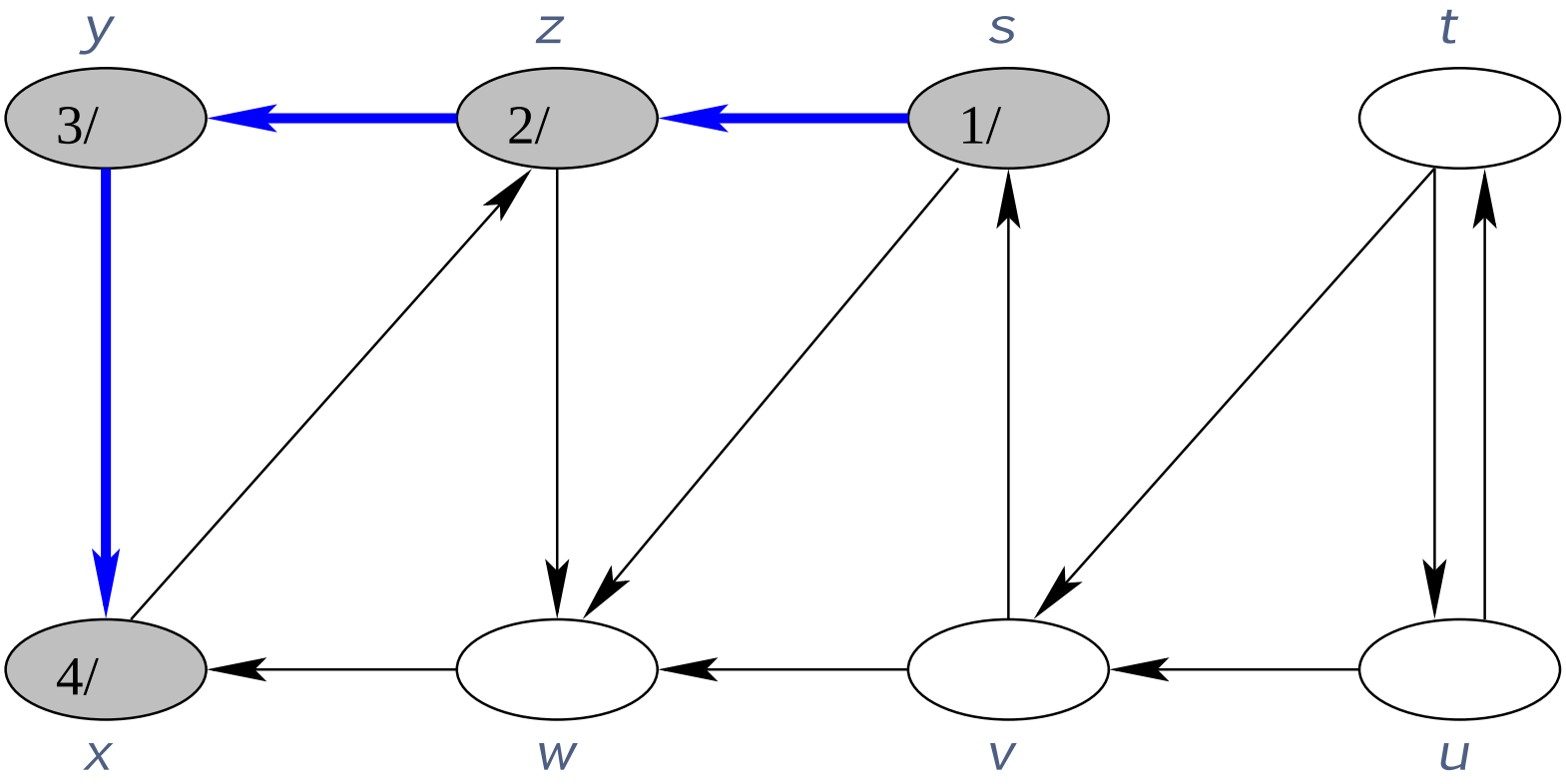
Exemplo de busca em profundidade



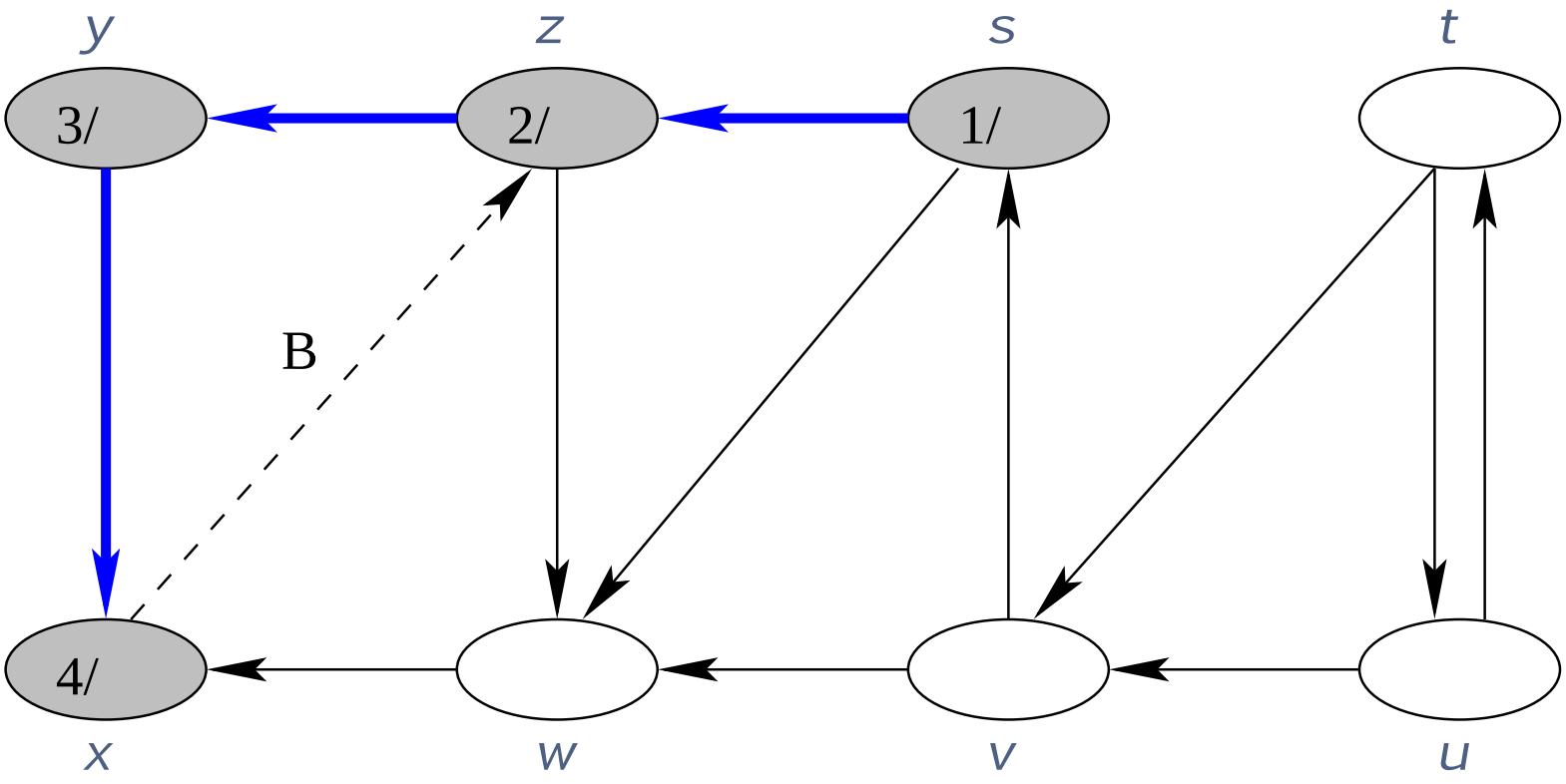
Exemplo de busca em profundidade



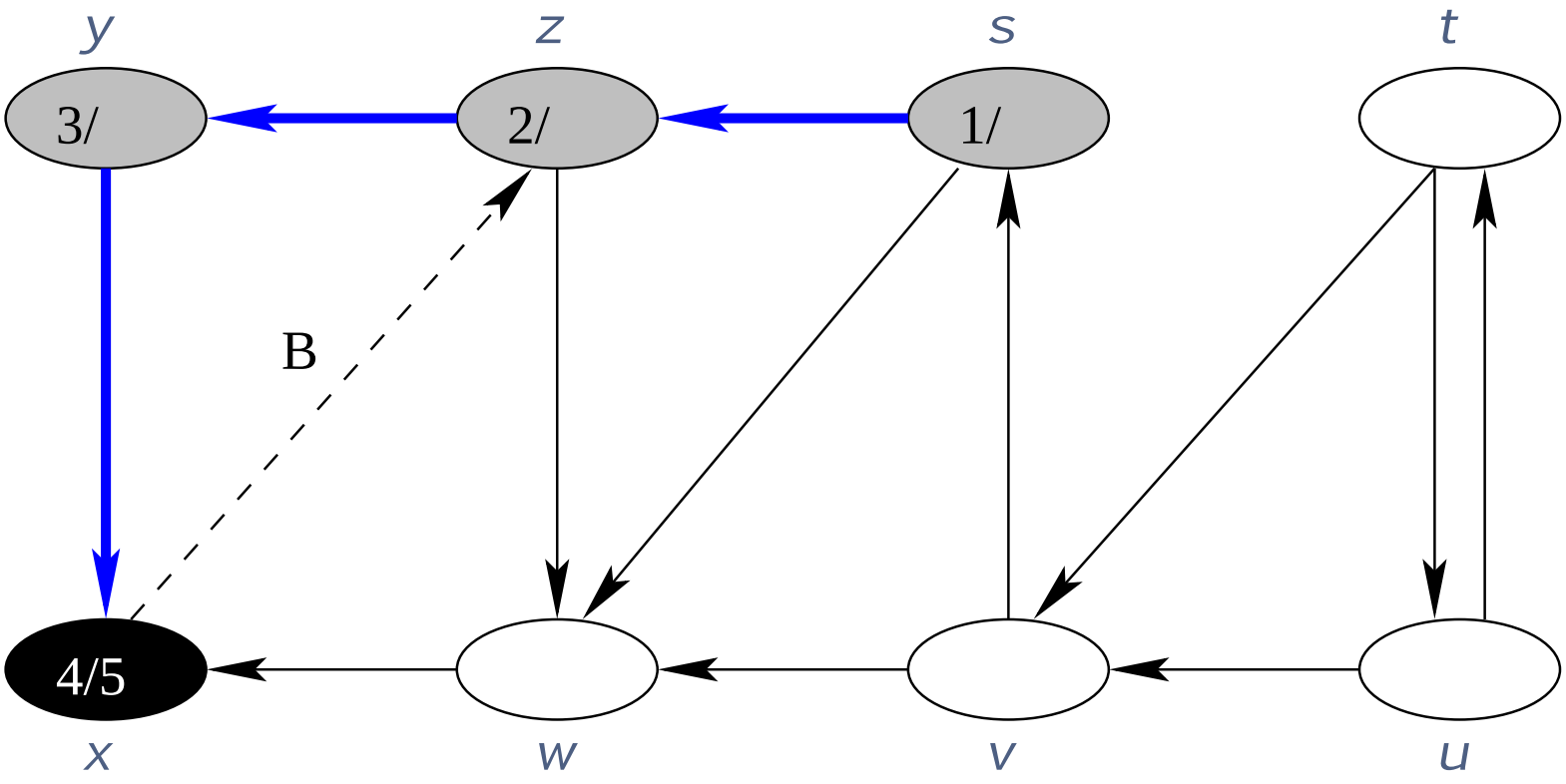
Exemplo de busca em profundidade



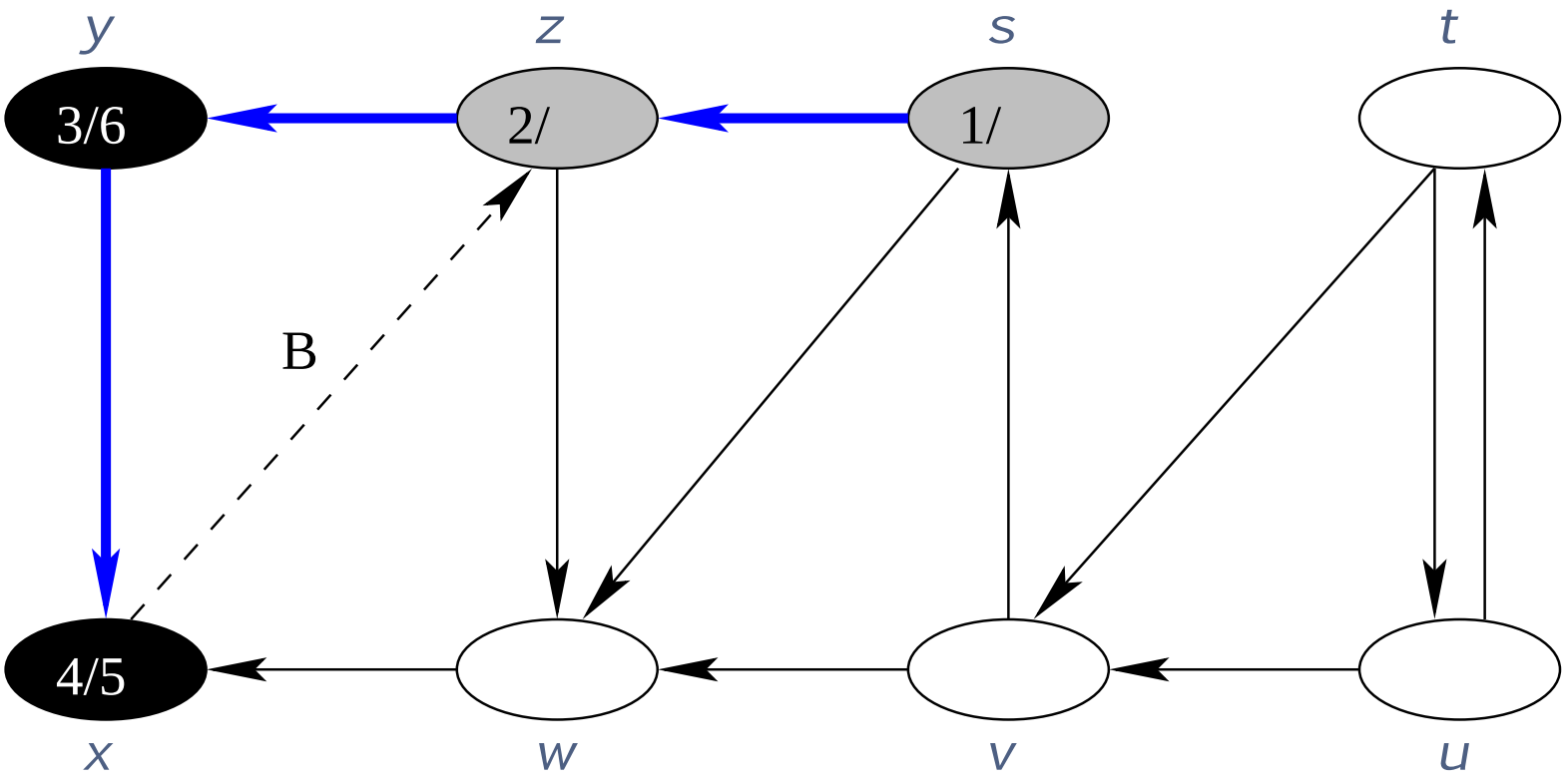
Exemplo de busca em profundidade



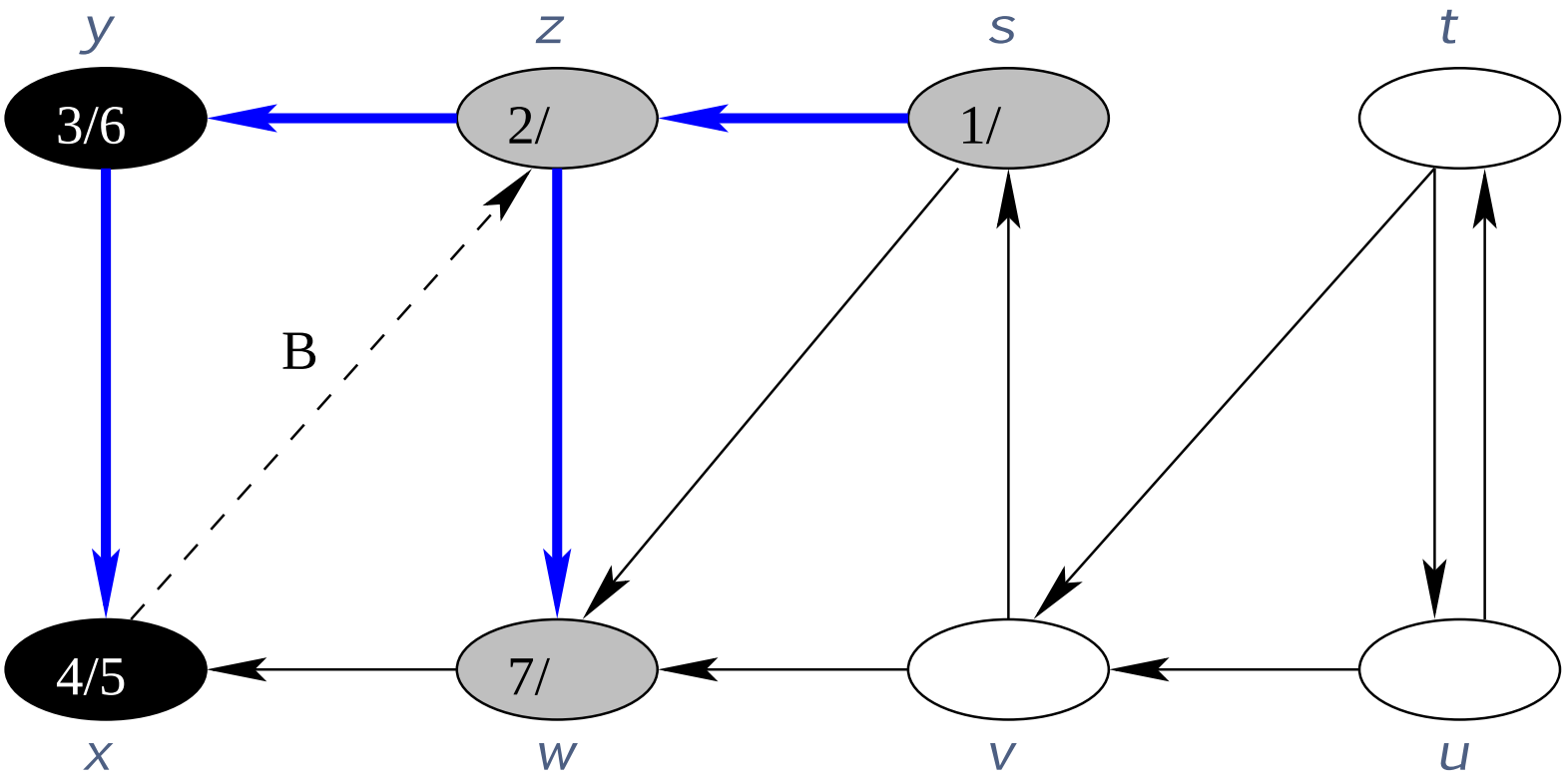
Exemplo de busca em profundidade



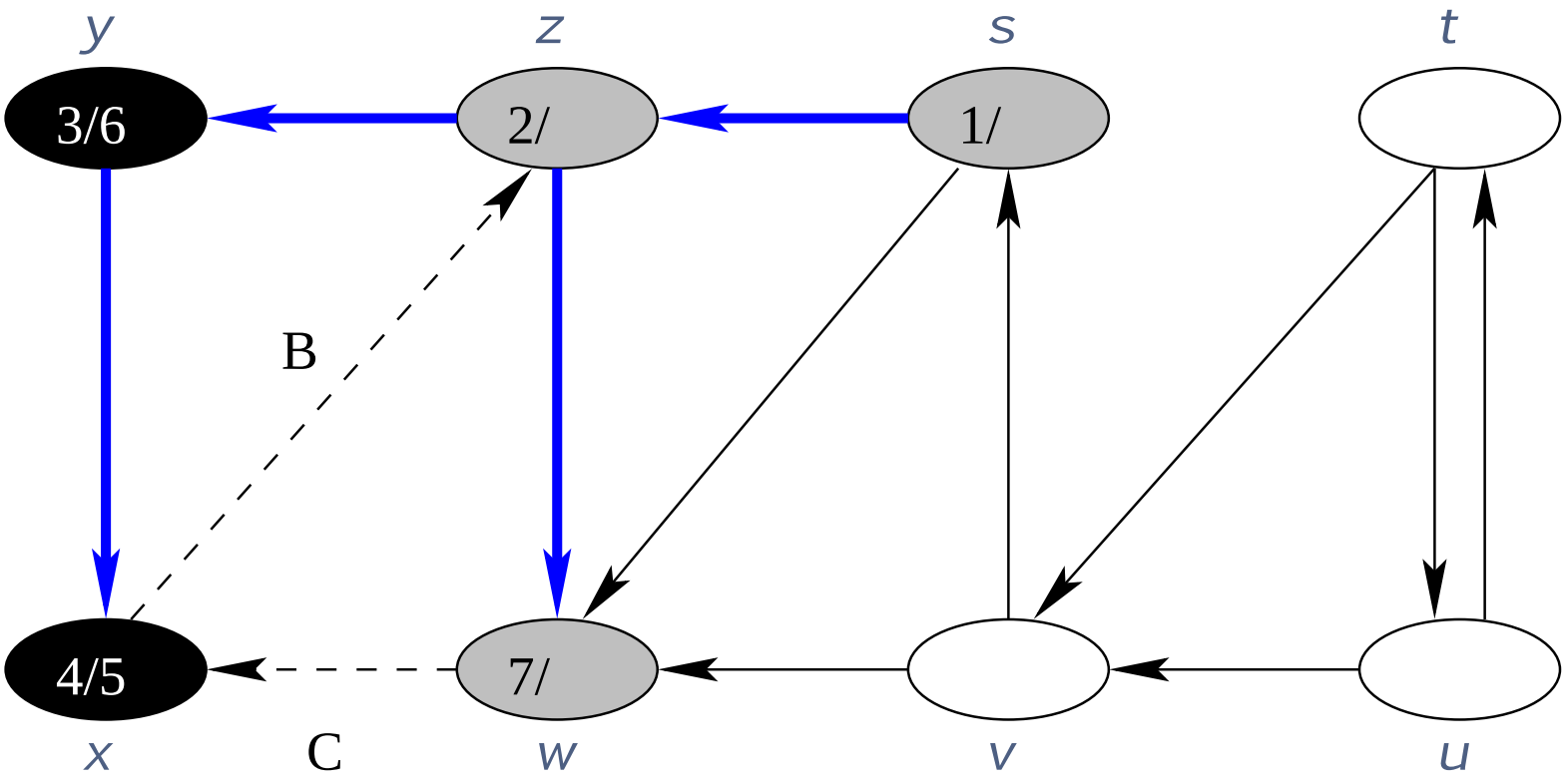
Exemplo de busca em profundidade



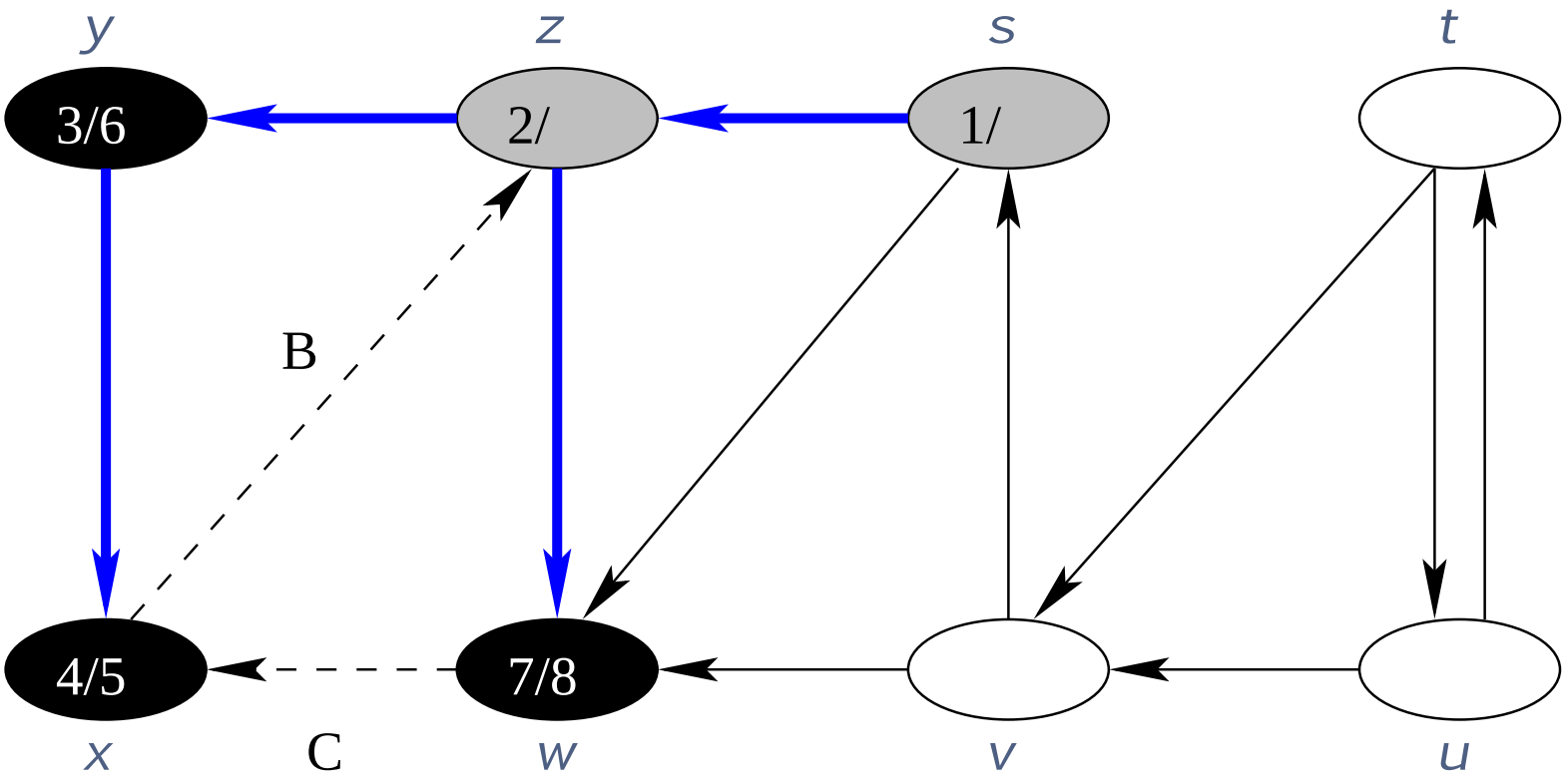
Exemplo de busca em profundidade



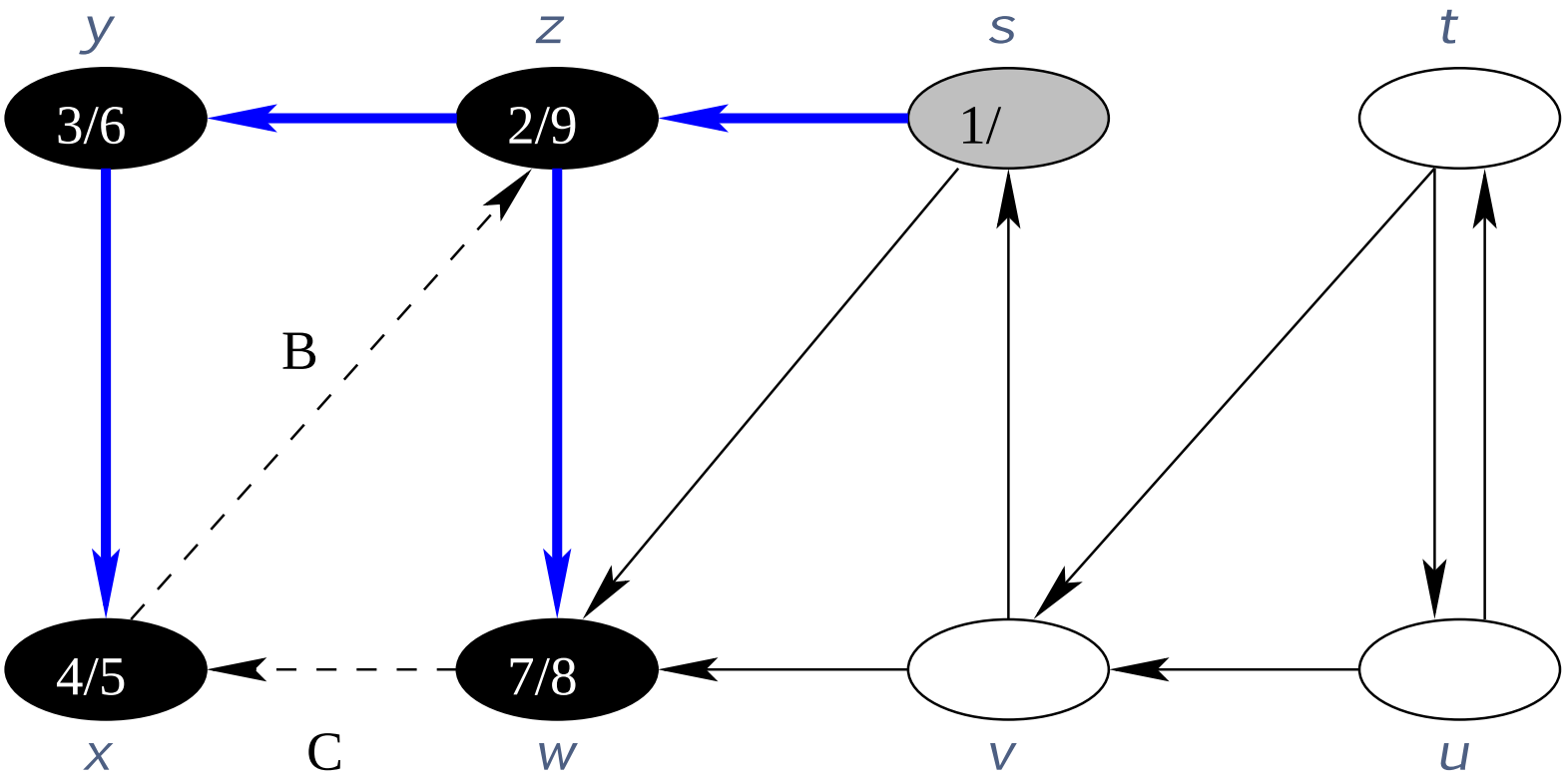
Exemplo de busca em profundidade



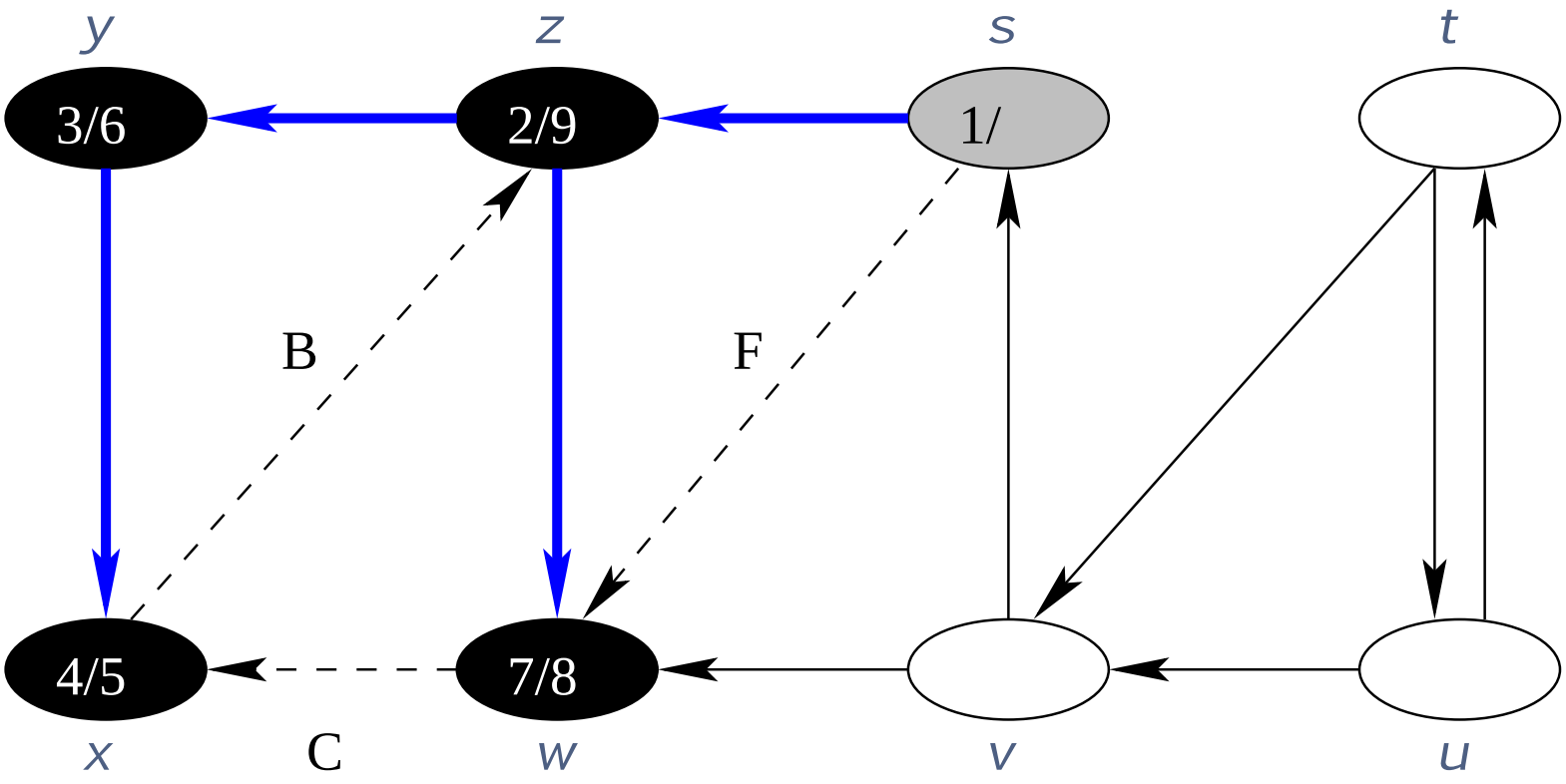
Exemplo de busca em profundidade



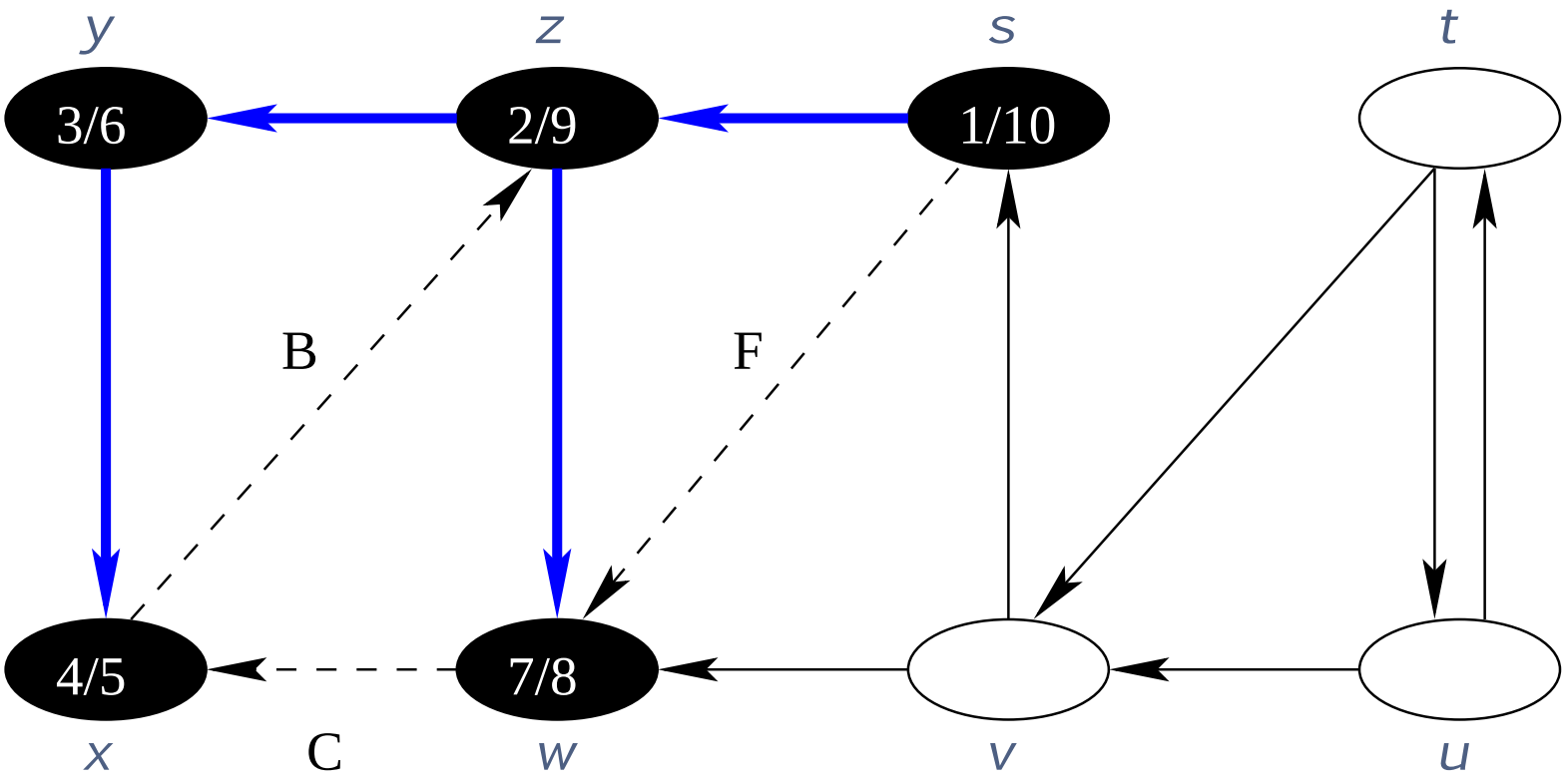
Exemplo de busca em profundidade



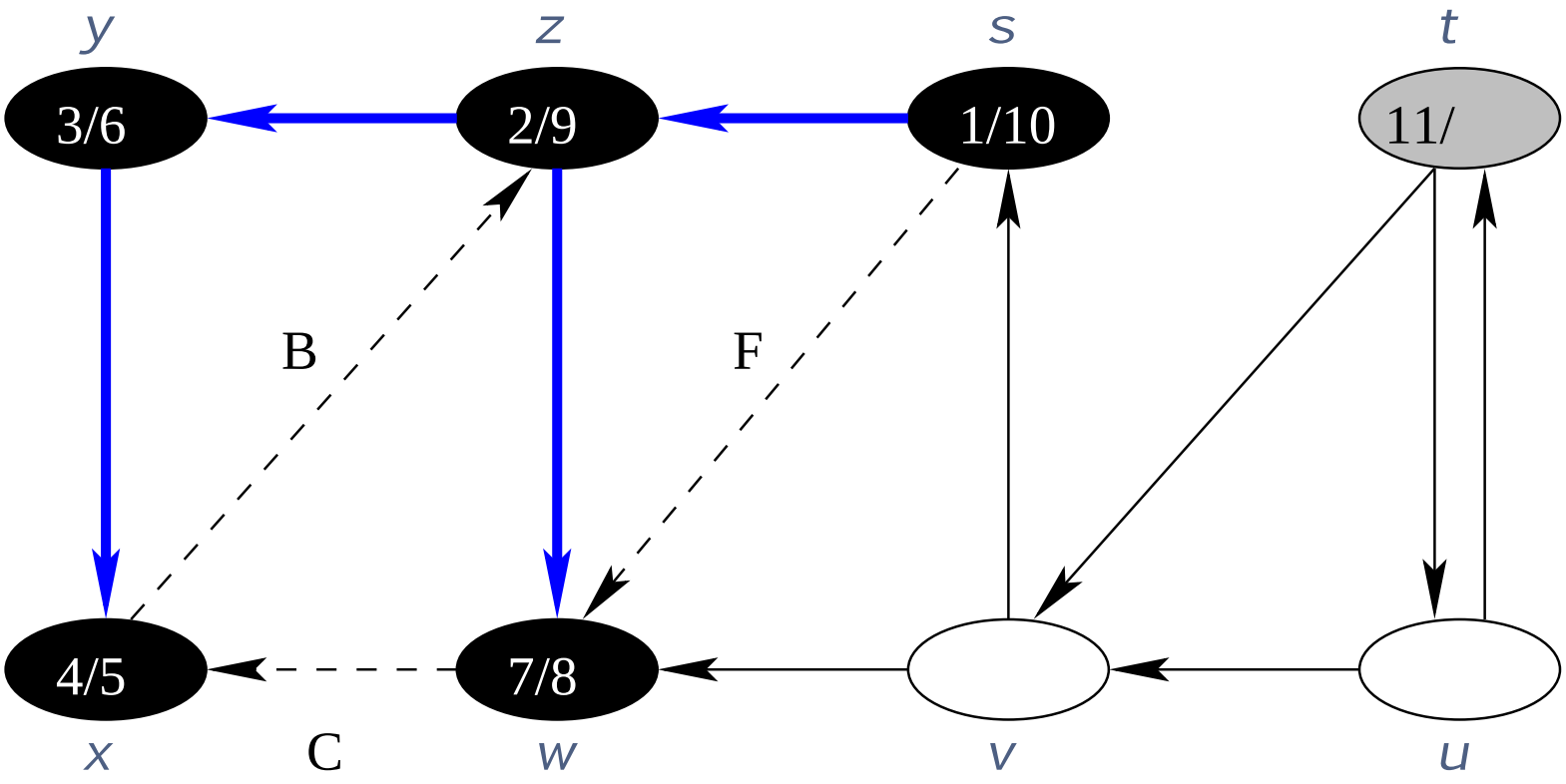
Exemplo de busca em profundidade



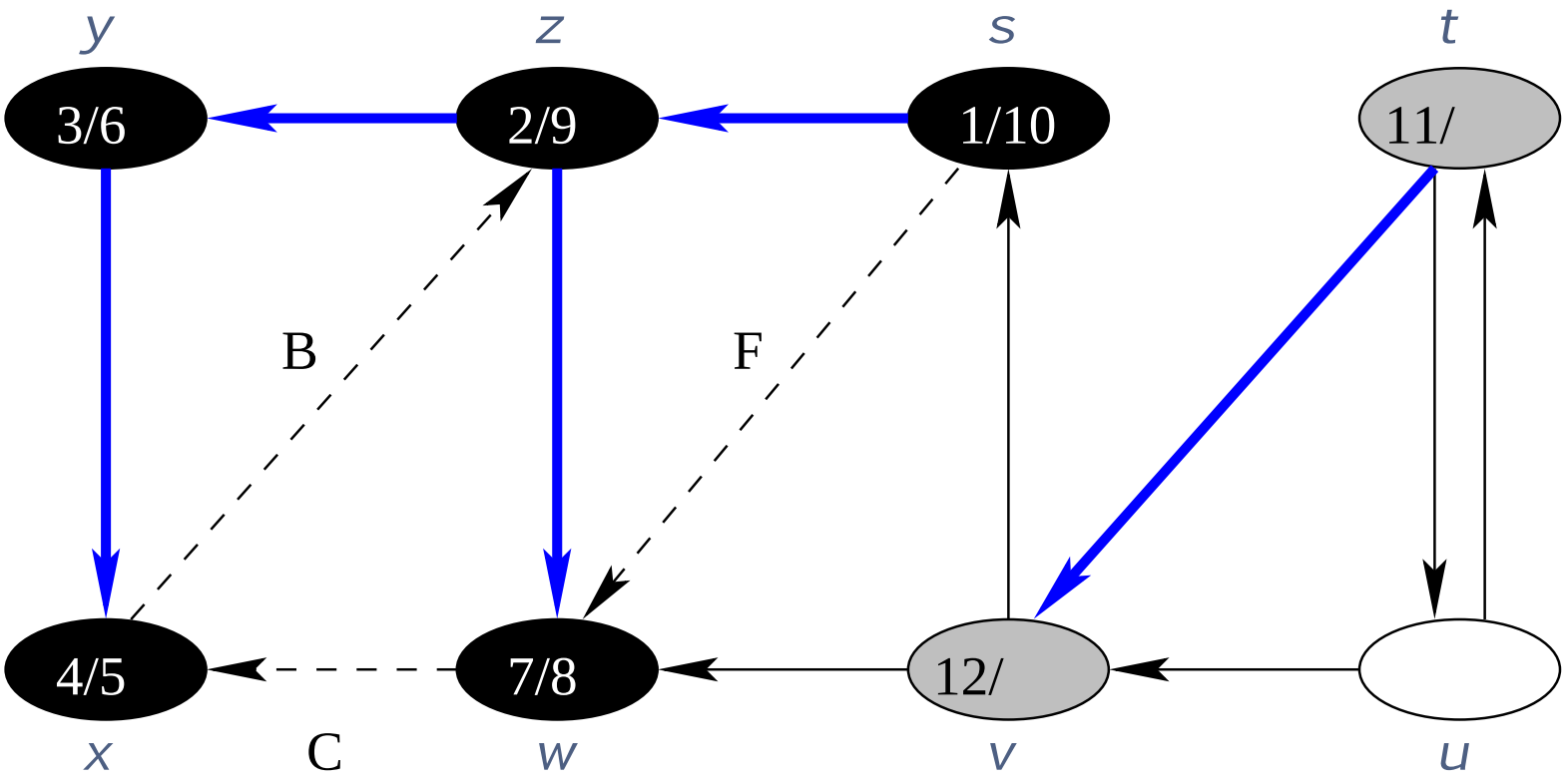
Exemplo de busca em profundidade



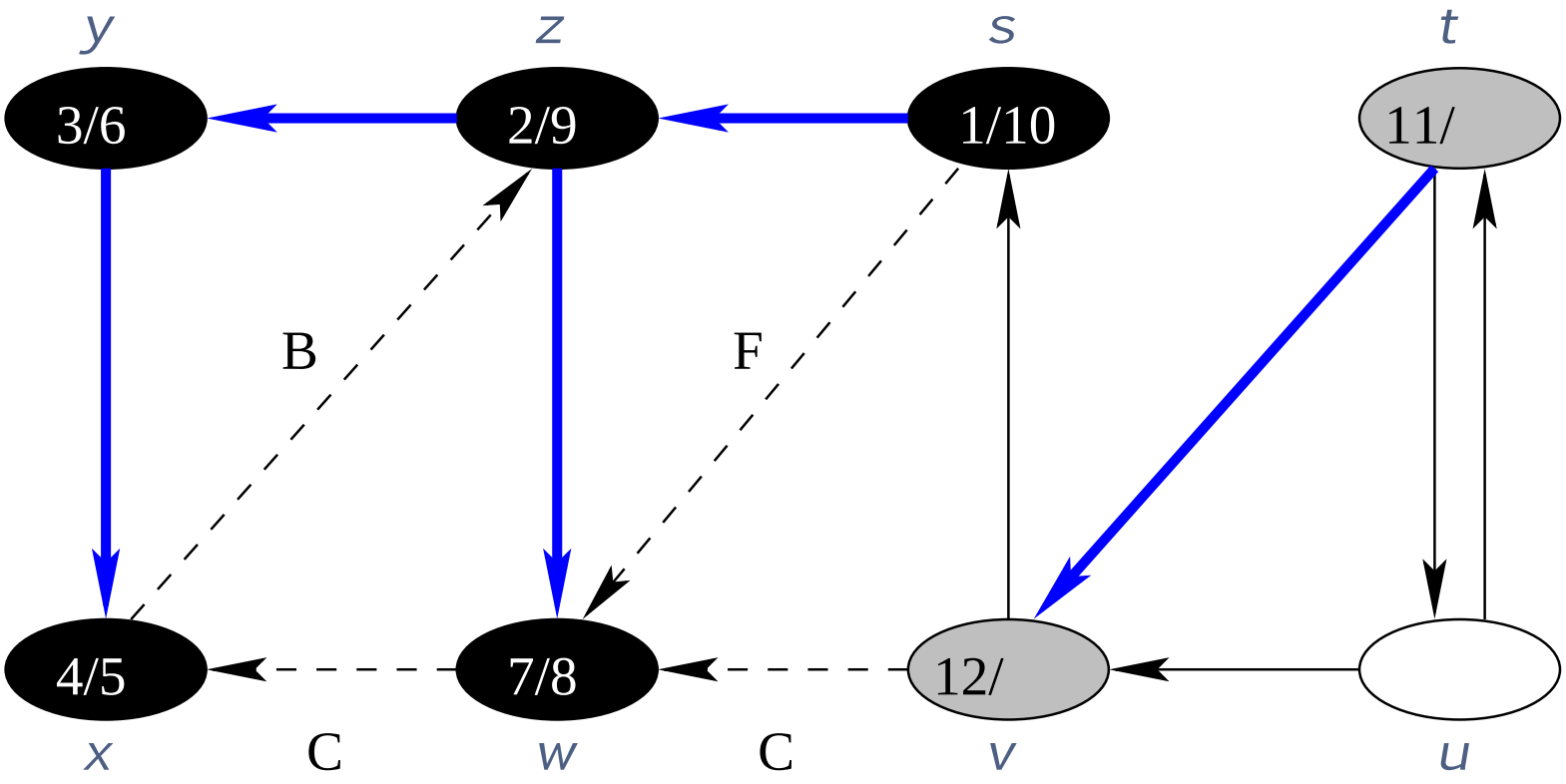
Exemplo de busca em profundidade



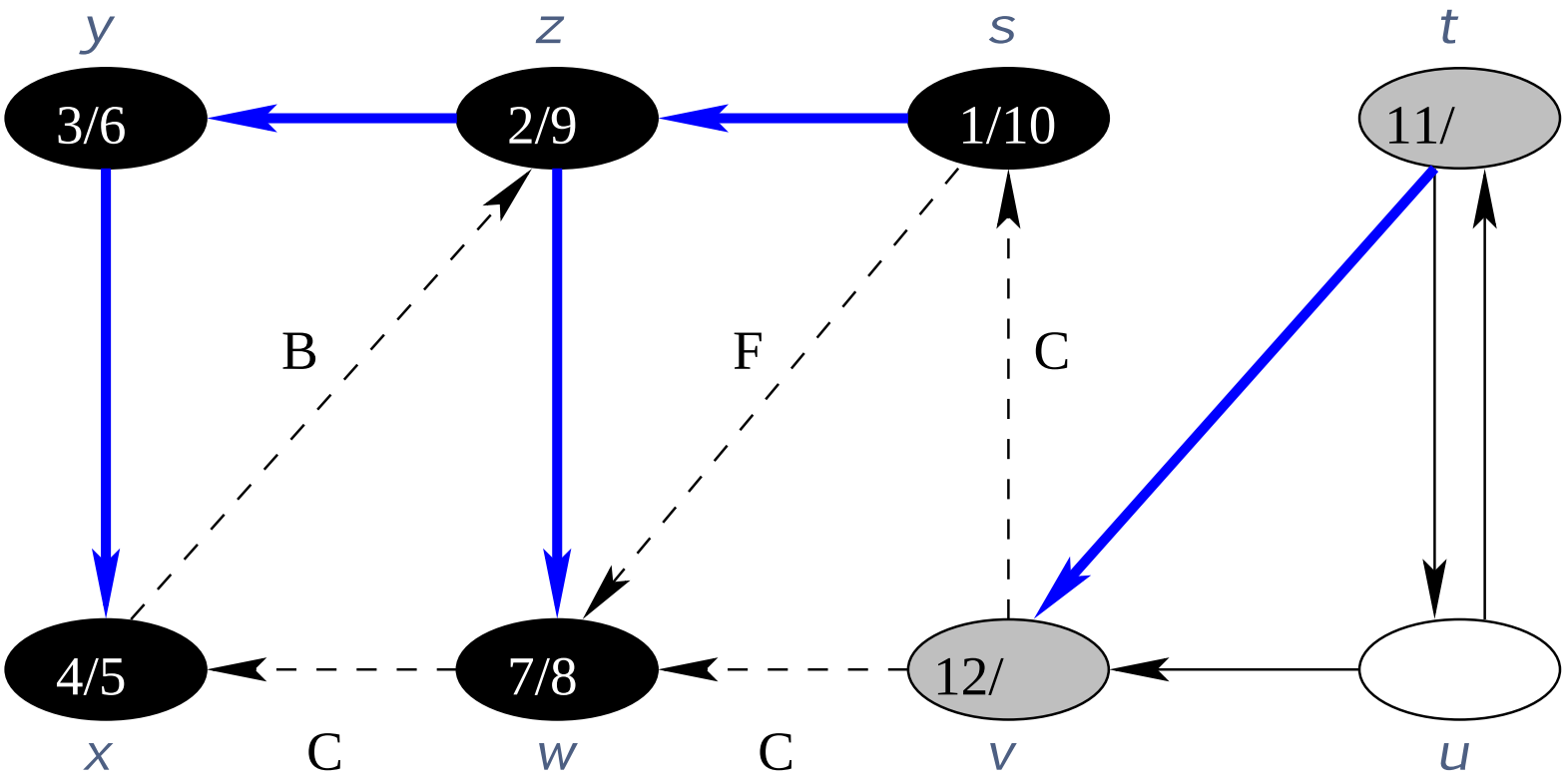
Exemplo de busca em profundidade



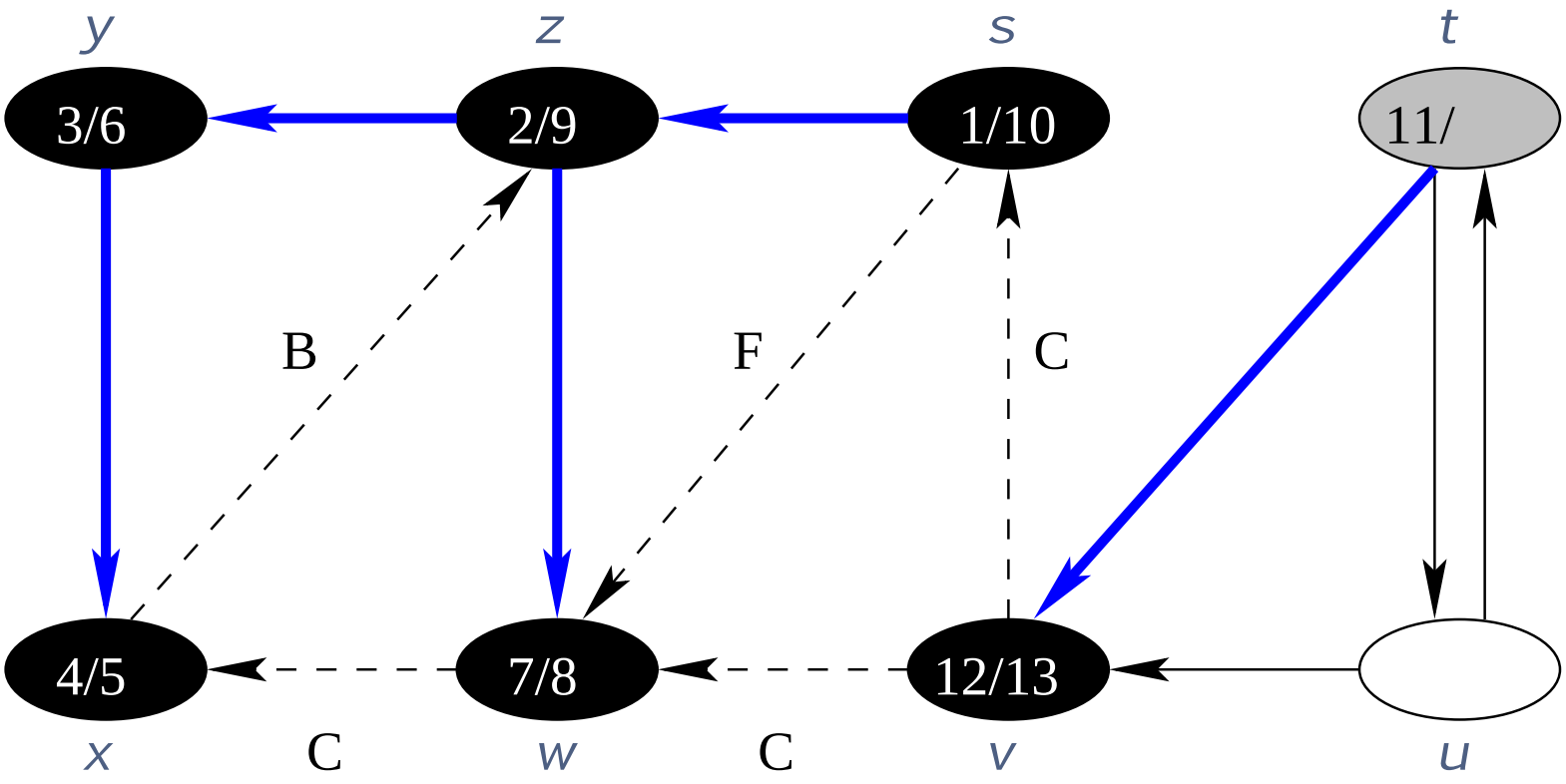
Exemplo de busca em profundidade



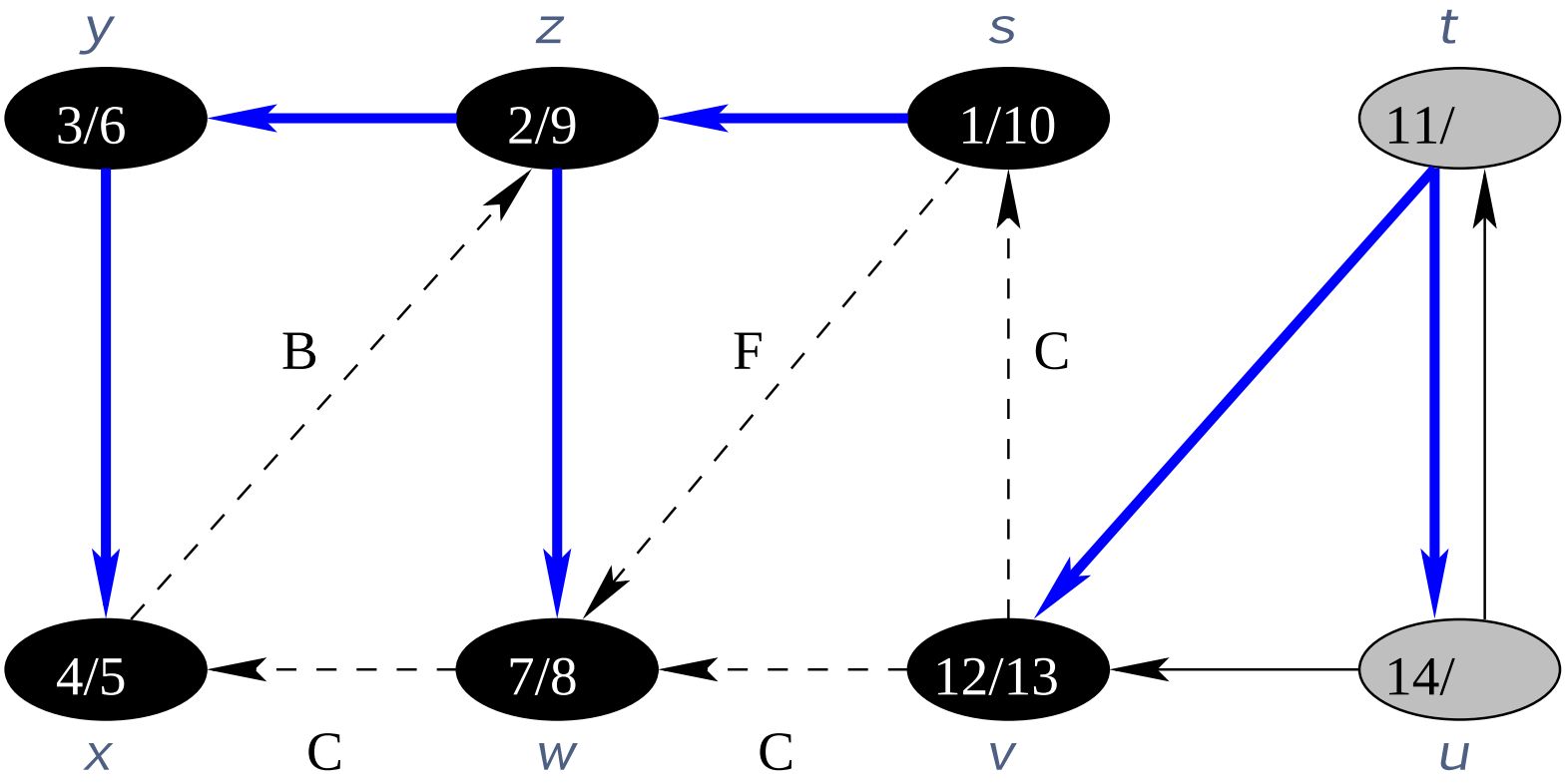
Exemplo de busca em profundidade



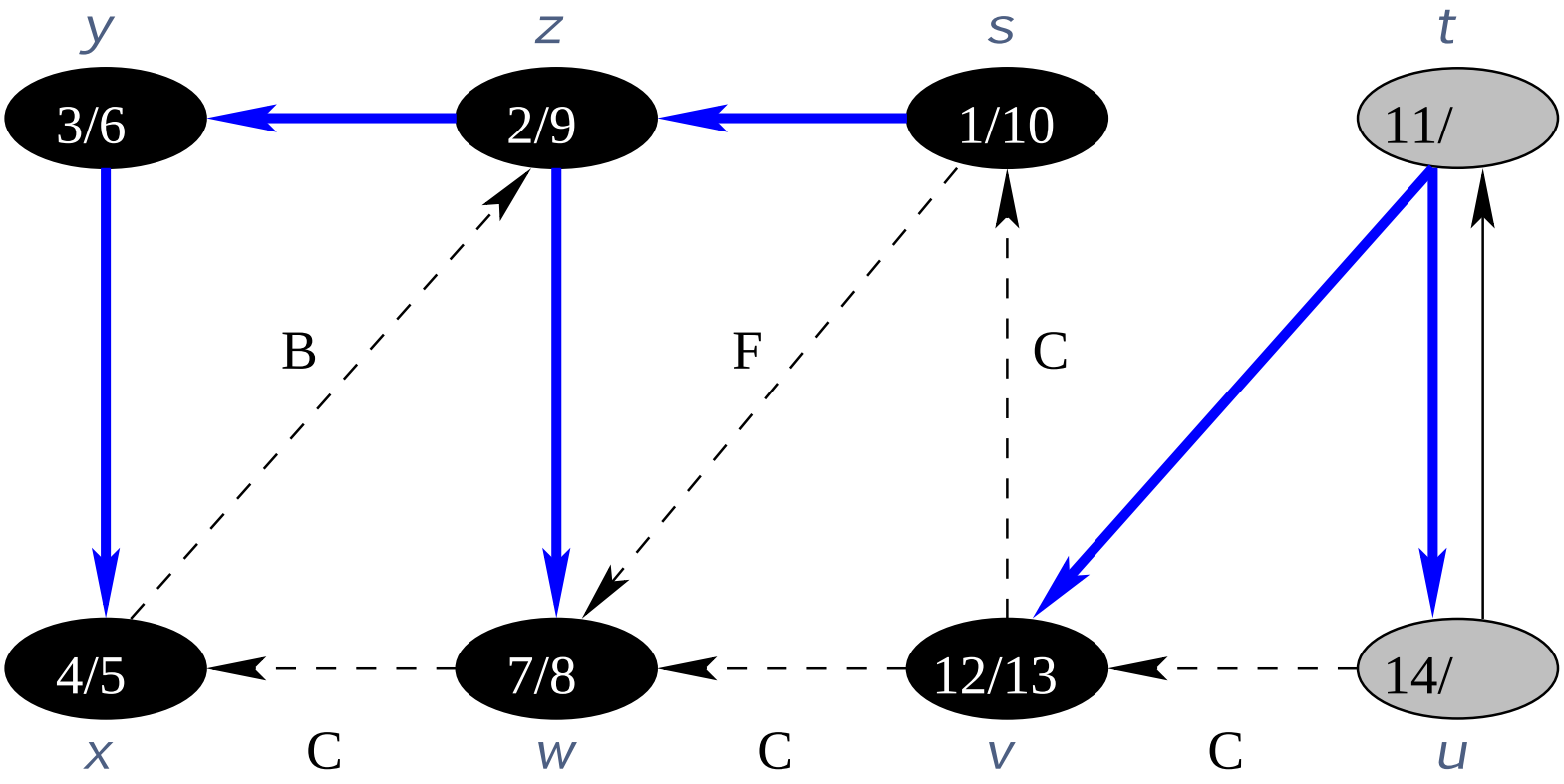
Exemplo de busca em profundidade



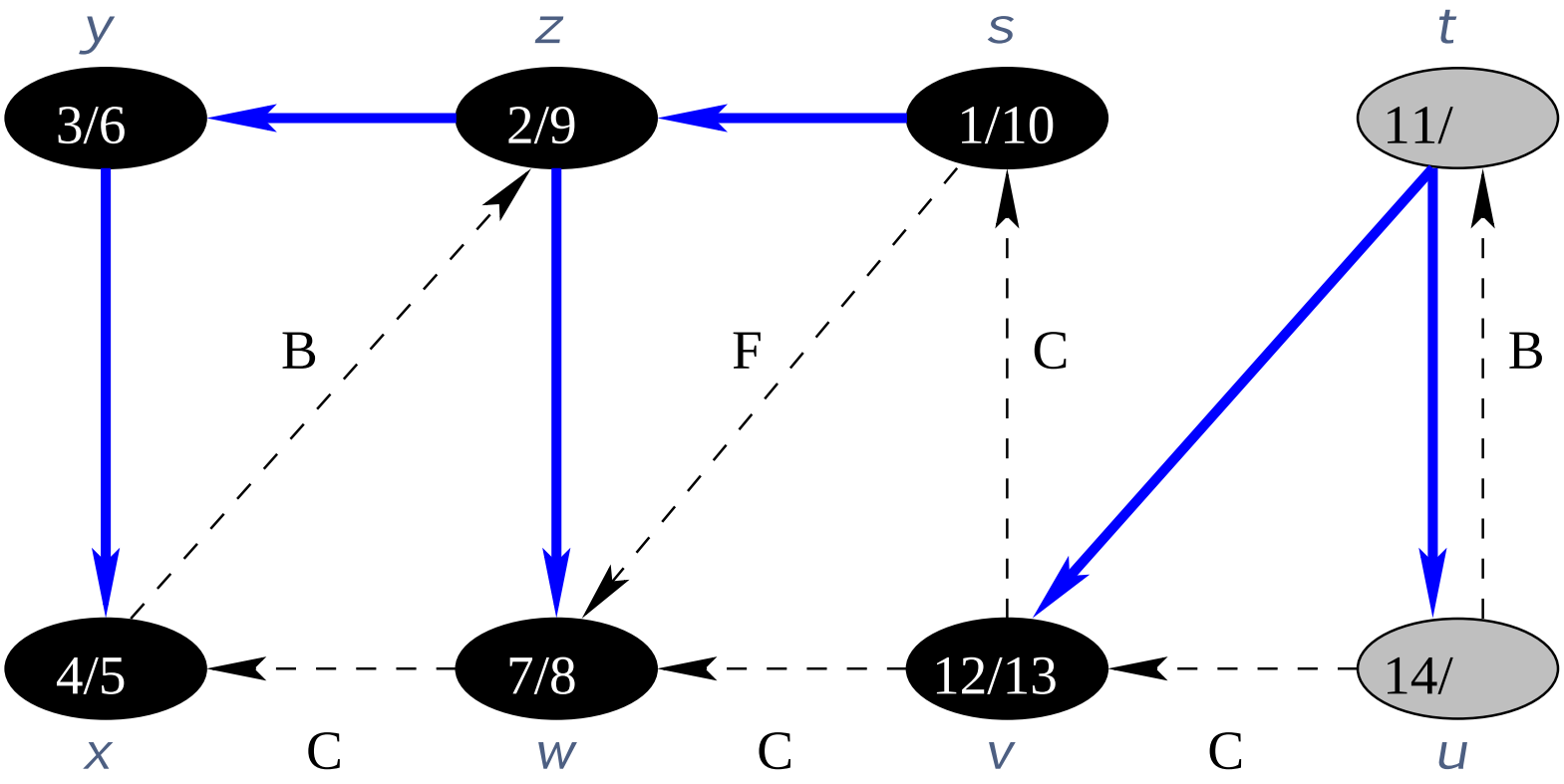
Exemplo de busca em profundidade



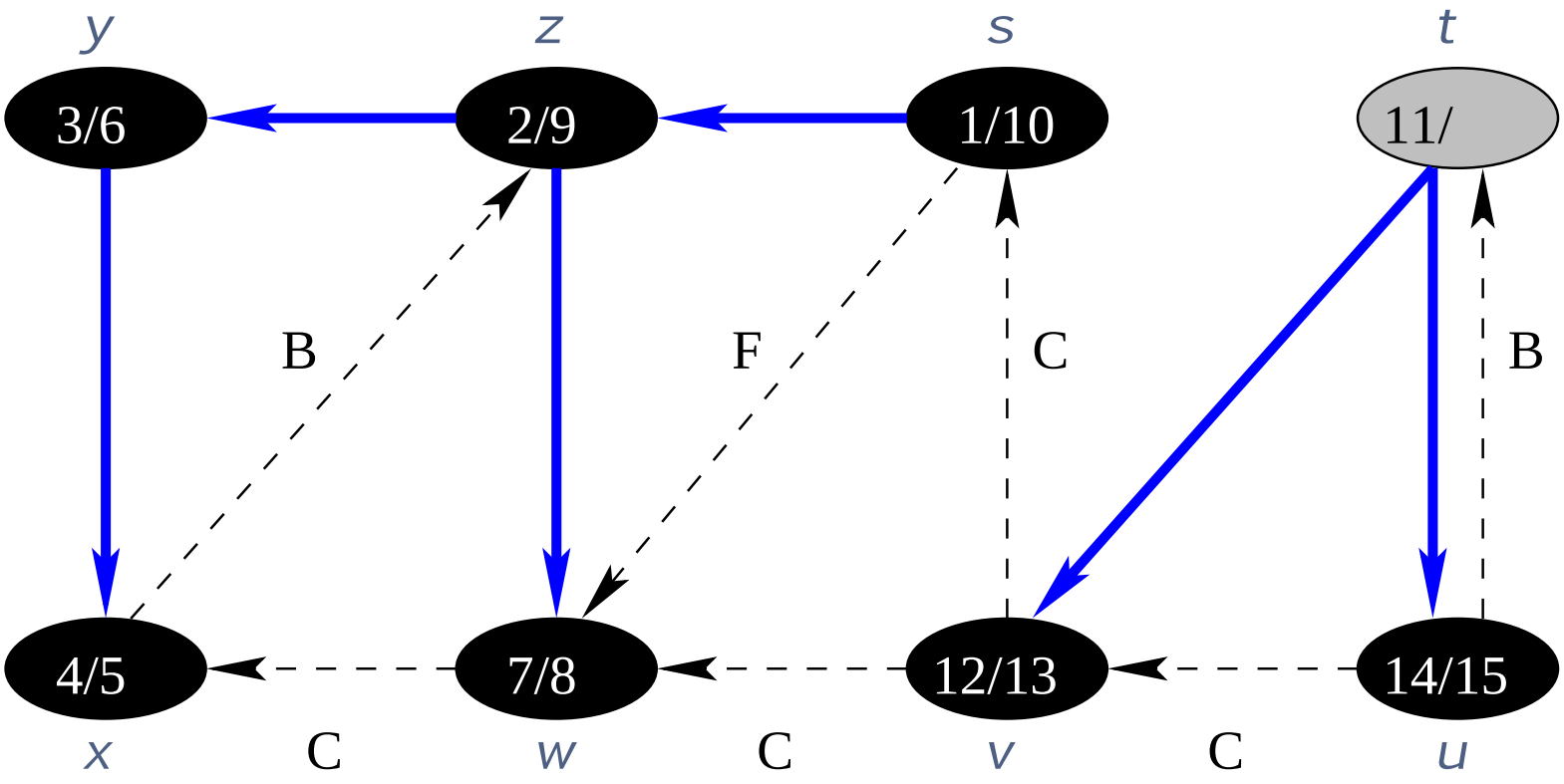
Exemplo de busca em profundidade



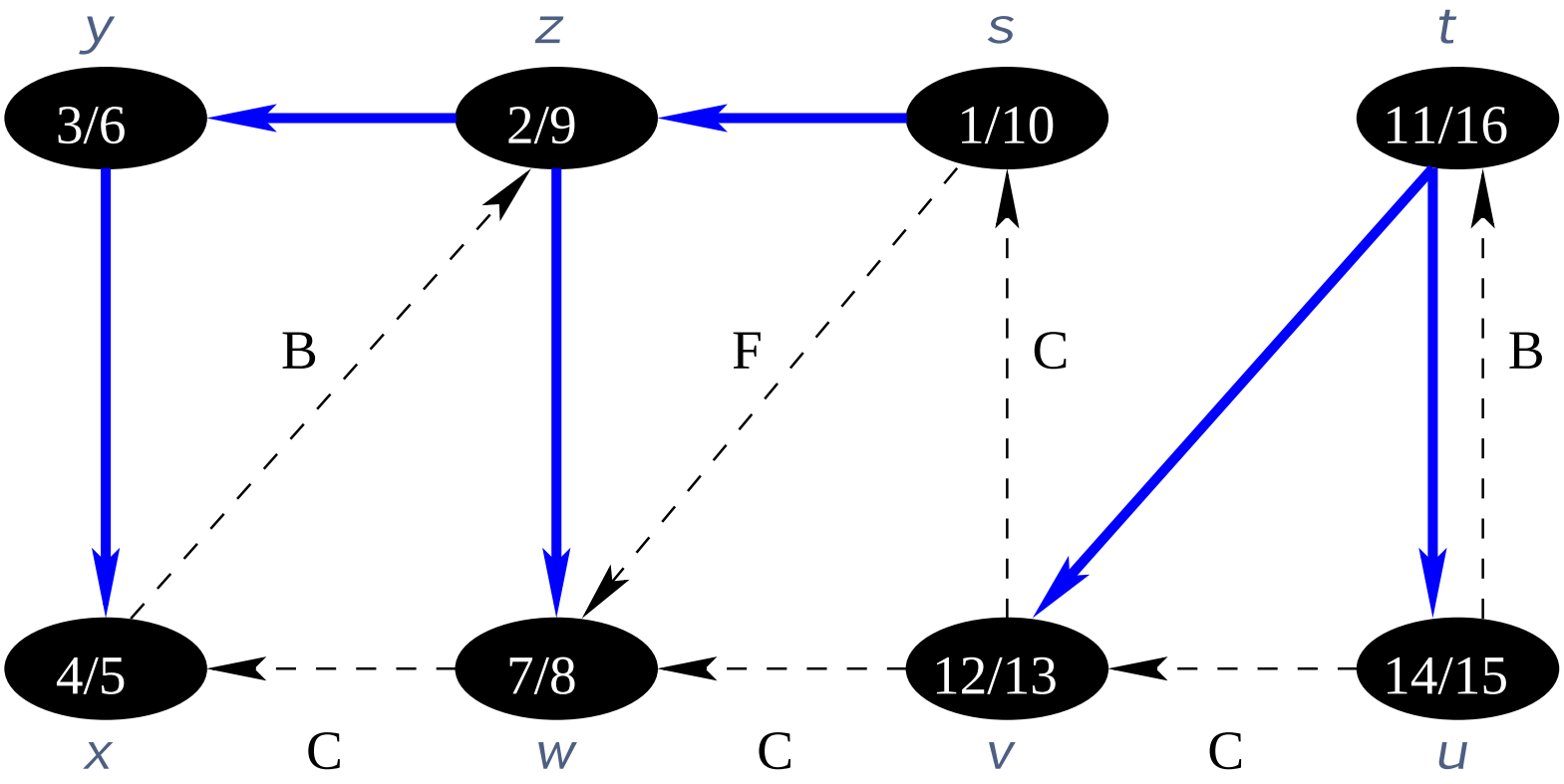
Exemplo de busca em profundidade



Exemplo de busca em profundidade



Exemplo de busca em profundidade



Rótulos versus cores

Observe que para todo vértice v

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

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 então
7           DFS-VISIT(u)
```

- representamos G com listas de adjacências
- a floresta de busca em profundidade é representada por π
- calcula os instantes $d[v]$ e $f[v]$

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 então
 6           π[v] ← u
 7           DFS-VISIT(v)
 8   cor[u] ← preto
 9   tempo ← tempo + 1
10   f[u] ← tempo
```

- constrói uma arvore de busca com origem u

Análise de complexidade

Analisamos o tempo do algoritmo principal DFS

- a inicialização consome tempo $O(V)$
- realizamos $|V|$ chamadas a DFS-VISIT

E o tempo da sub-rotina DFS-VISIT

- processamos cada vértice exatamente uma vez
- cada chamada percorre sua lista de adjacências
- o tempo gasto percorrendo adjacências é $O(E)$

A complexidade da busca em profundidade é $O(V + E)$

Estrutura de parênteses

Teorema dos parênteses

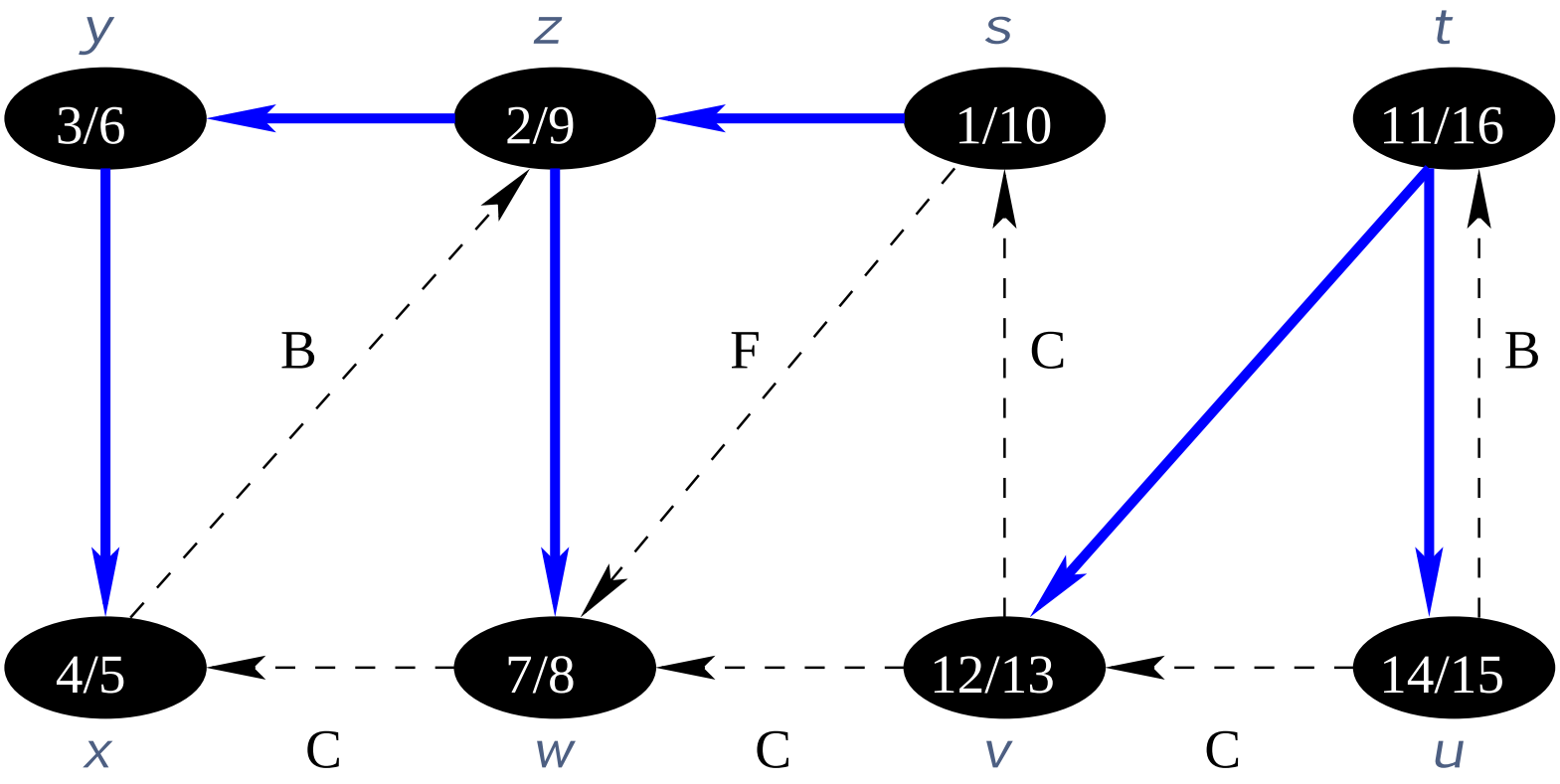
Se u e v são vértices de uma árvore de busca em profundidade, então ocorre exatamente um entre os três casos abaixo:

1. (a) os intervalos $[d[u], f[u]]$ e $[d[v], f[v]]$ são disjuntos
   (b) nesse caso u e v não são descendentes um do outro

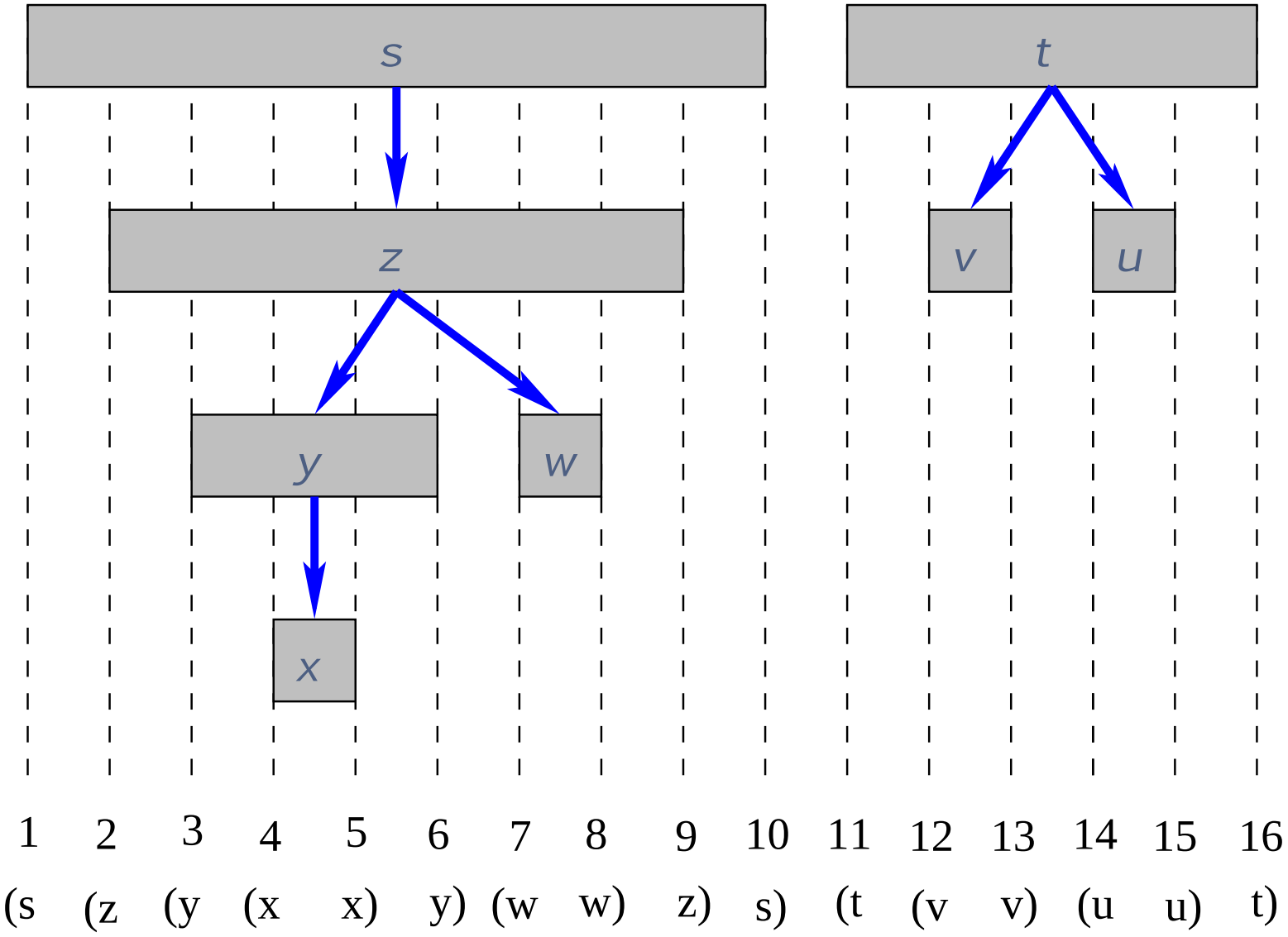
2. (a) o intervalo $[d[u], f[u]]$ está contido em $[d[v], f[v]]$
   (b) nesse caso u é descendente de v

3. (a) o intervalo $[d[v], f[v]]$ está contido em $[d[u], f[u]]$
   (b) nesse caso v é descendente de u

Exemplo de floresta de busca



Exemplo de estrutura de parênteses



Prova do teorema

Demonstração do teorema dos parênteses

- podemos supor que $d[u] < d[v]$
- analisamos dois casos

  Caso 1: suponha que $d[v] < f[u]$
  - então v foi descoberto enquanto u era cinza
  - e a chamada recursiva para v termina antes da de u
  - portanto v é descendente de u

  Caso 2: suponha que $f[u] < d[v]$
  - então u foi finalizado enquanto v era branco
  - e a chamada de u termina antes que a de v comece
  - portanto u e v não são descendentes um do outro

- no primeiro caso, $[d[v], f[v]]$ está contido em $[d[u], f[u]]$
- e no segundo, $[d[v], f[v]]$ e $[d[u], f[u]]$ são disjuntos

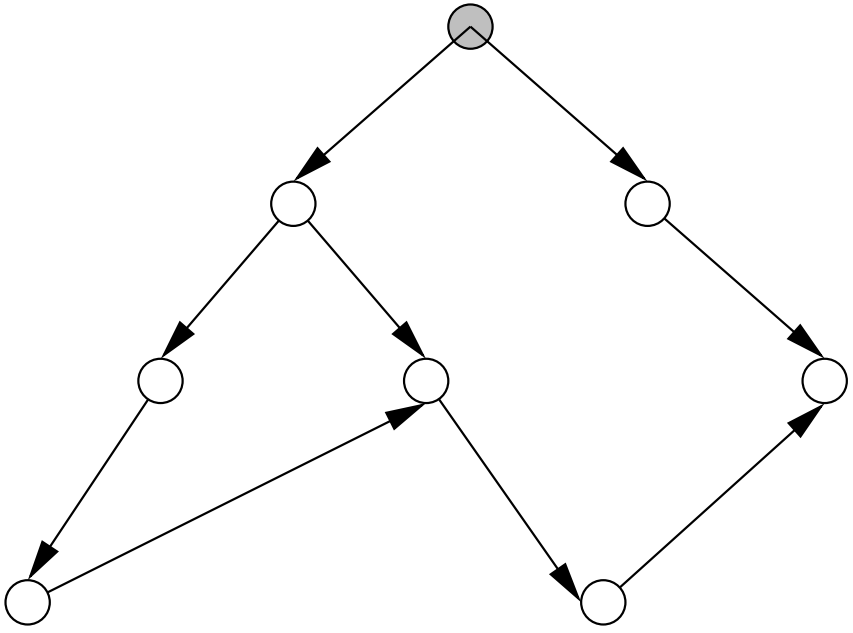
Vértices alcançáveis

Teorema do caminho branco

Considere dois vértices u e v. São equivalentes:

(1) v é descendente de u na floresta de busca

(2) quando u foi descoberto, existia um caminho de u a v formado apenas por vértices brancos



Prova do teorema

Demonstração do teorema do caminho branco

- $(1) \Rightarrow (2)$
  - suponha que v é um descendente de u
  - seja z um vértice no caminho de u até v na floresta
  - então z é descendente de u e daí $d[u] < d[z]$
  - assim z era branco no instante $d[u]$
  - assim como todos os vértices no caminho

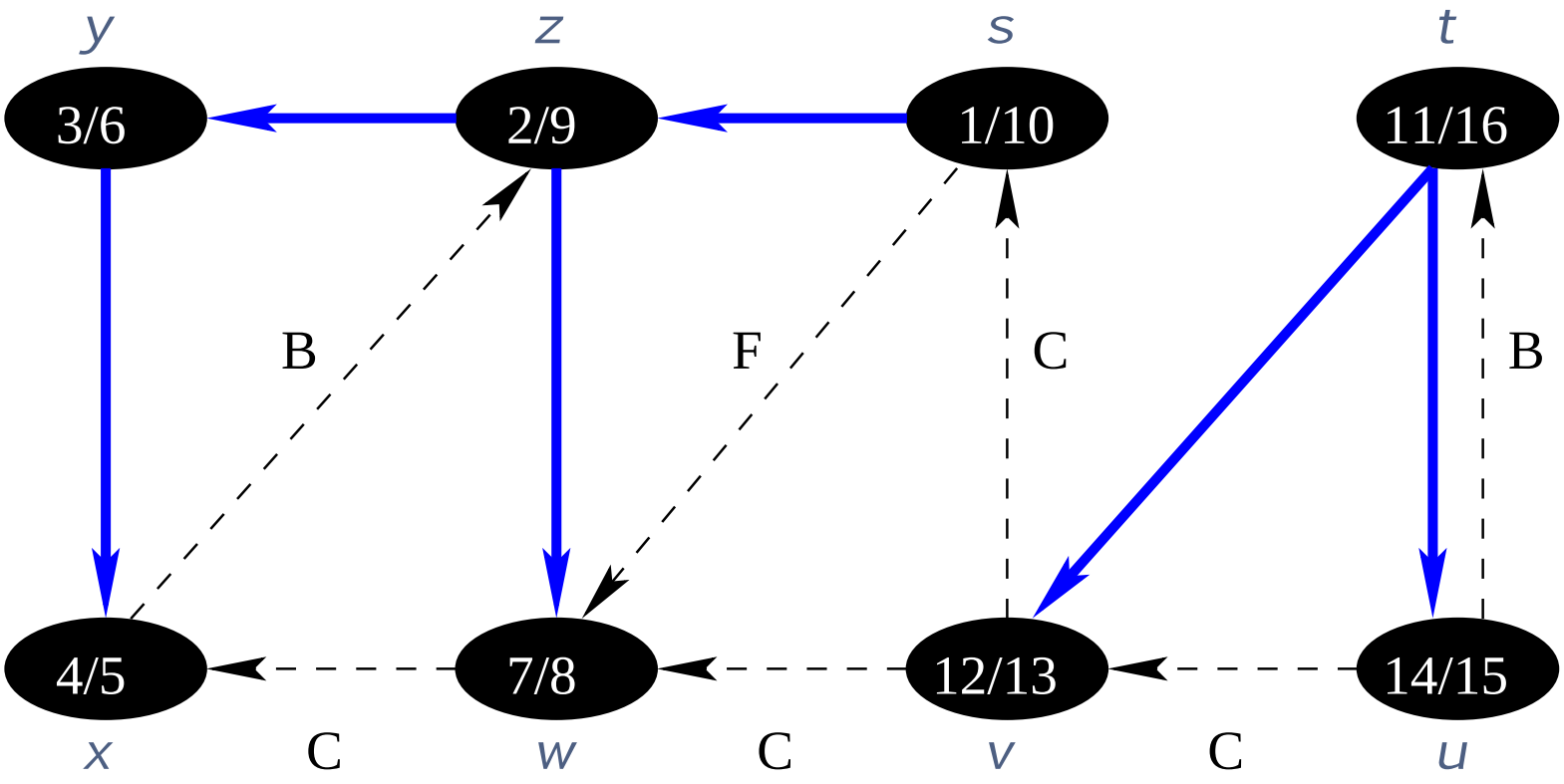
- $(2) \Rightarrow (1)$
  - considere um caminho branco de u a v no instante $d[u]$
  - suponha que há vértice no caminho não descendente de u
  - seja z o primeiro vértice não descendente de u no caminho
  - e w o vértice antecessor de z nesse caminho
  - como w é descendente de u, temos $f[w] \leq f[u]$
  - como z não é descendente de u, temos $f[u] < d[z]$
  - assim v era um vizinho branco de z no instante $f[z]$
  - isso é uma contradição, então todo vértice do caminho branco é descendente de u na floresta de busca

Classificação de arestas

Dada a floresta de busca, podemos classificar arestas do grafo

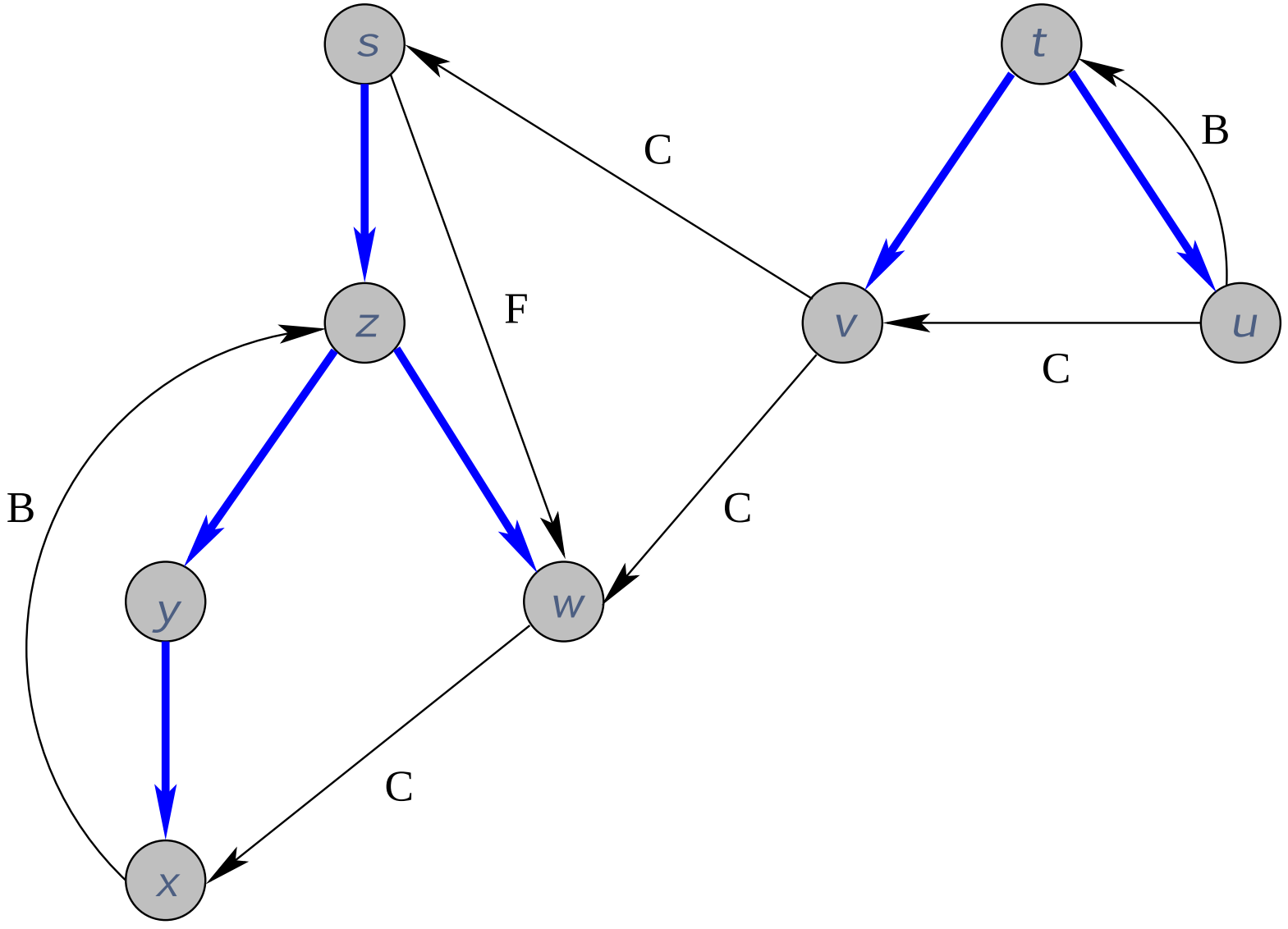
- arestas de árvore (*tree edges*) são arestas da floresta de busca em profundidade
- arestas de retorno (*backward edges*) ligam um vértice a um ancestral
- arestas de avanço (*forward edges*) ligam um vértice a um descendente
- arestas de cruzamento (*cross edges*) são todas as outras arestas do grafo

Classificação de arestas



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

Classificação de arestas

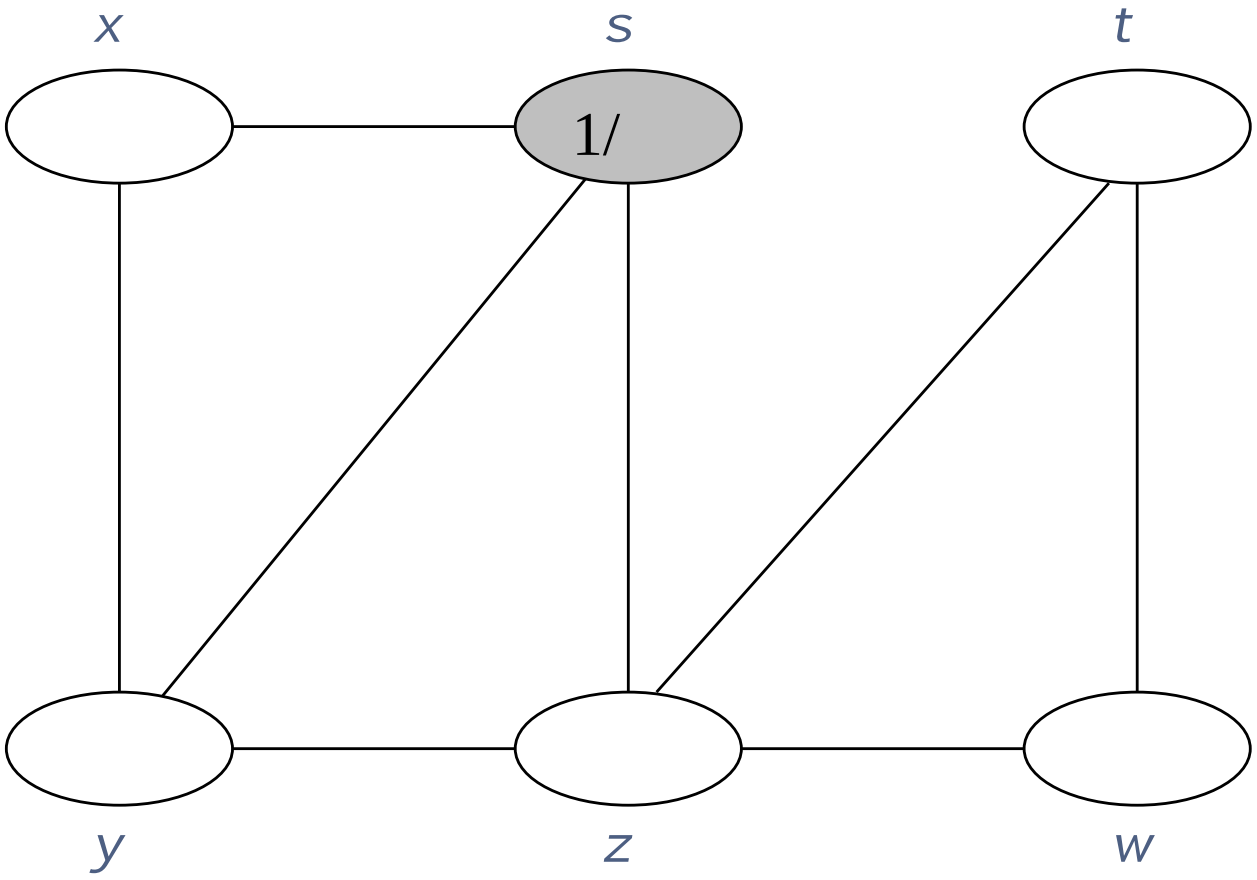


Grafos não direcionados

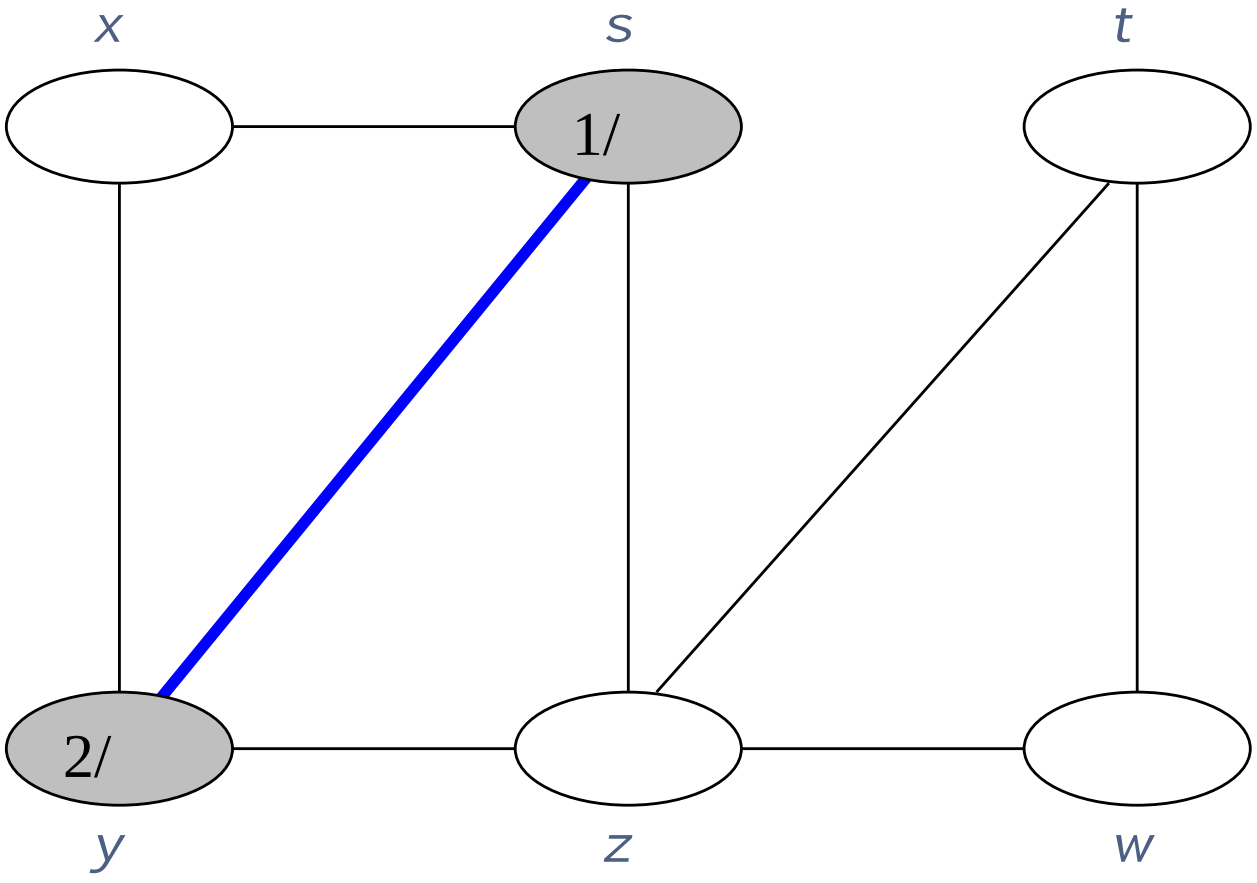
Classificando arestas não direcionadas

- não pode haver aresta de avanço (por quê?)
- tampouco aresta de cruzamento (por quê?)
- daí cada aresta é aresta de árvore ou aresta de retorno

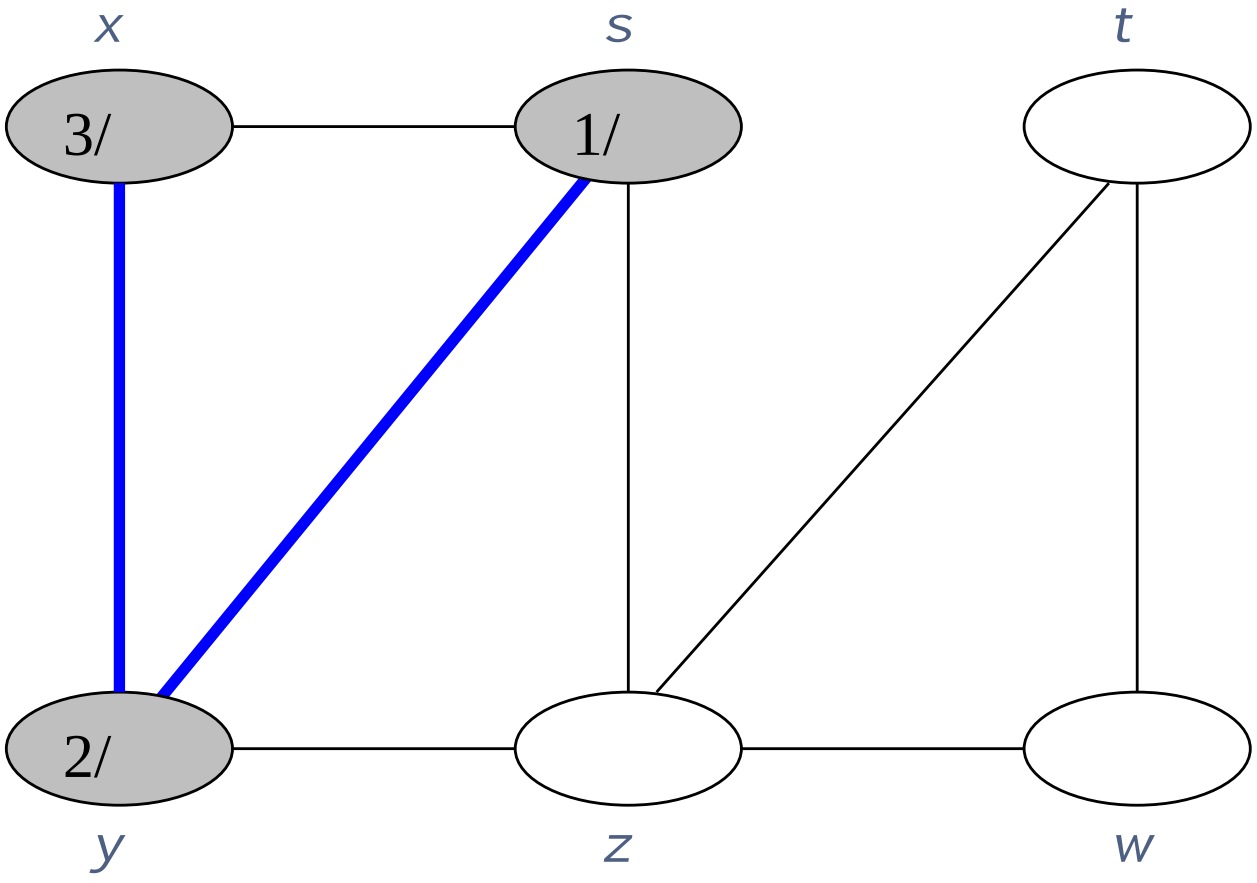
DFS em grafo não direcionado



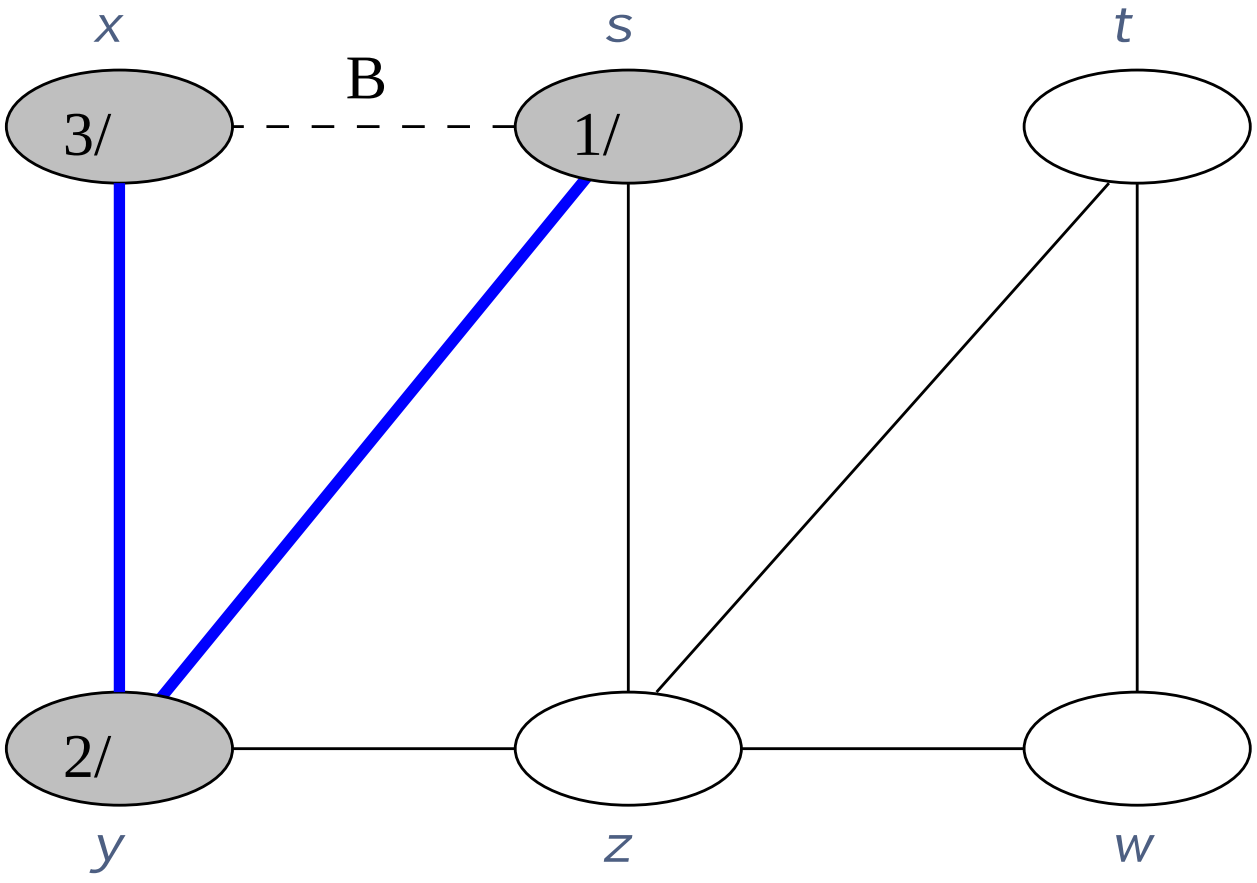
DFS em grafo não direcionado



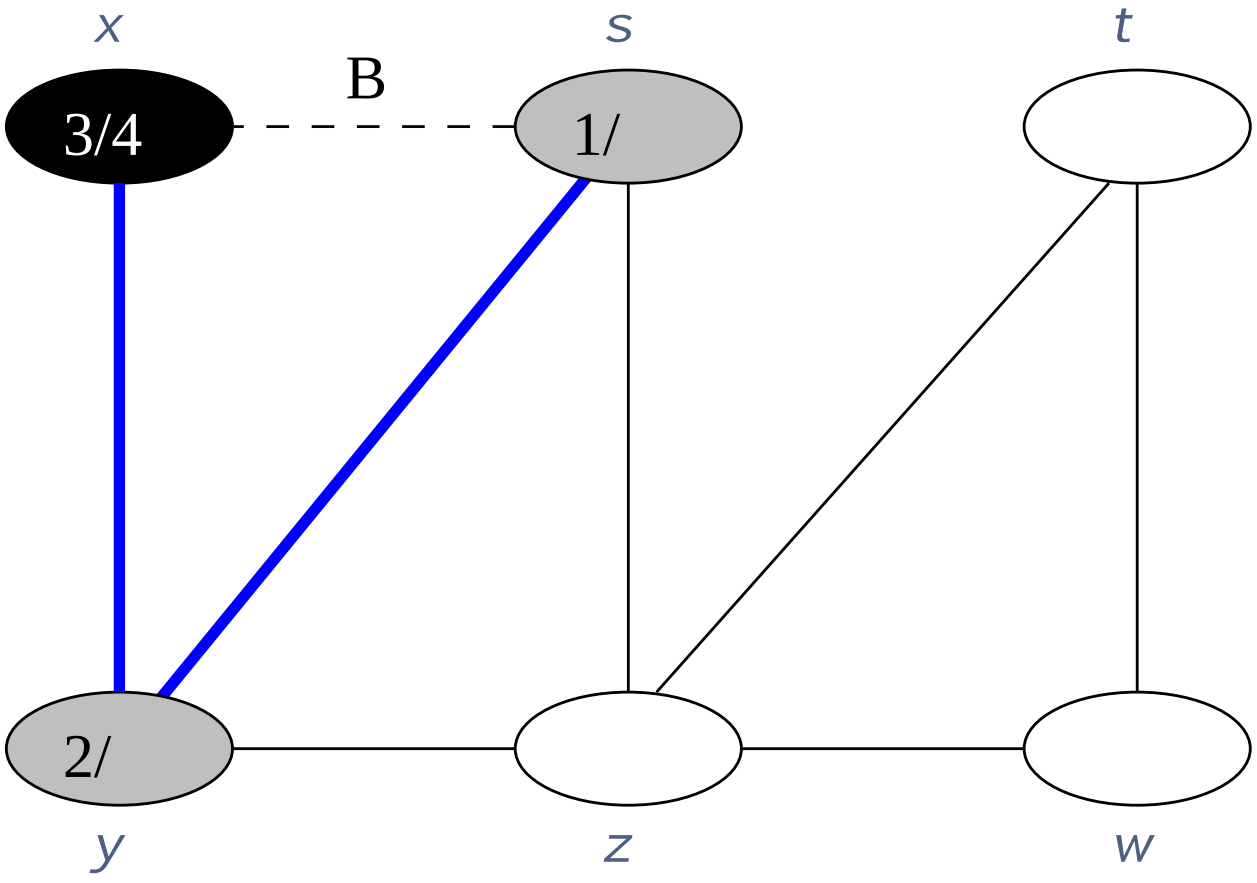
DFS em grafo não direcionado



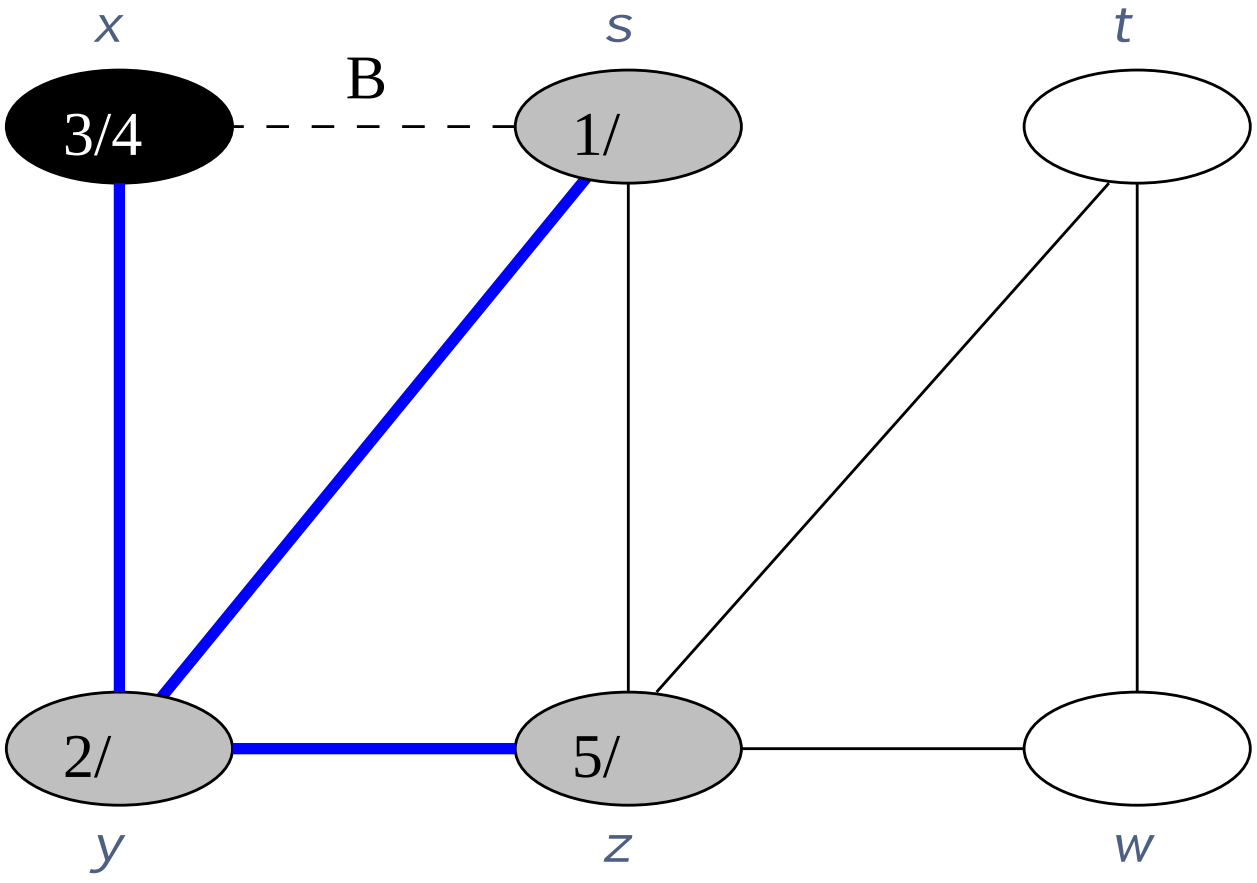
DFS em grafo não direcionado



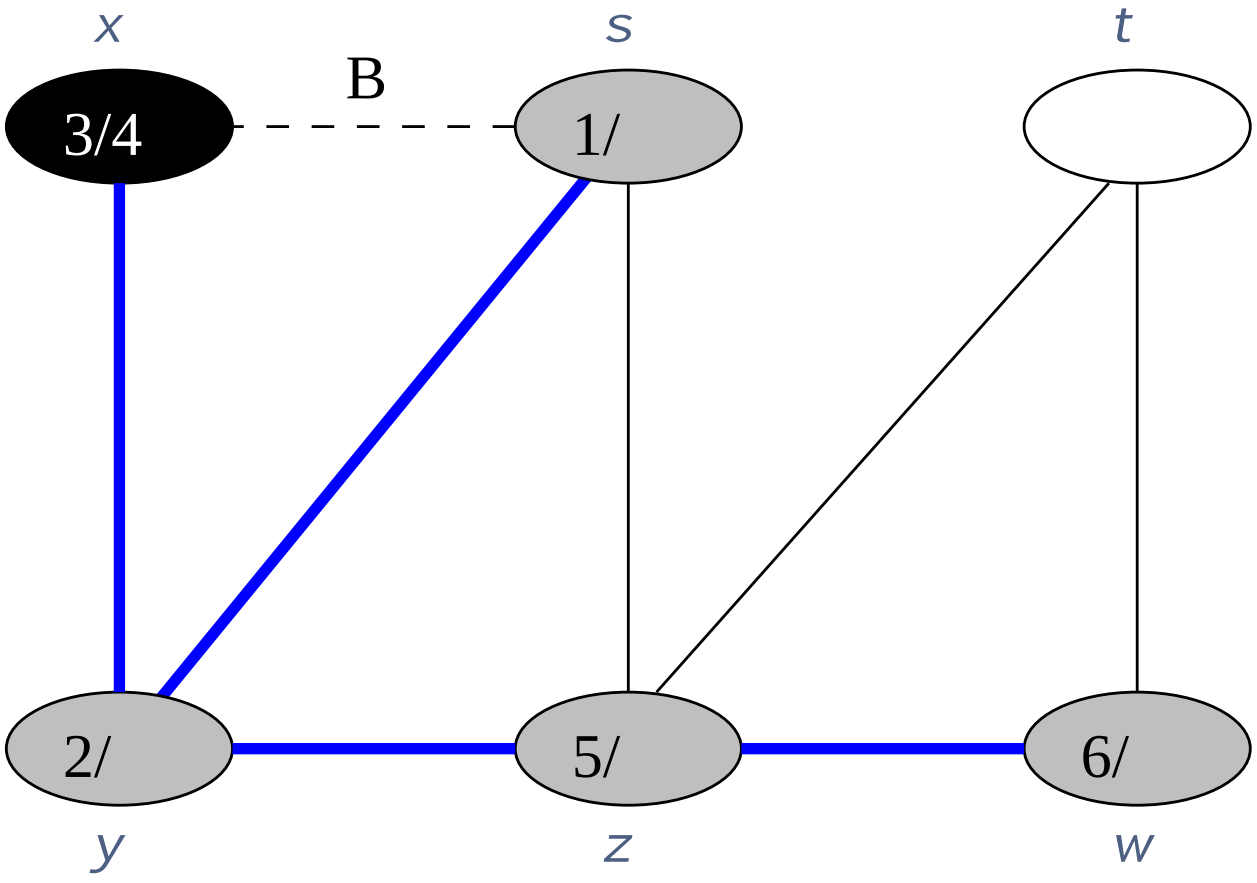
DFS em grafo não direcionado



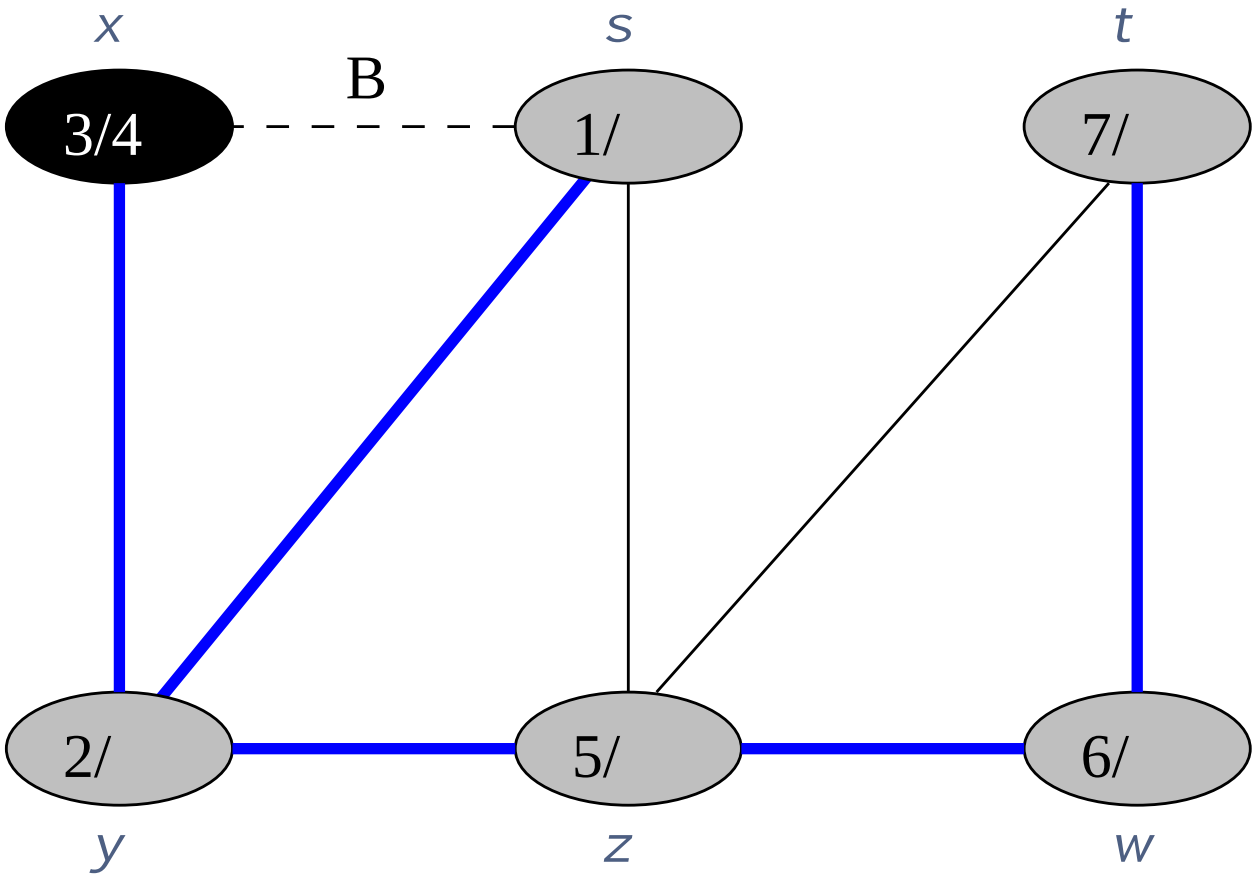
DFS em grafo não direcionado



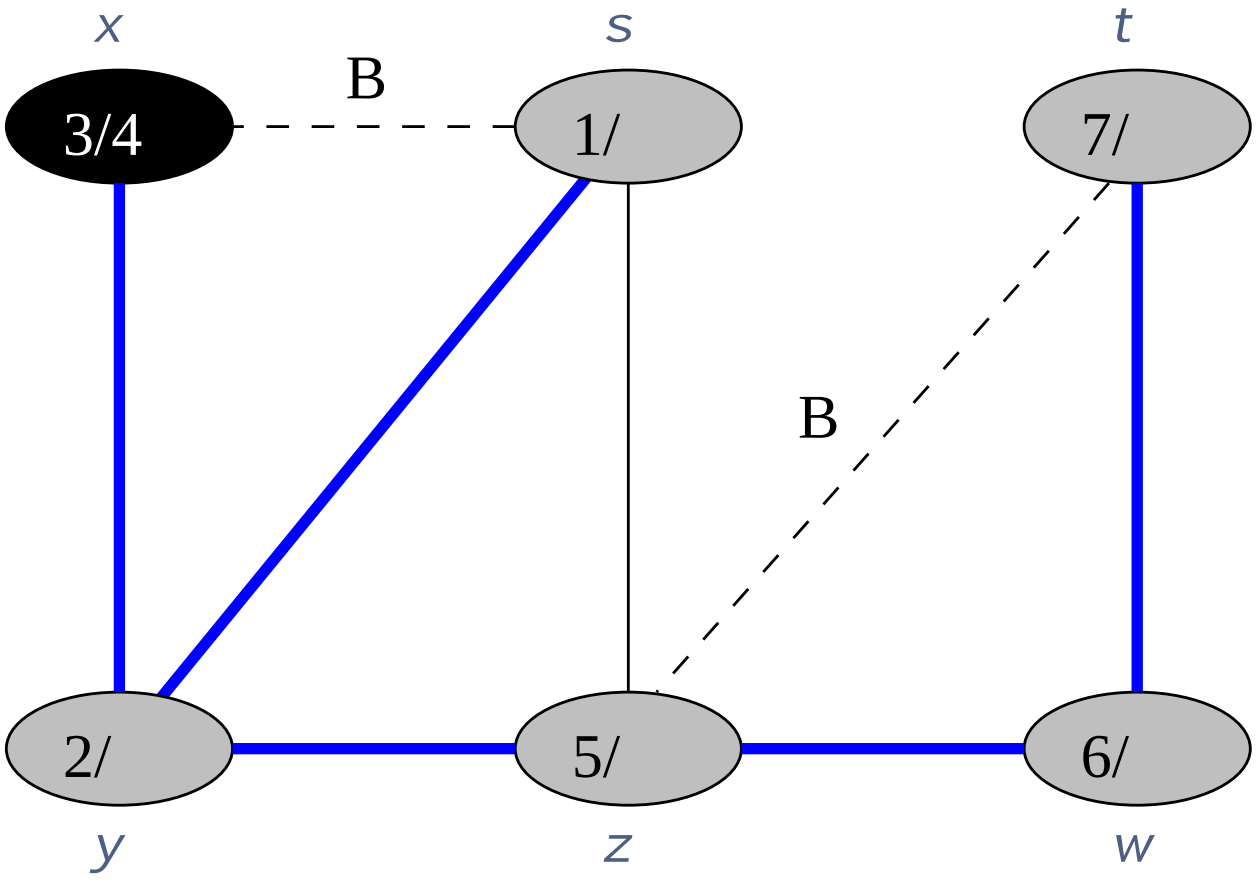
DFS em grafo não direcionado



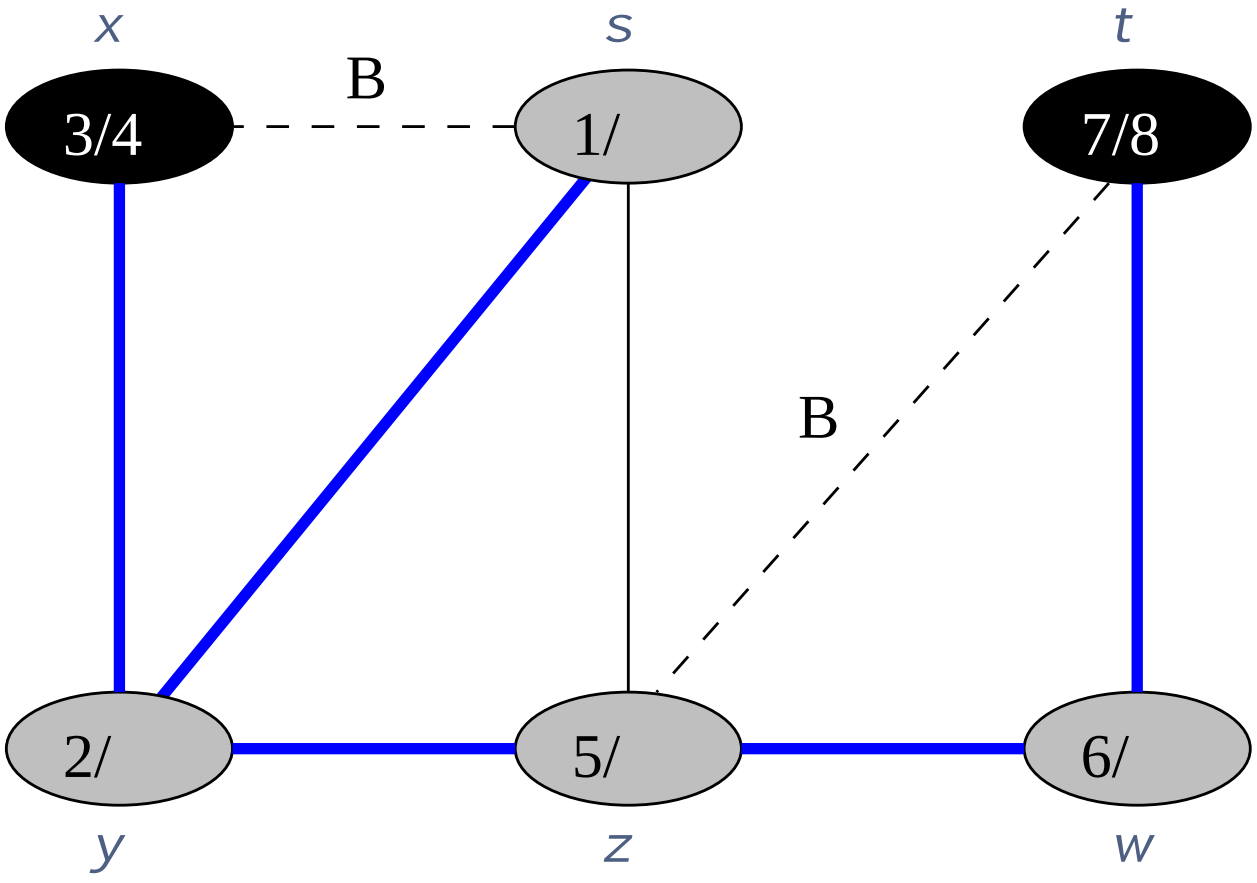
DFS em grafo não direcionado



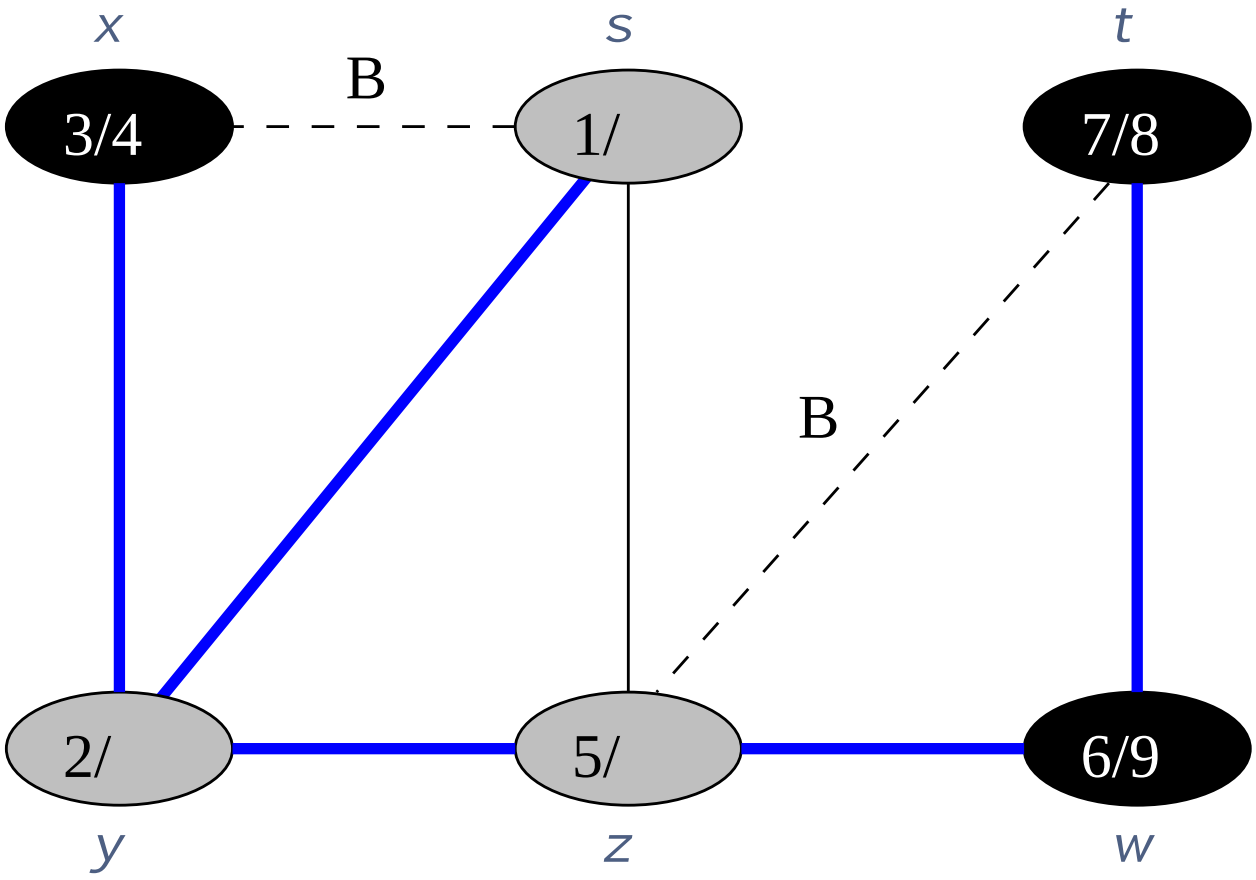
DFS em grafo não direcionado



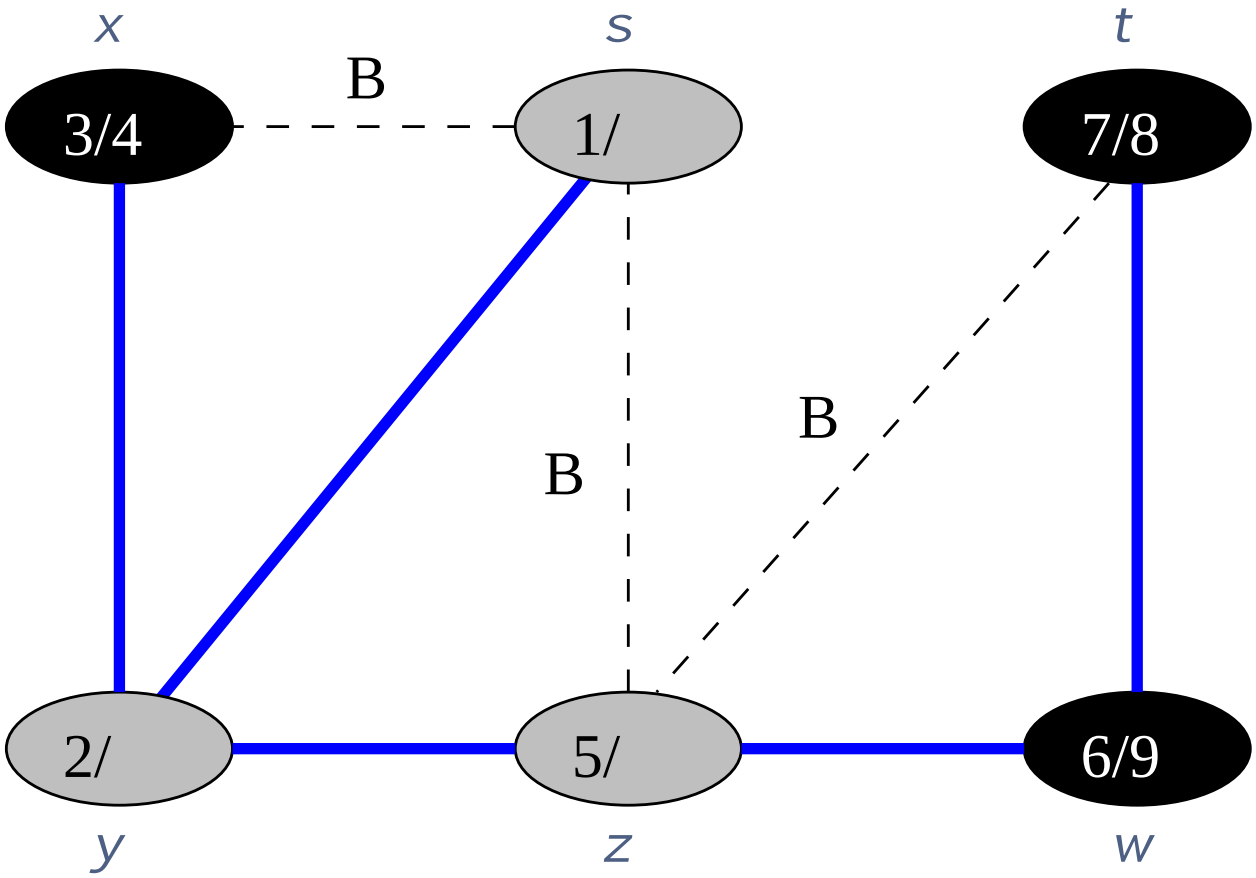
DFS em grafo não direcionado



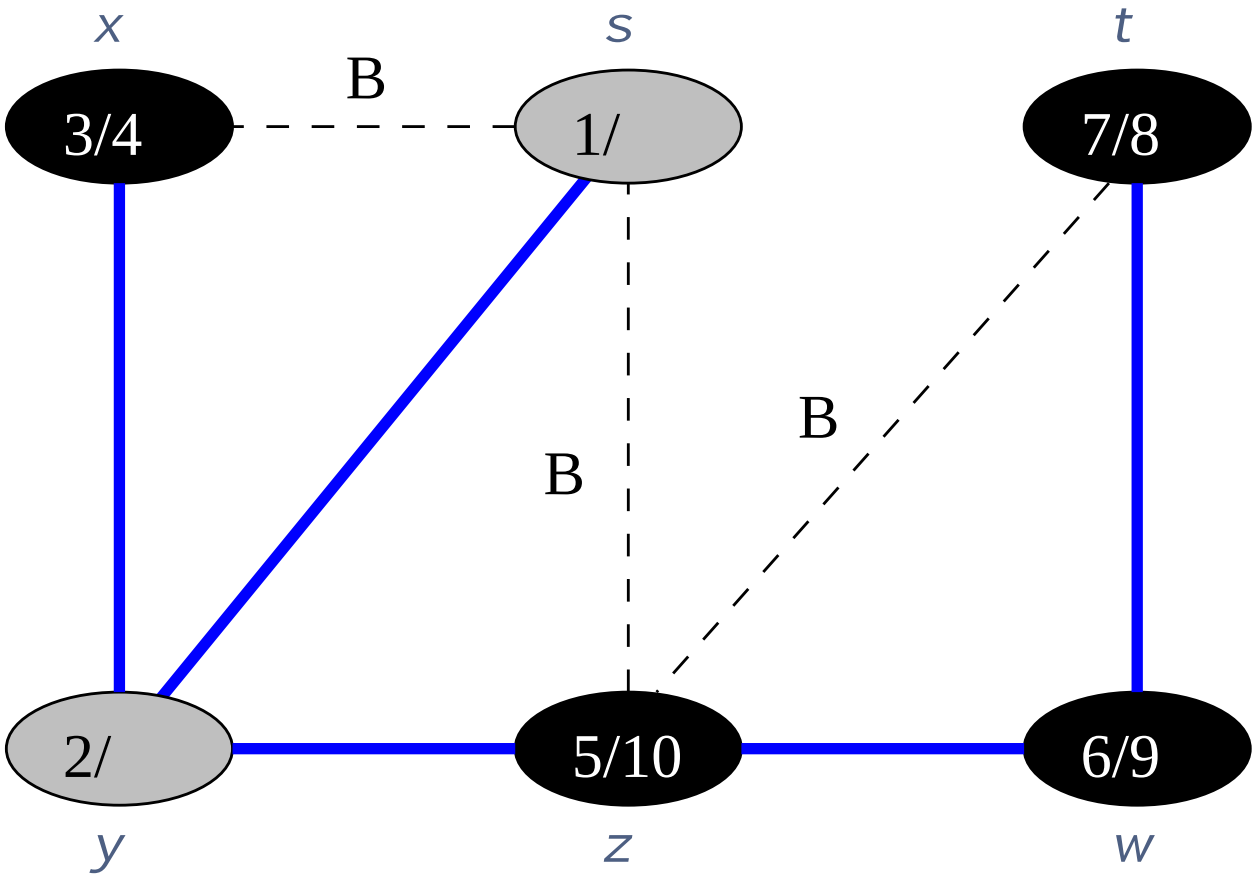
DFS em grafo não direcionado



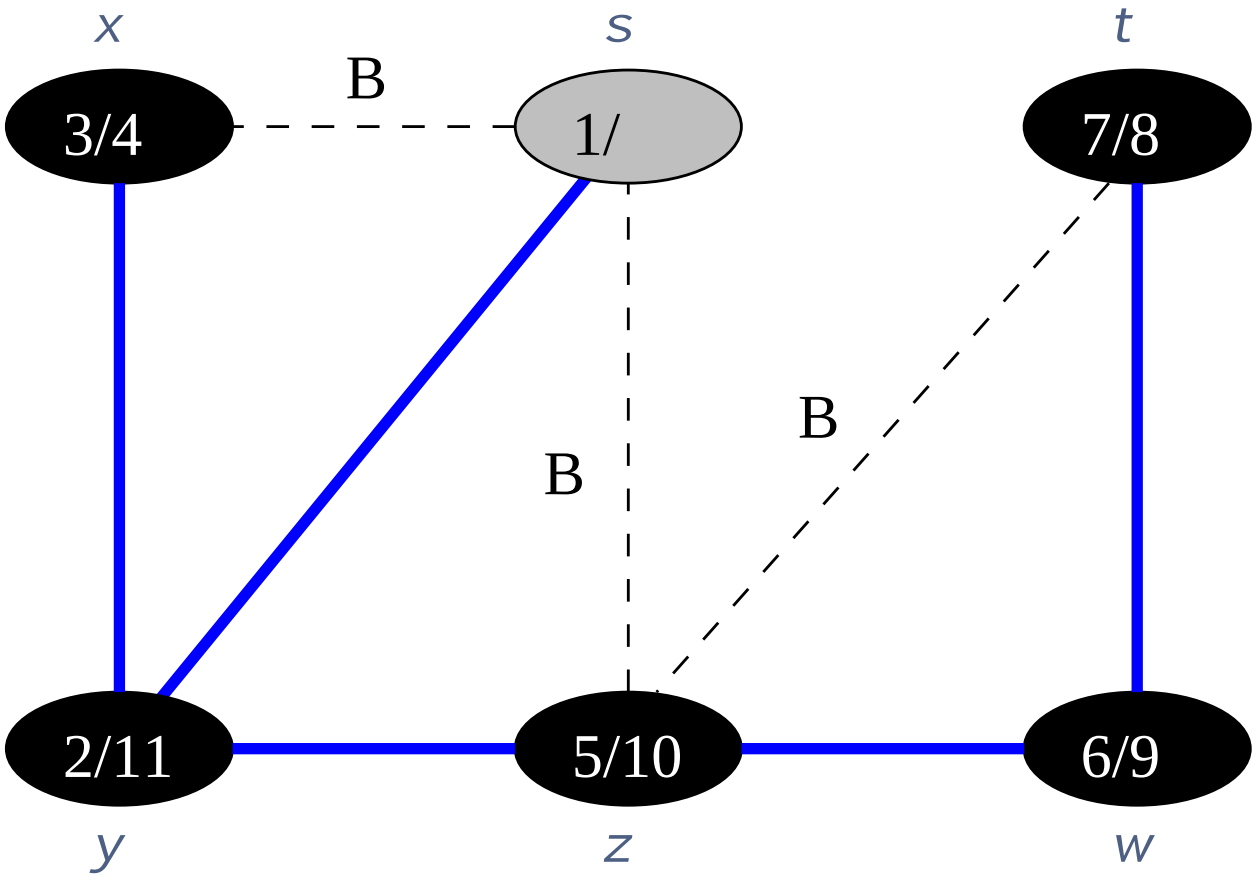
DFS em grafo não direcionado



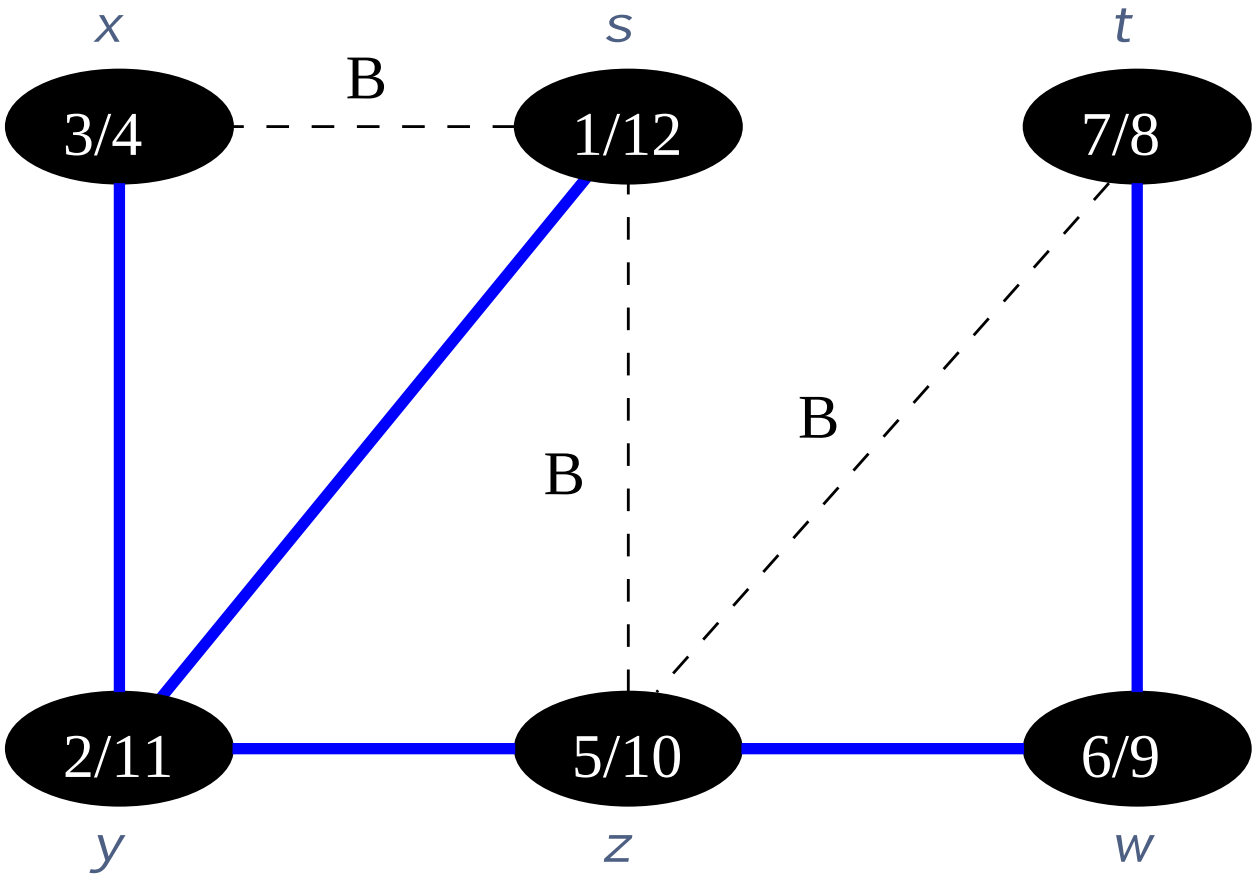
DFS em grafo não direcionado



DFS em grafo não direcionado

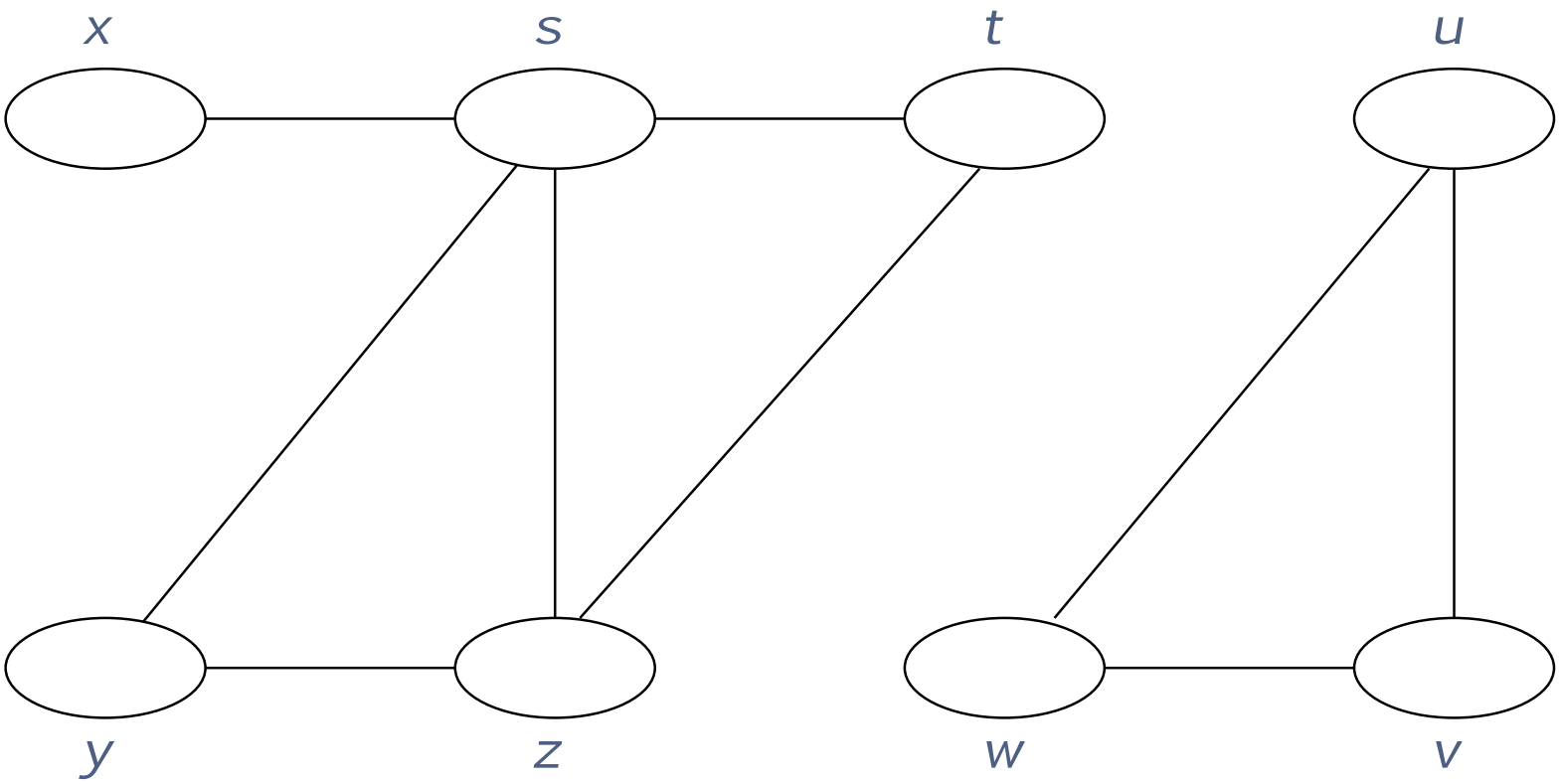


DFS em grafo não direcionado



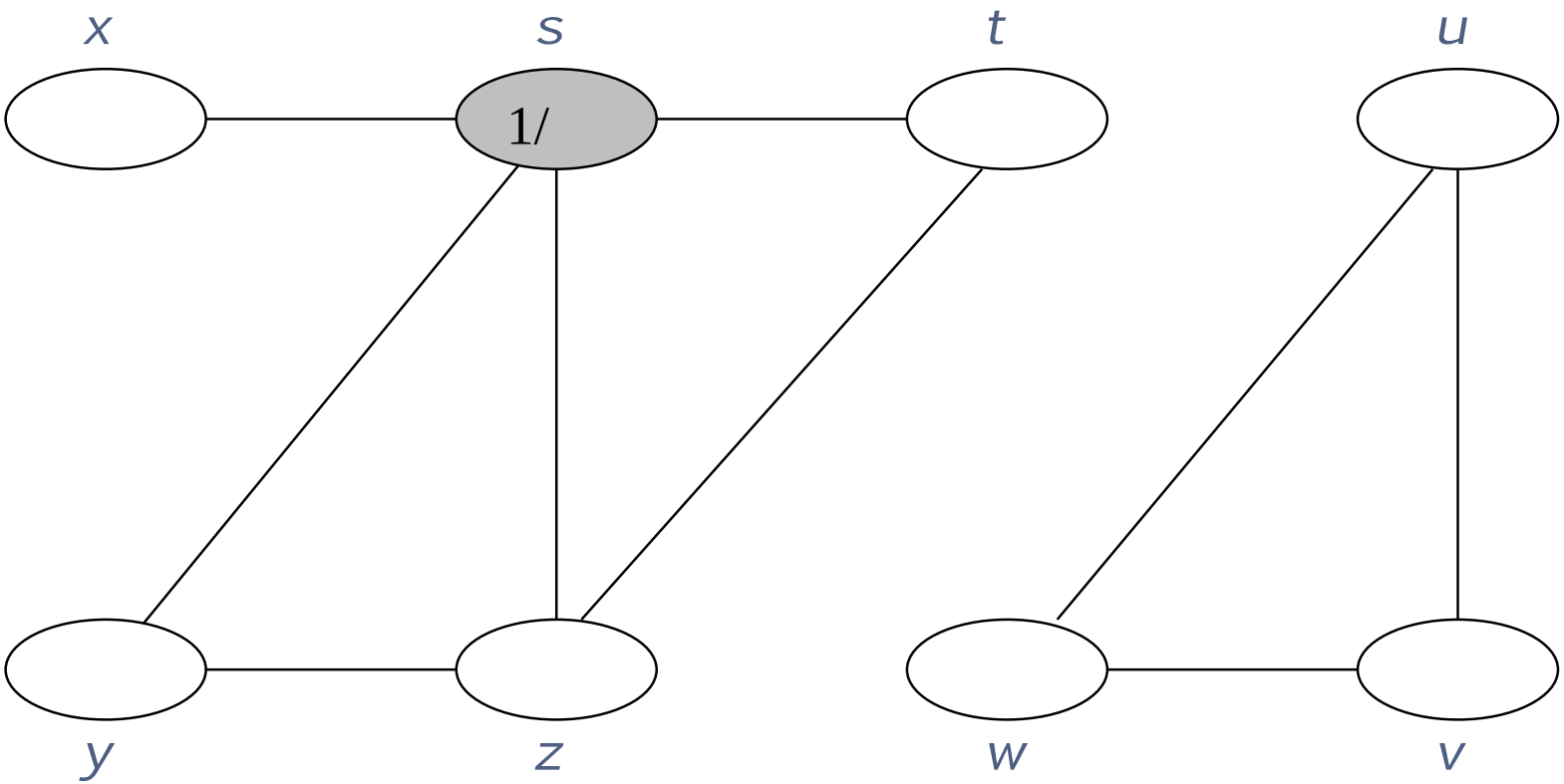
Componentes conexas

Componentes conexas

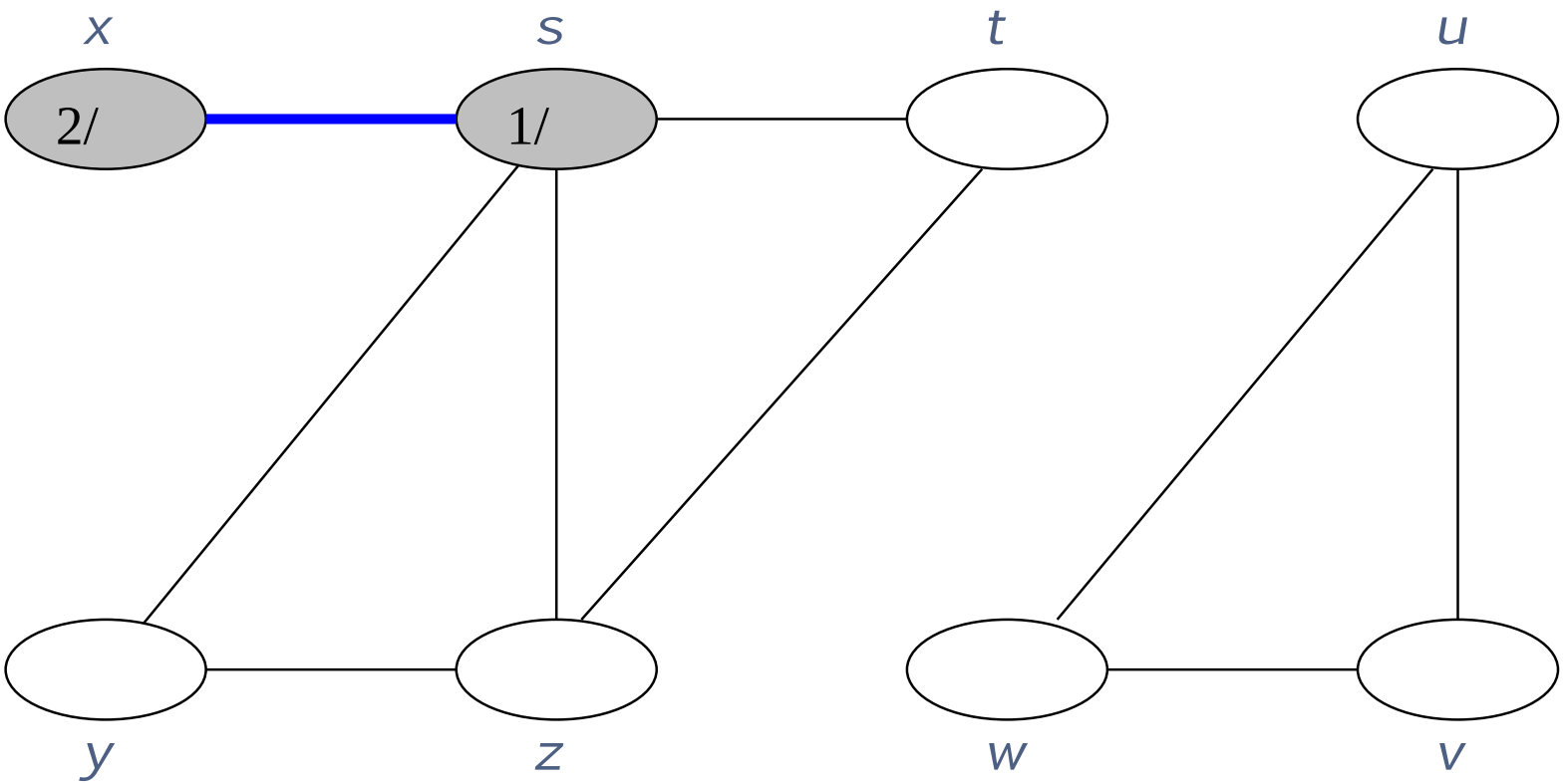


Problema: determinar as componentes conexas de um grafo.

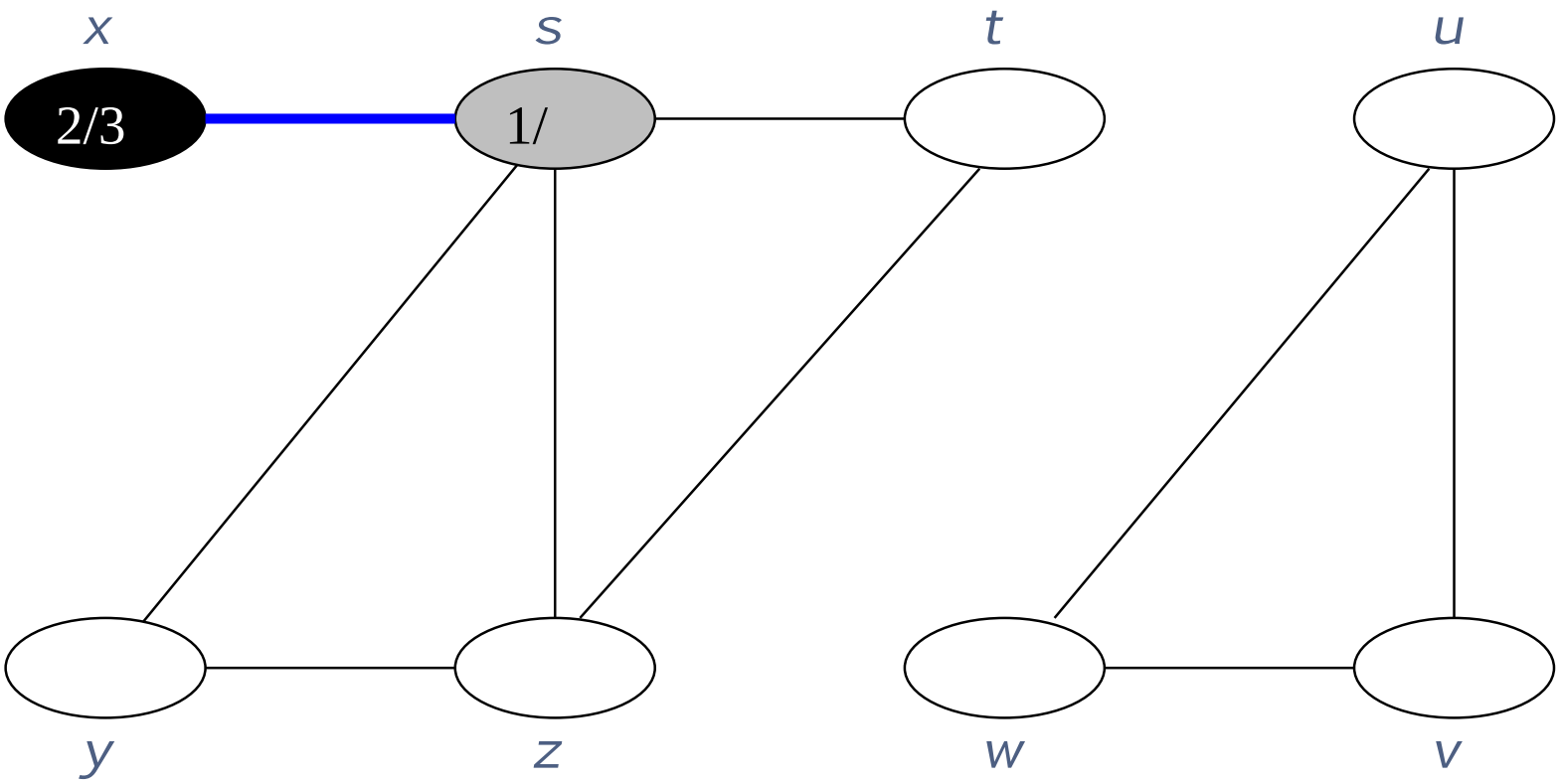
Executando DFS



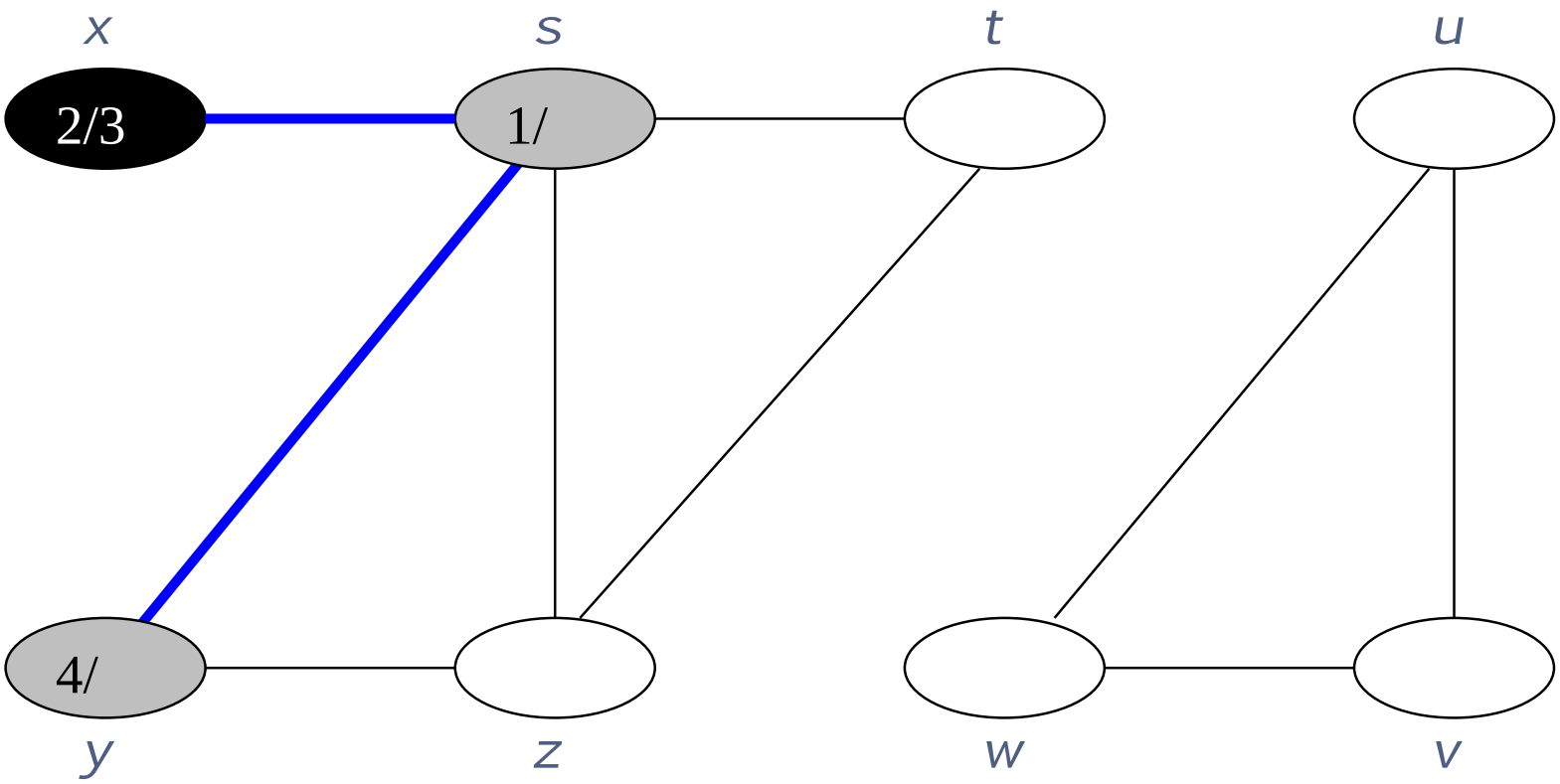
Executando DFS



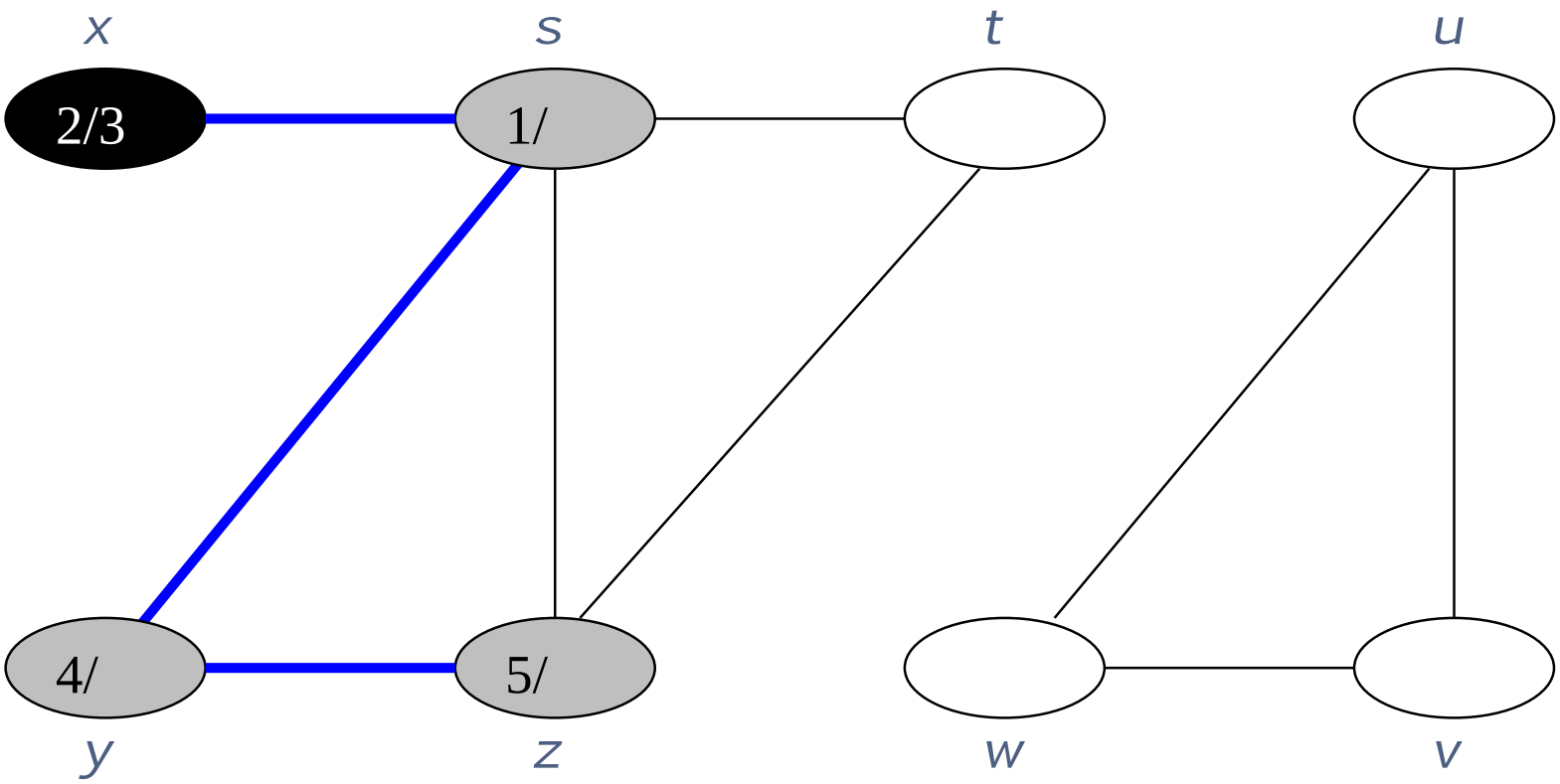
Executando DFS



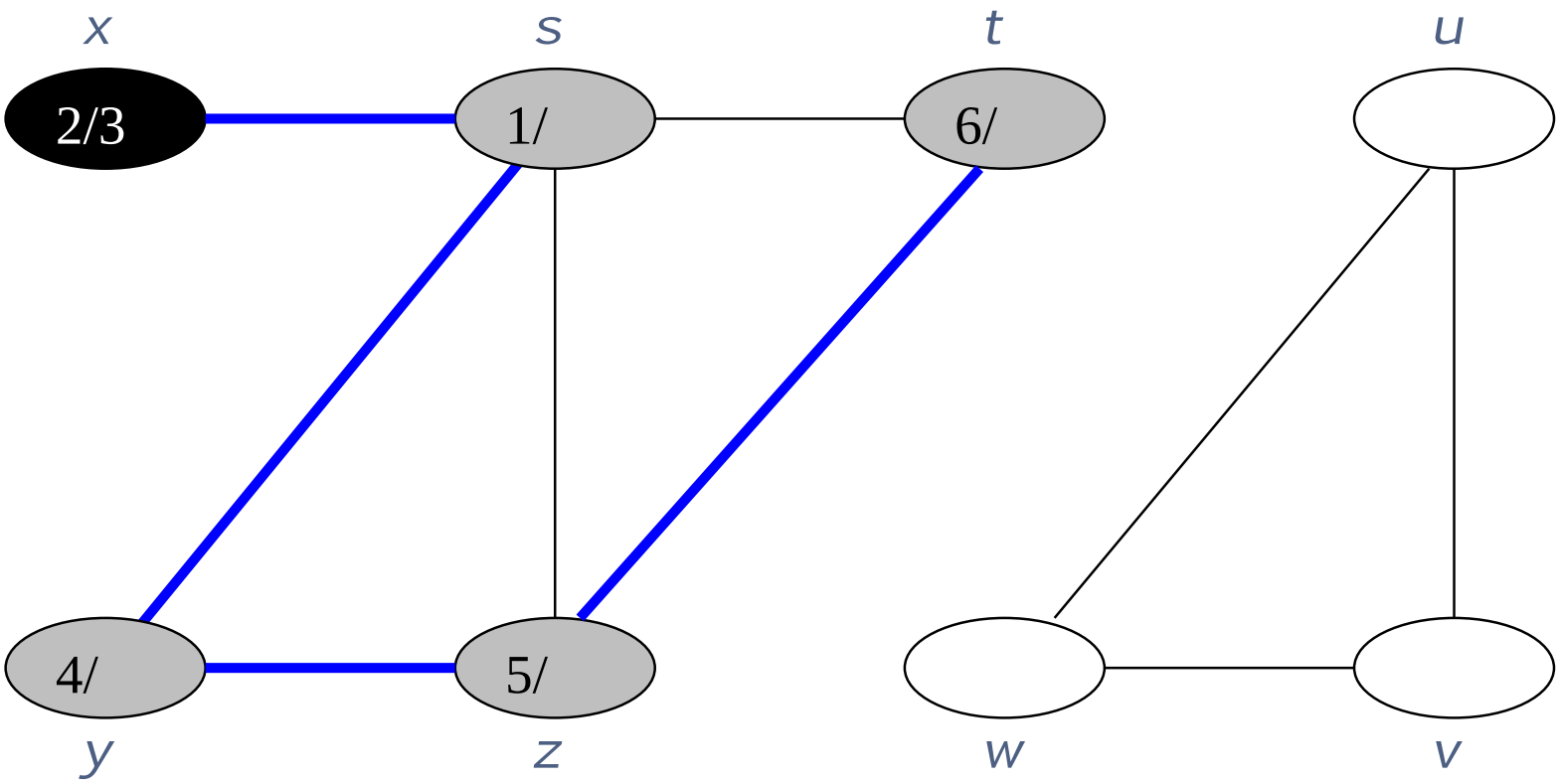
Executando DFS



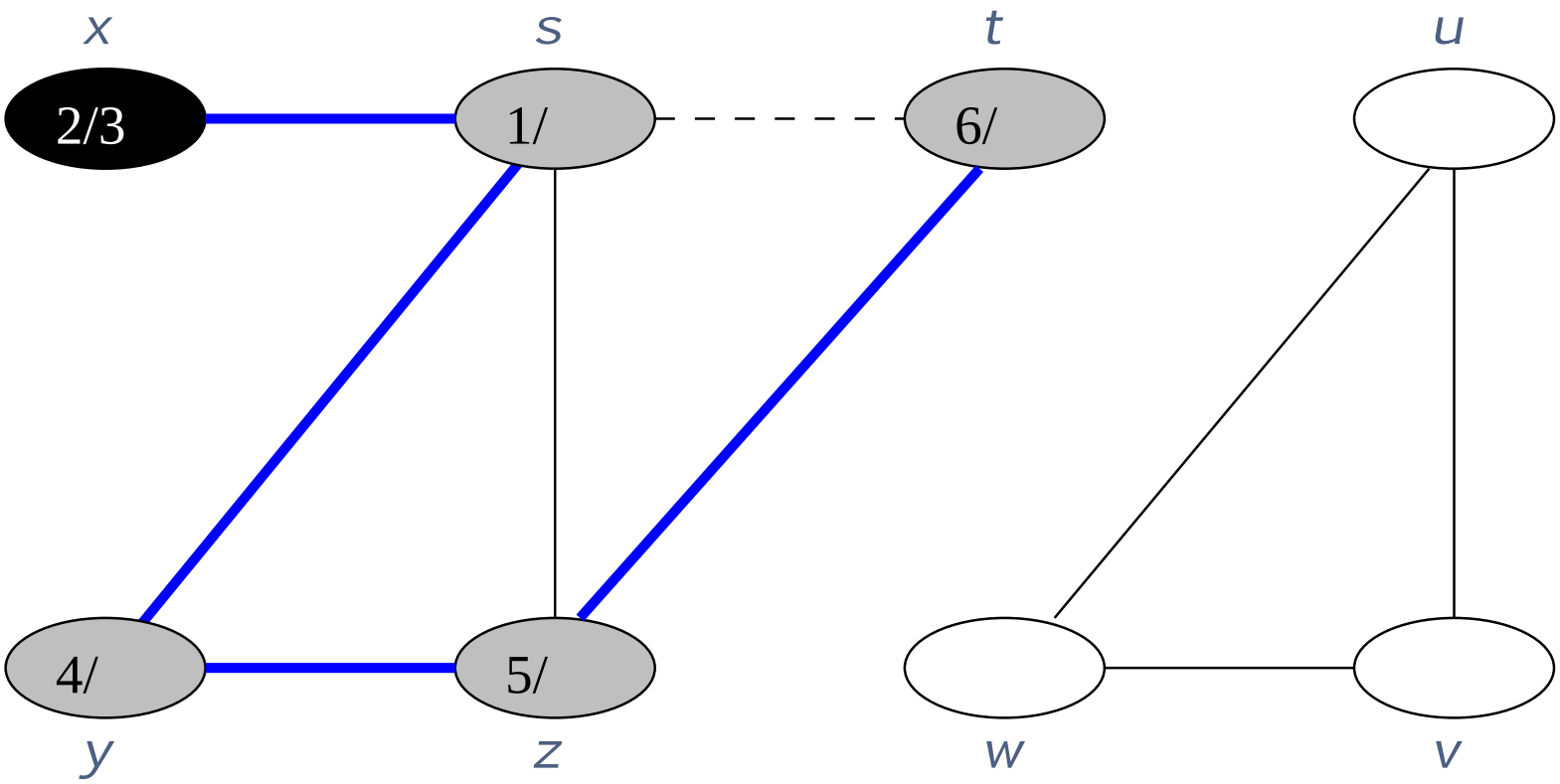
Executando DFS



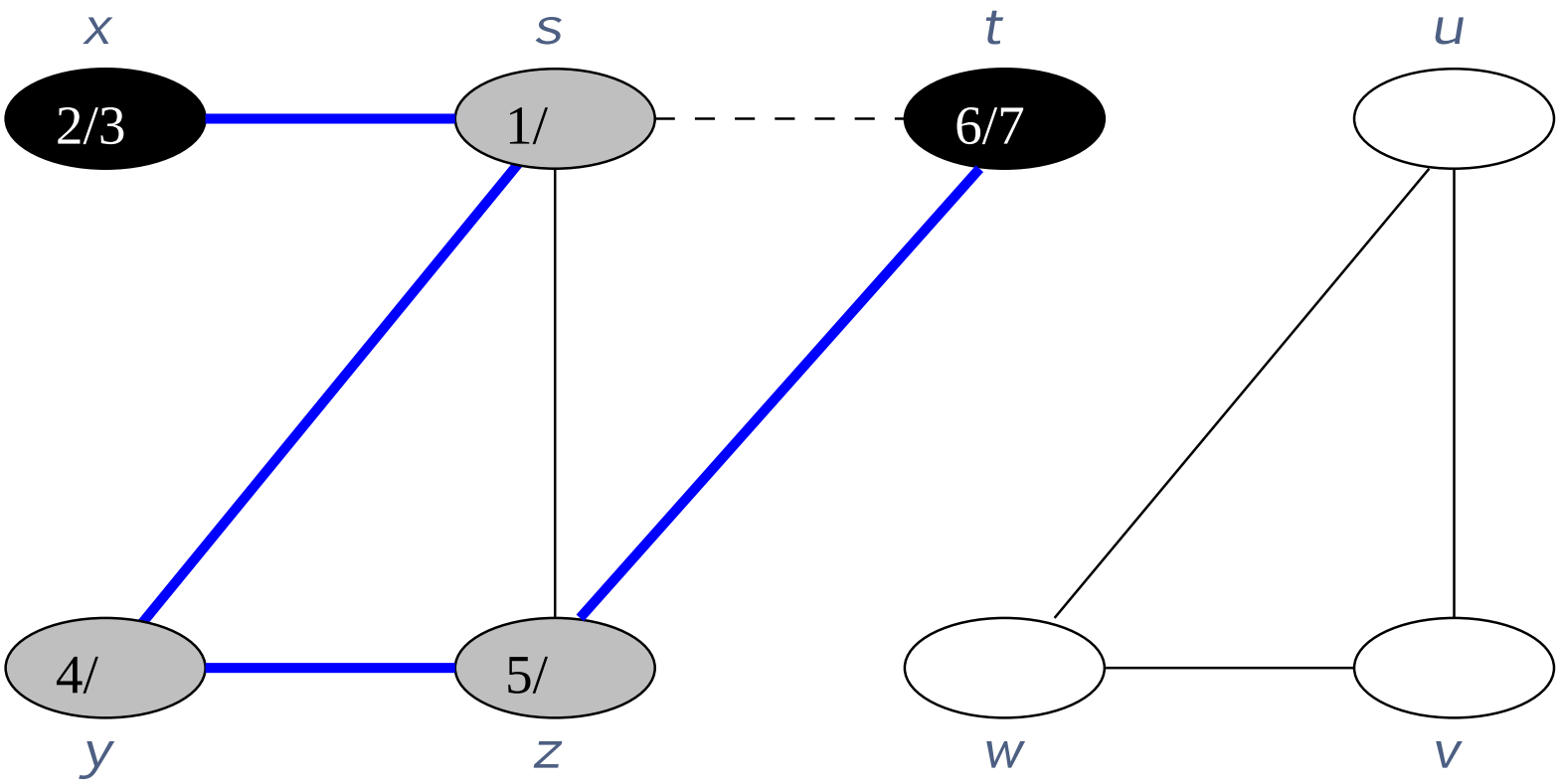
Executando DFS



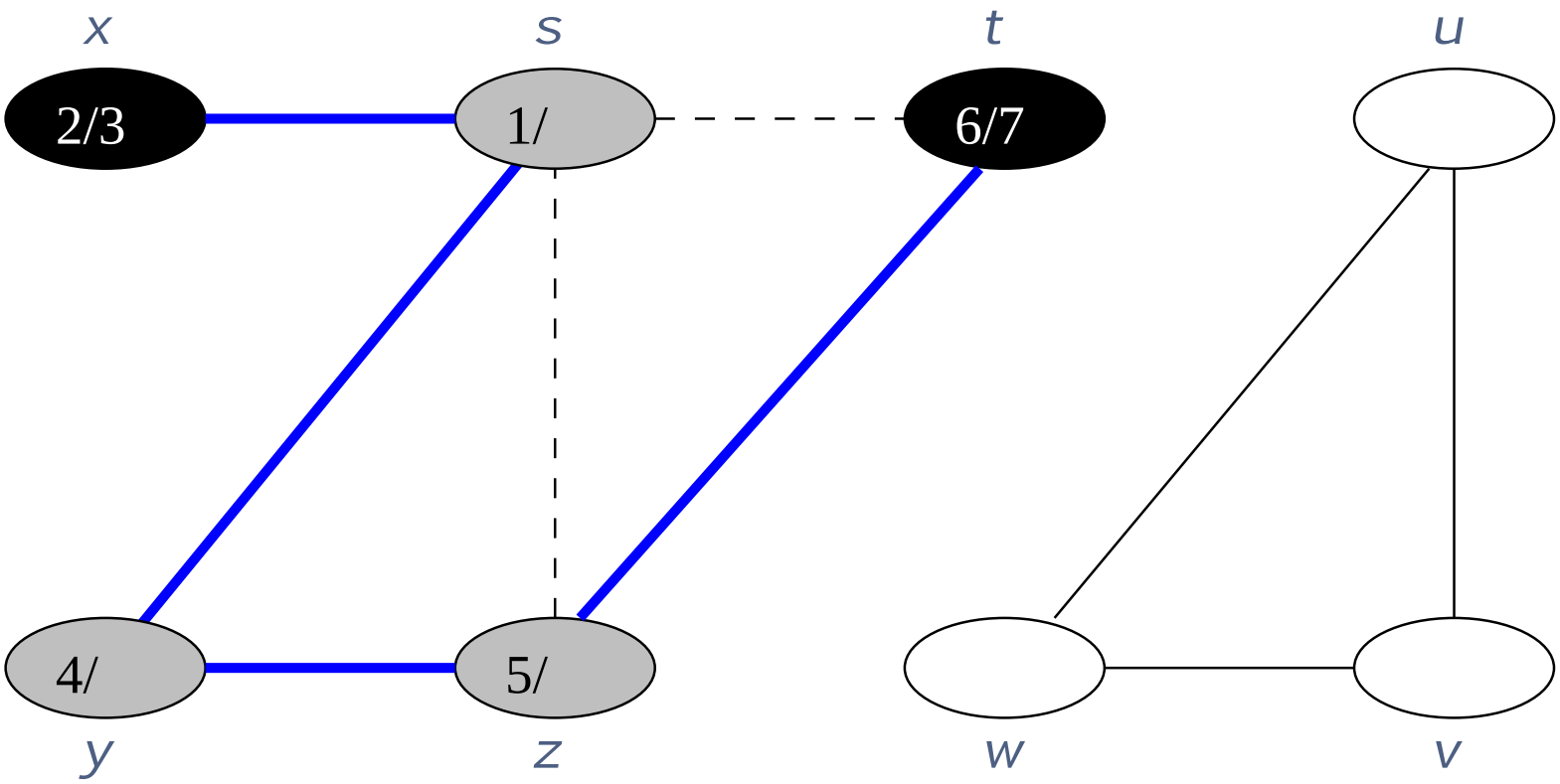
Executando DFS



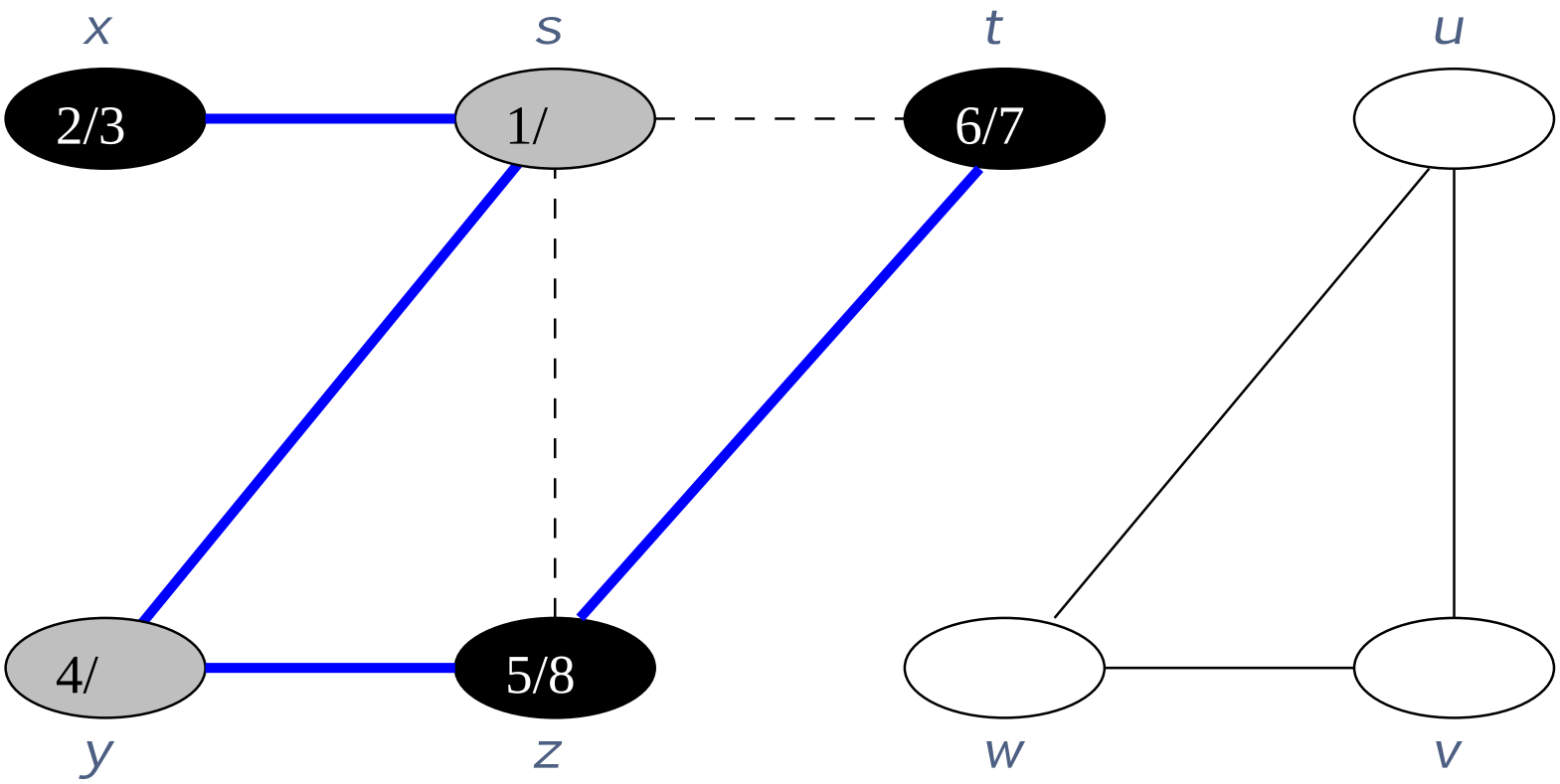
Executando DFS



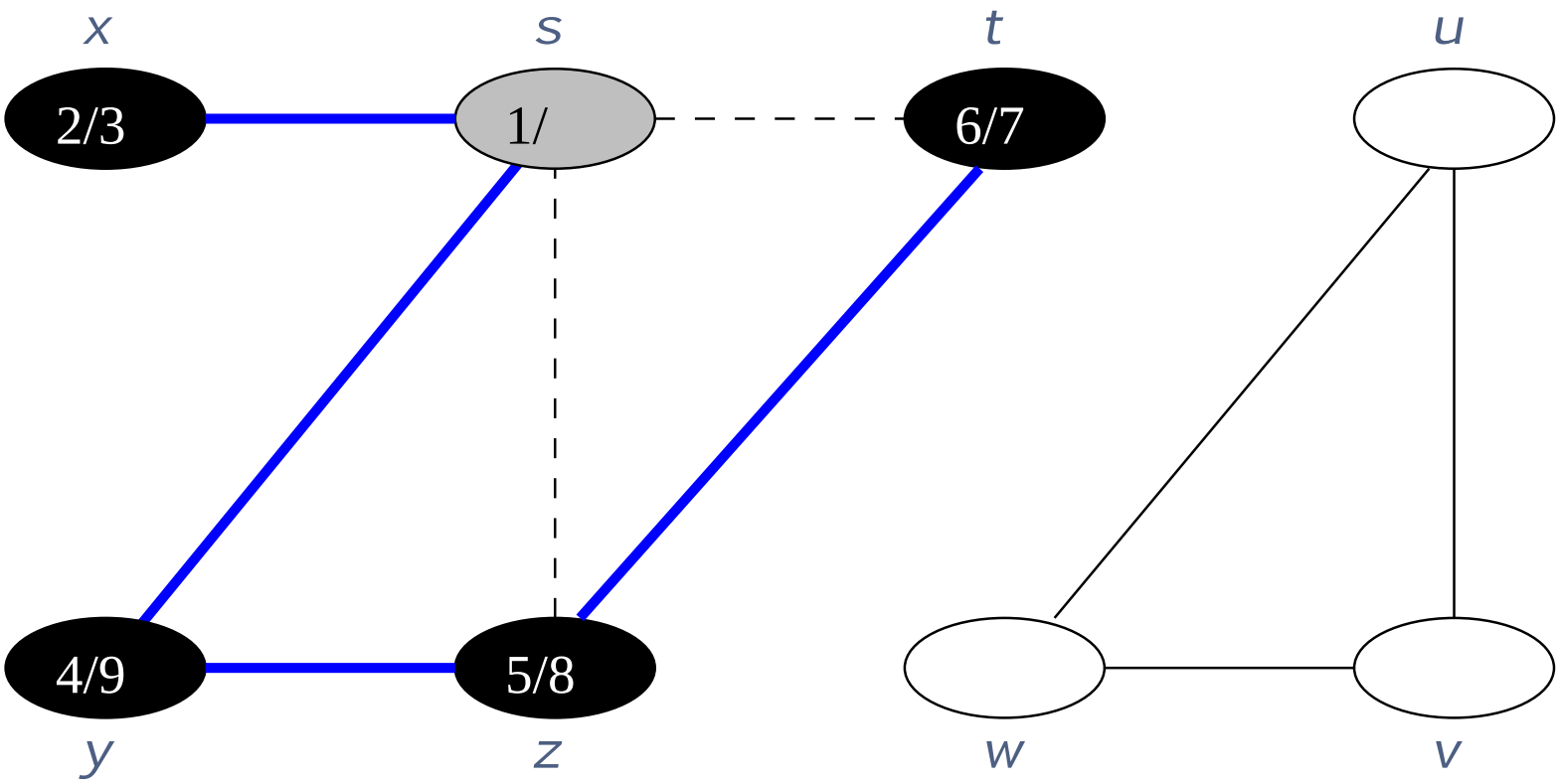
Executando DFS



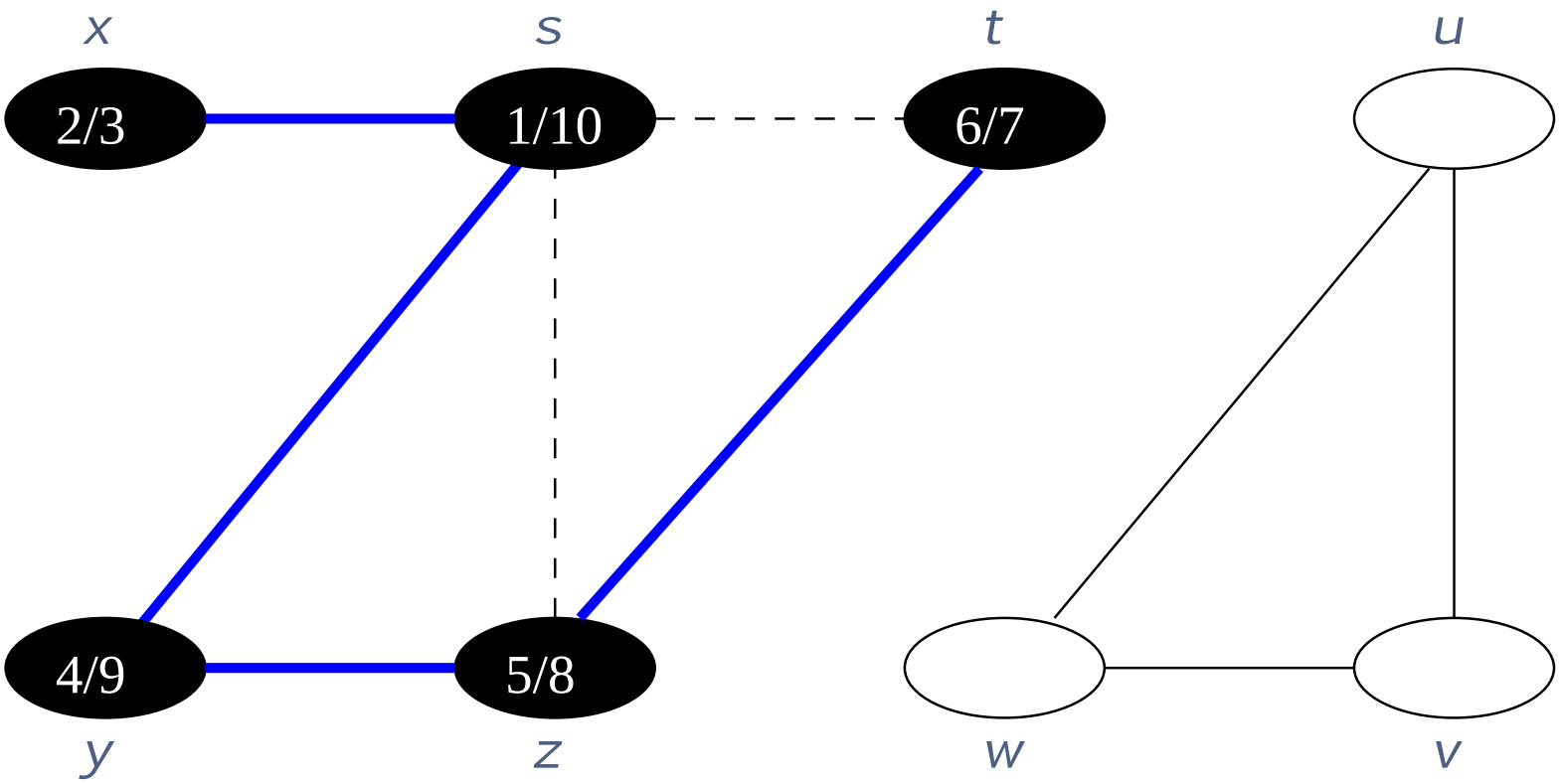
Executando DFS



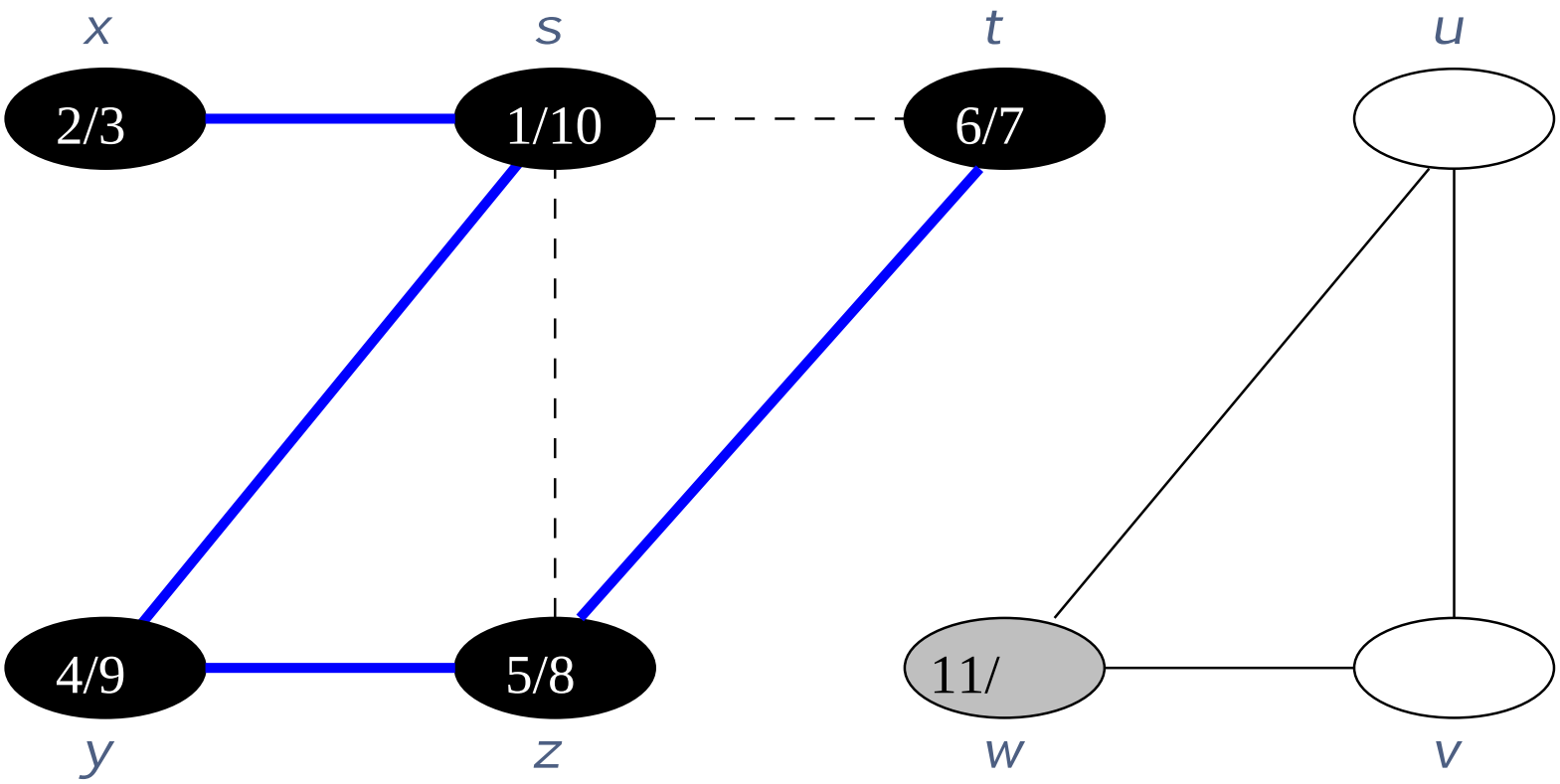
Executando DFS



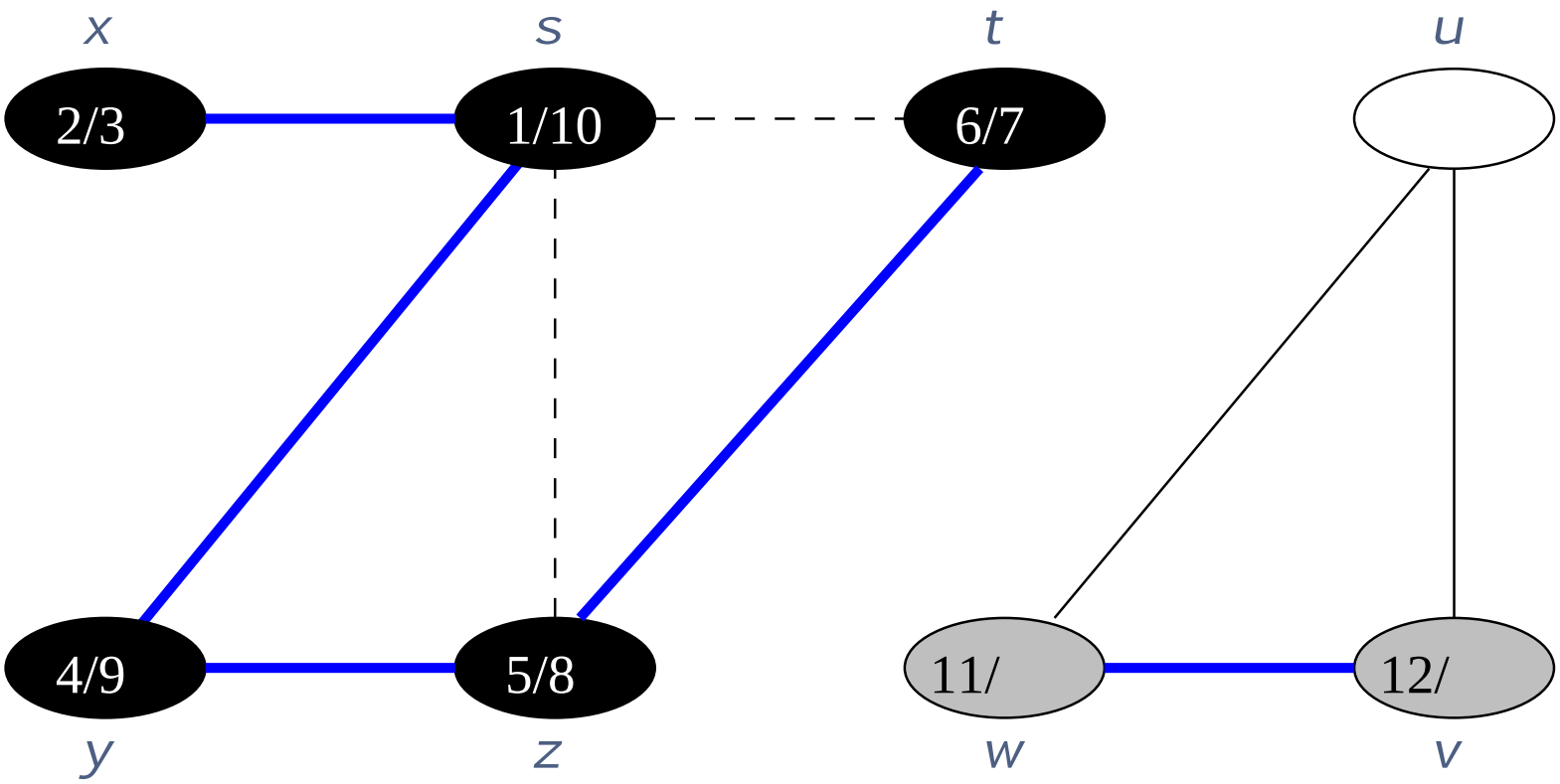
Executando DFS



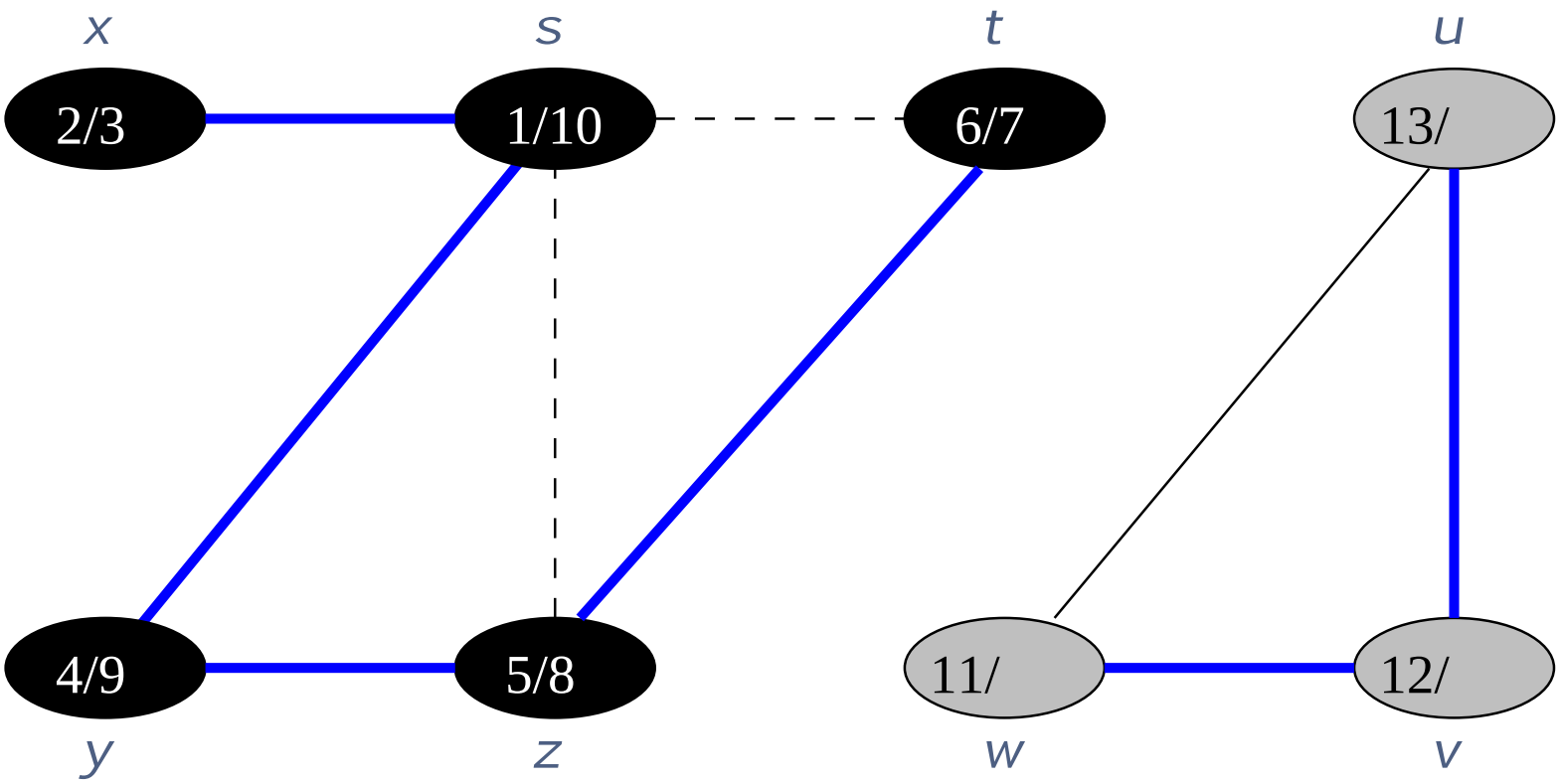
Executando DFS



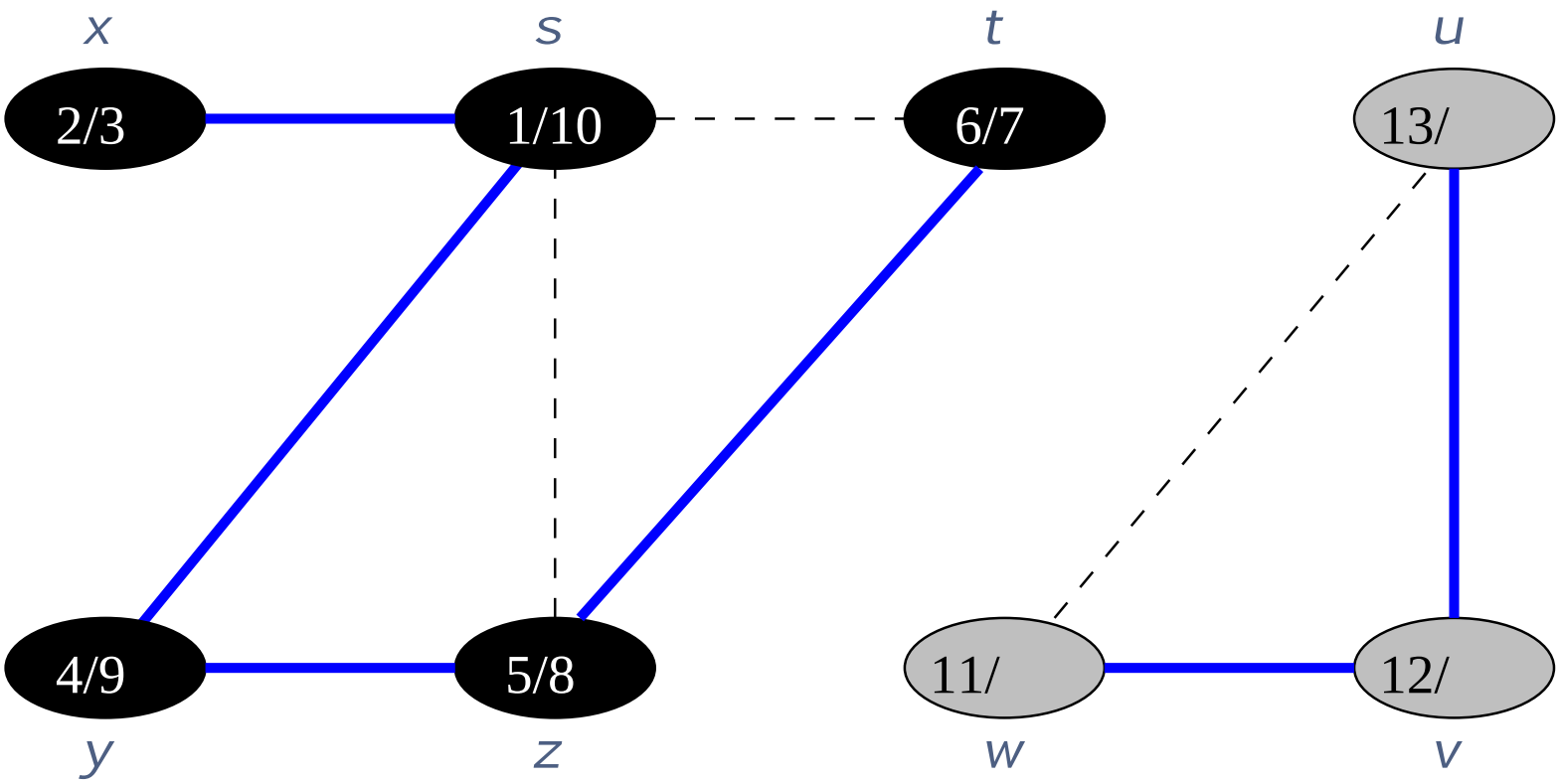
Executando DFS



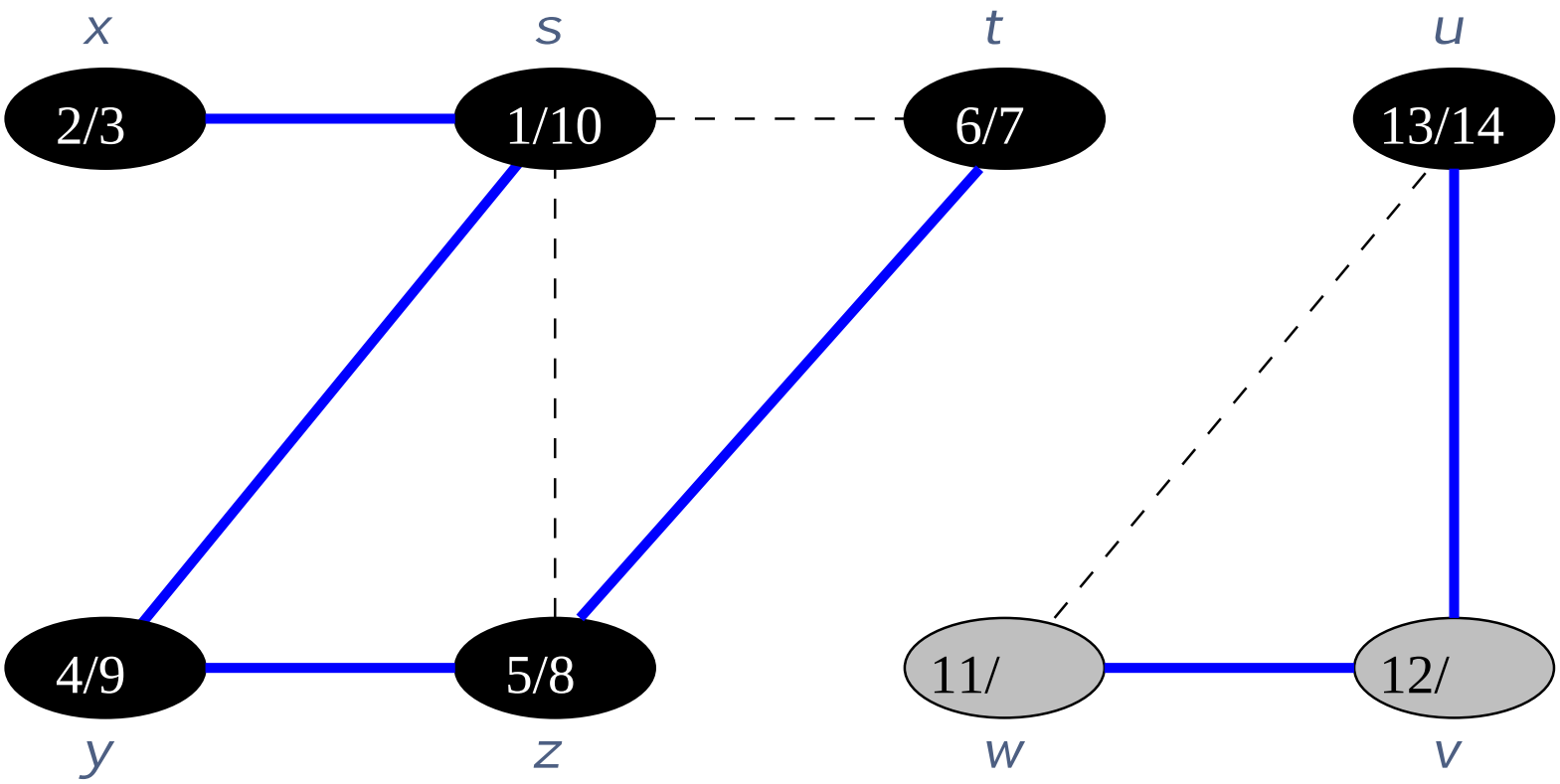
Executando DFS



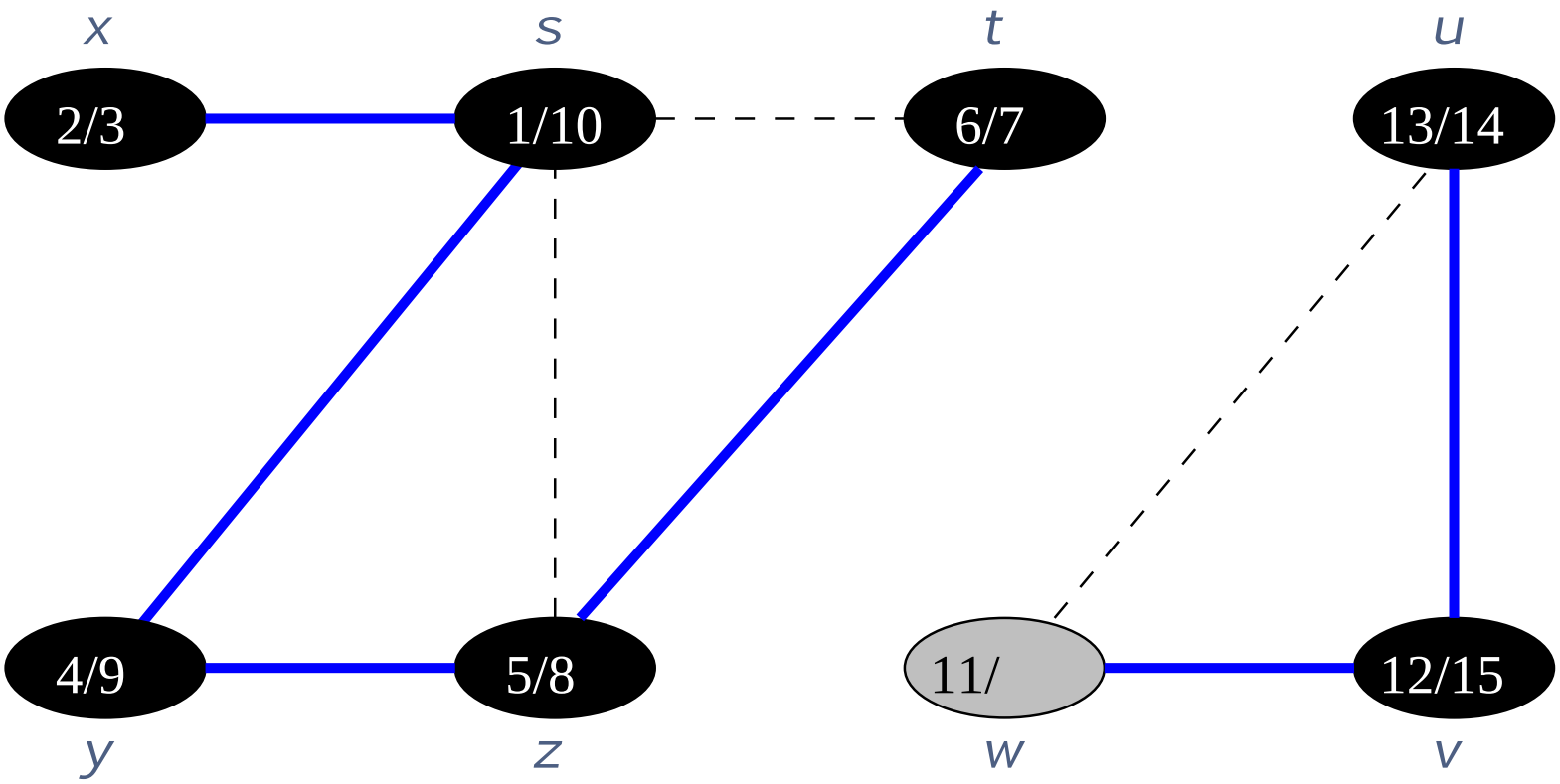
Executando DFS



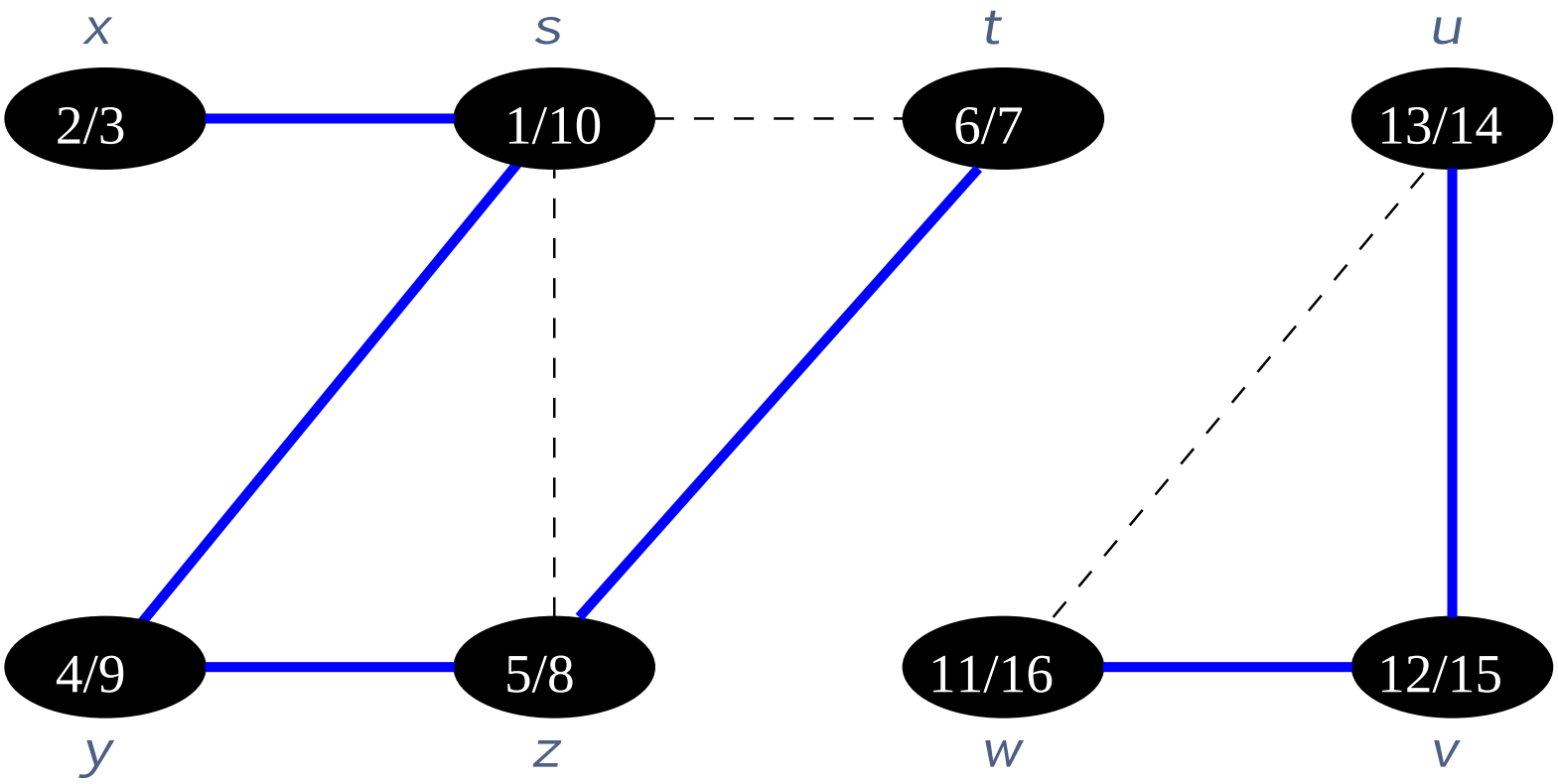
Executando DFS



Executando DFS



Executando DFS



Componentes conexas

Contando o número de componentes

- cada componente corresponde a uma árvore de busca
- é o número de chamadas a DFS-VISIT a partir de DFS

Vamos modificar DFS

- identificamos cada componente por um número
- denotaremos por $comp[v]$ a componente de v

Algoritmo DFS modificado

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

- ℓ é o número de chamadas a DFS-VISIT a partir de DFS

Algoritmo DFS-VISIT modificado

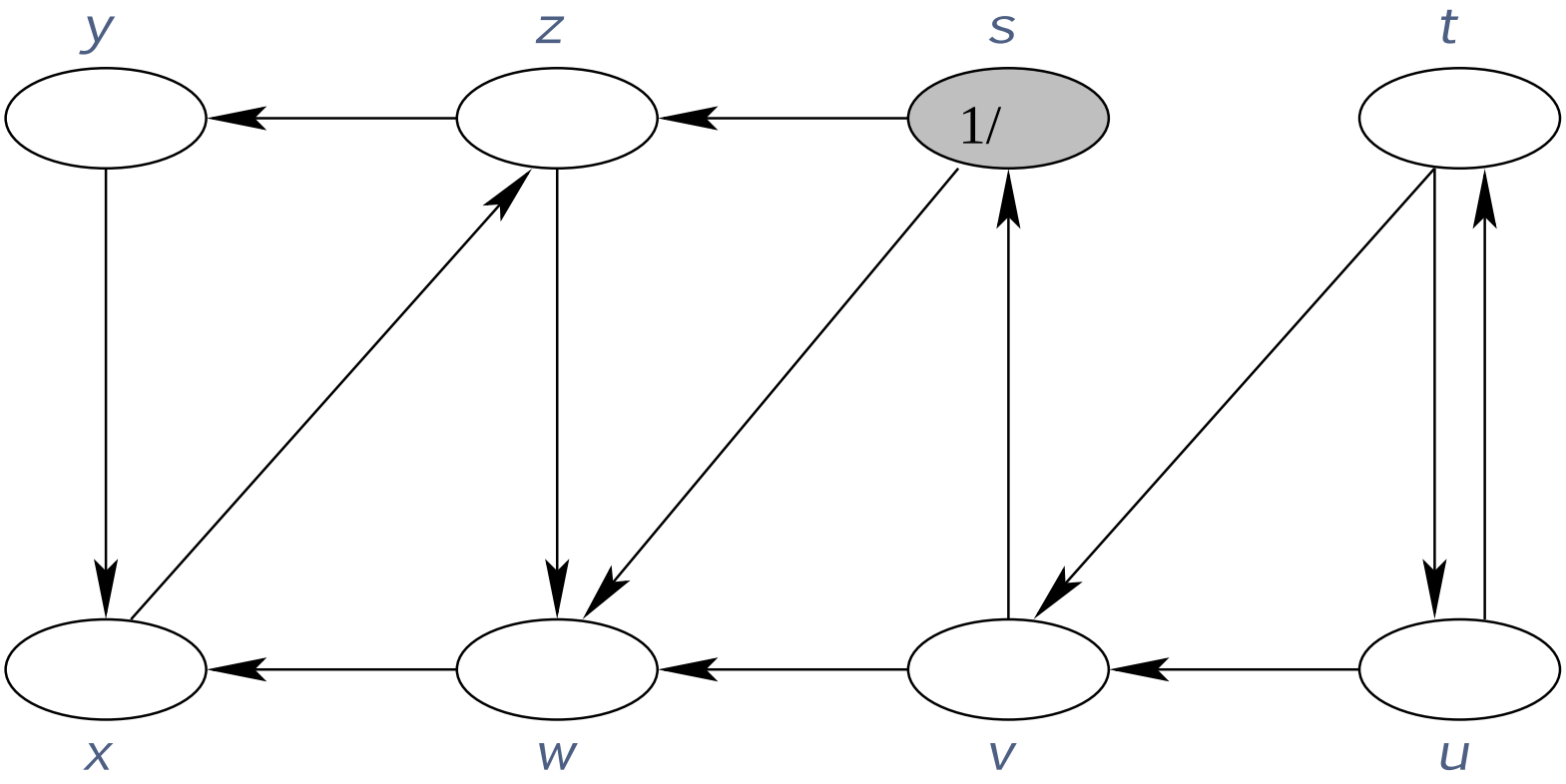
```
DFS-VISIT(u)
1   cor[u] ← cinza
2   para cada v ∈ Adj[u] faça
3       se cor[v] = branco então
4           DFS-VISIT(v)
5   cor[u] ← preto
6   comp[u] ← ℓ
```

Ordenação Topológica

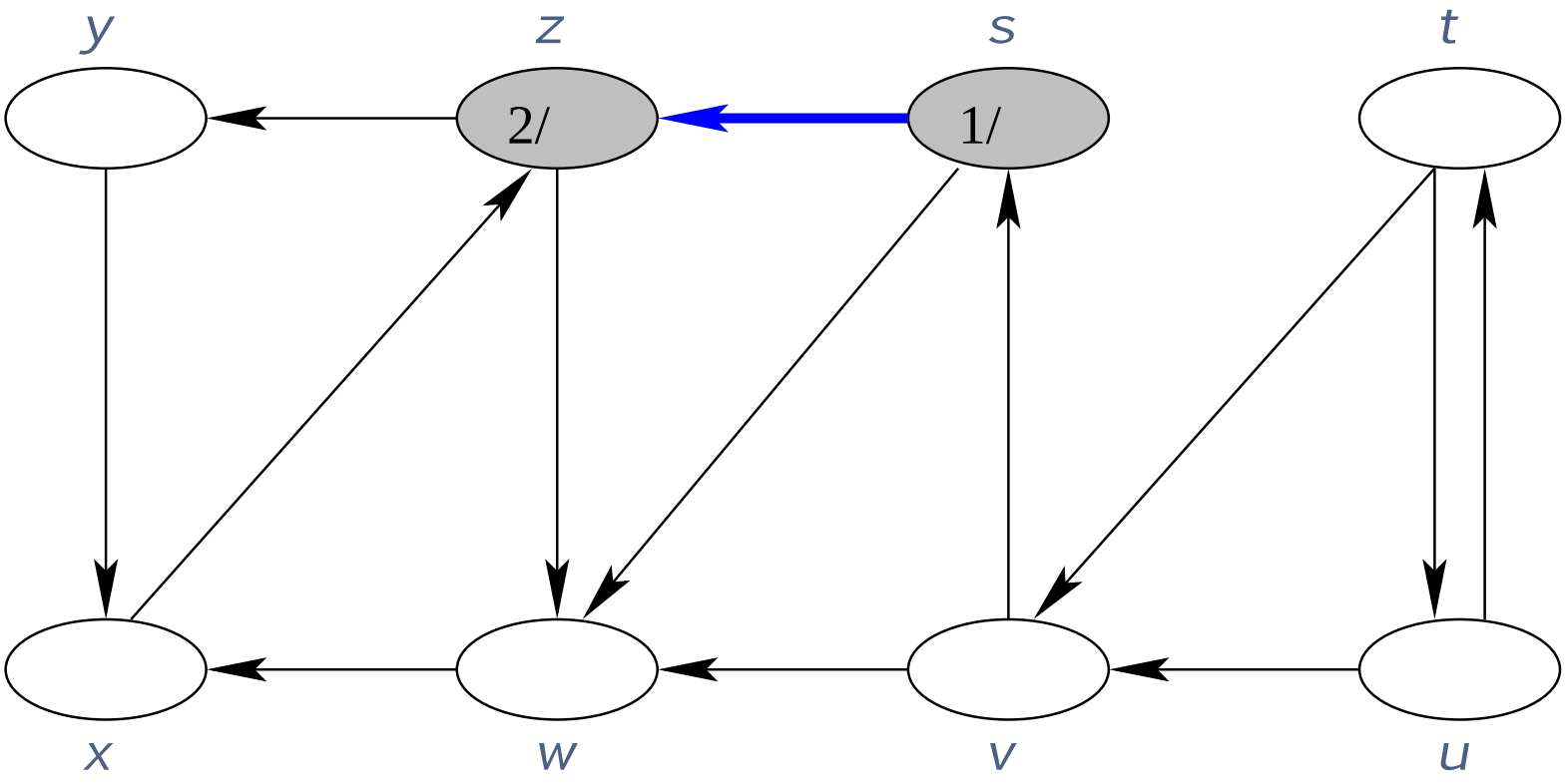
Antes de começar

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

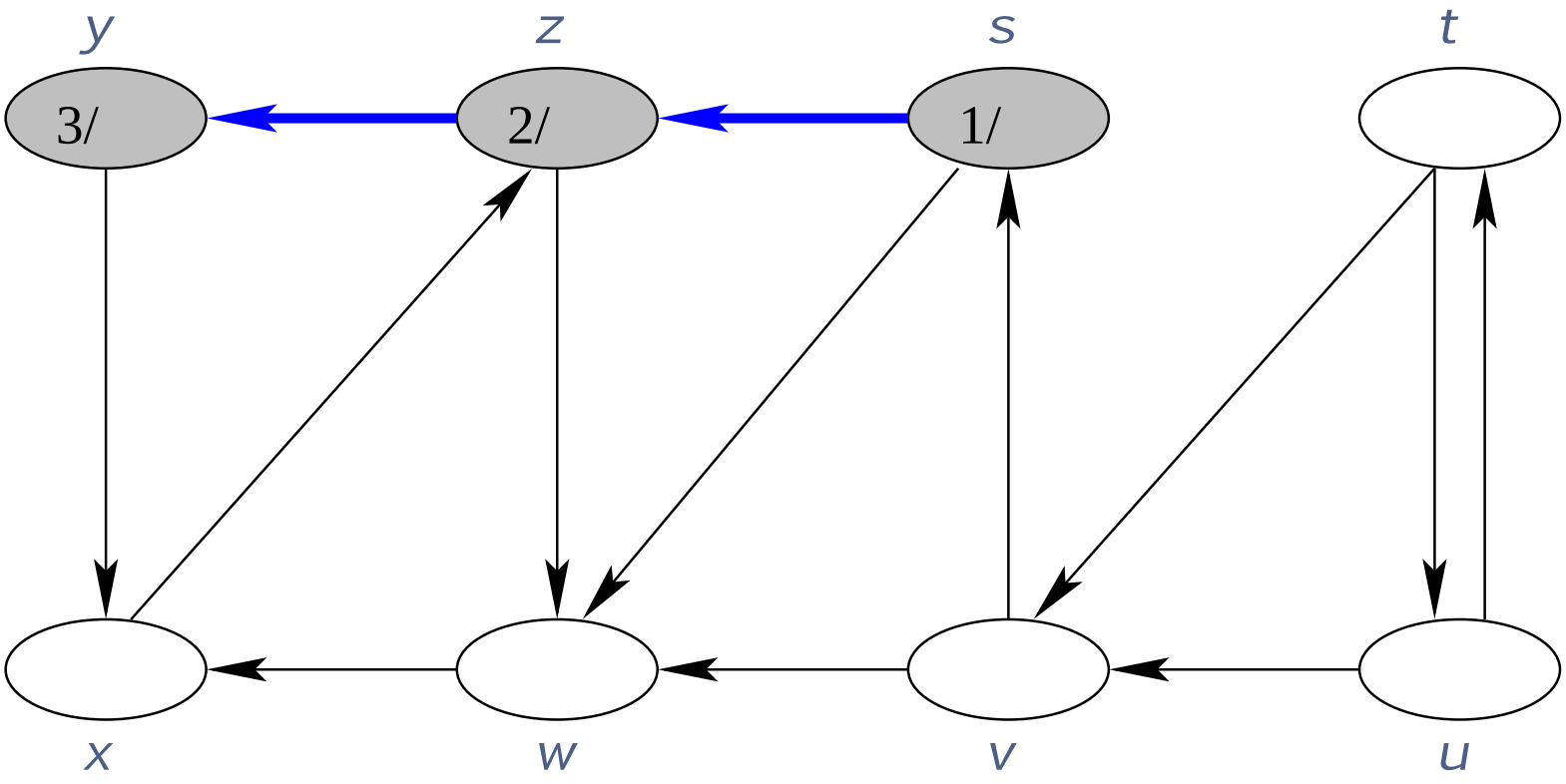
Exemplo de busca em profundidade



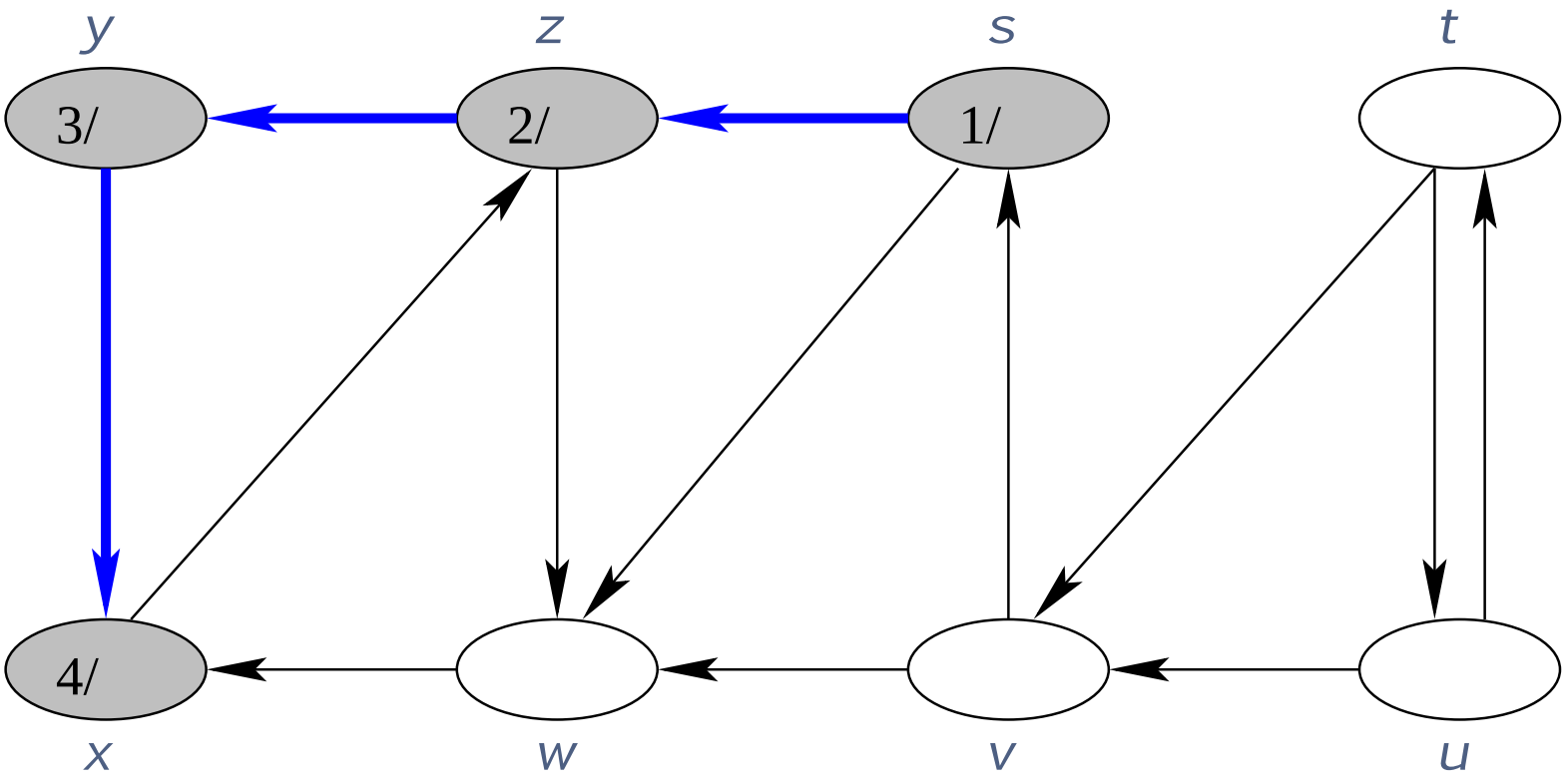
Exemplo de busca em profundidade



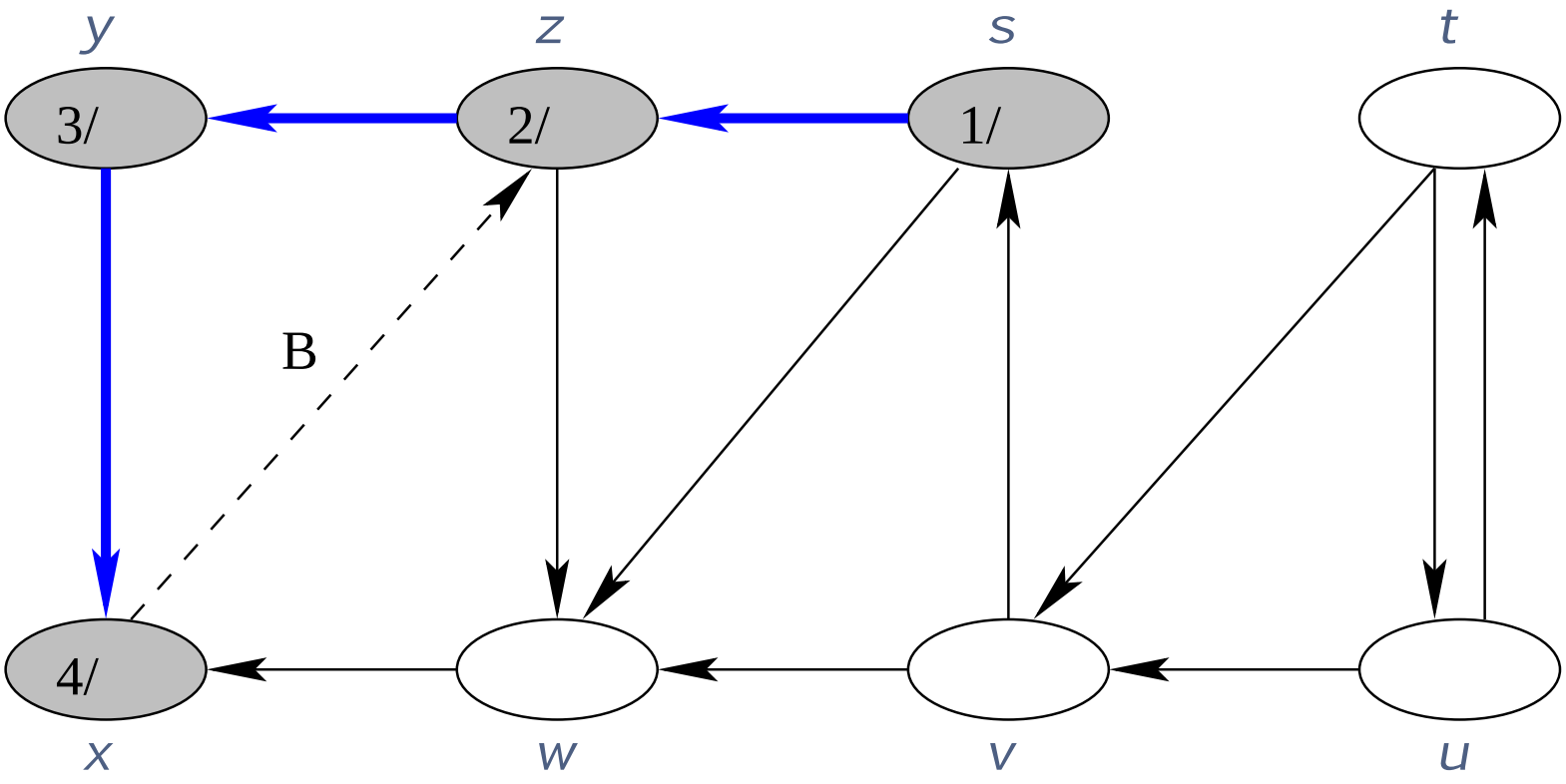
Exemplo de busca em profundidade



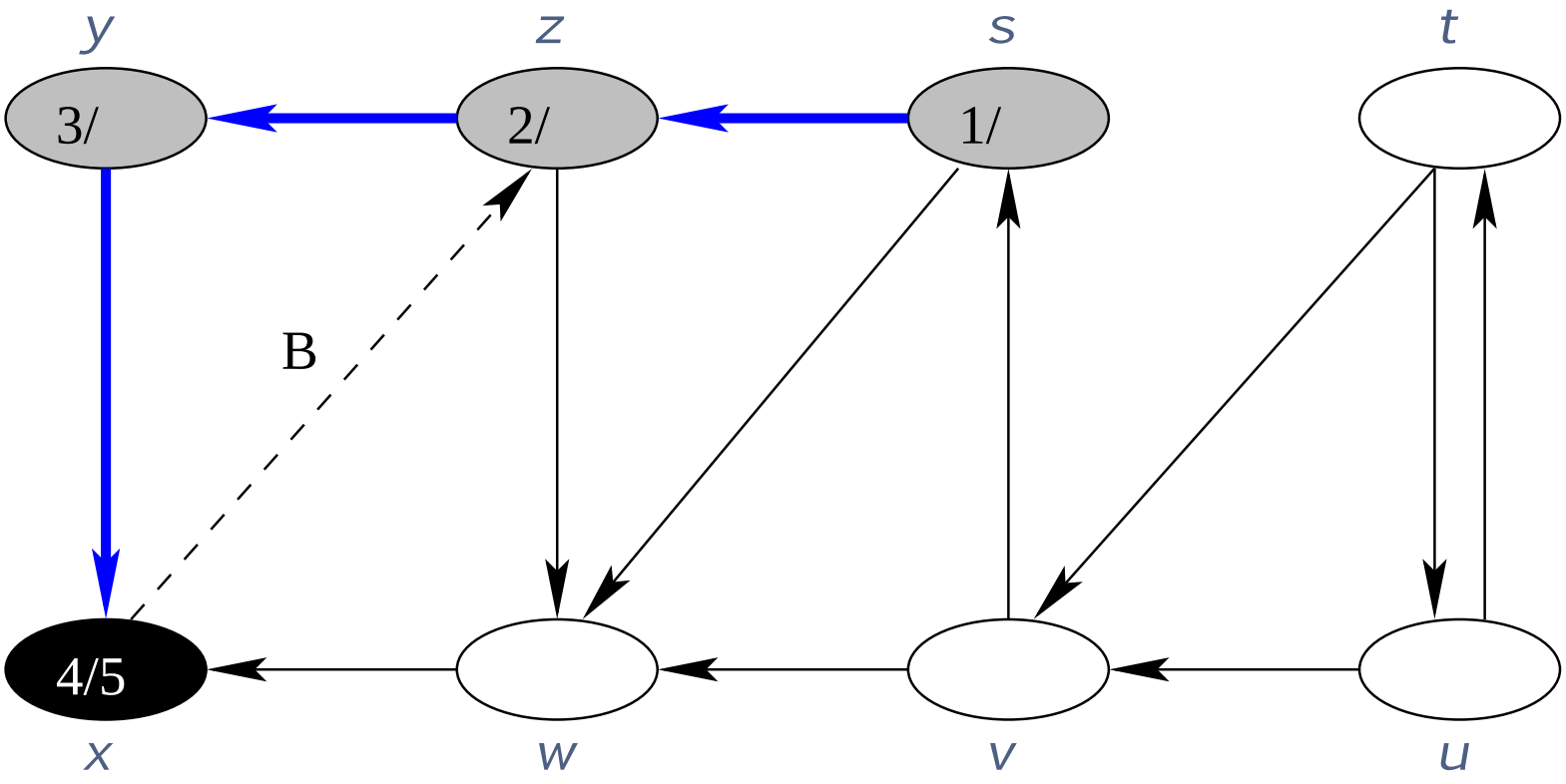
Exemplo de busca em profundidade



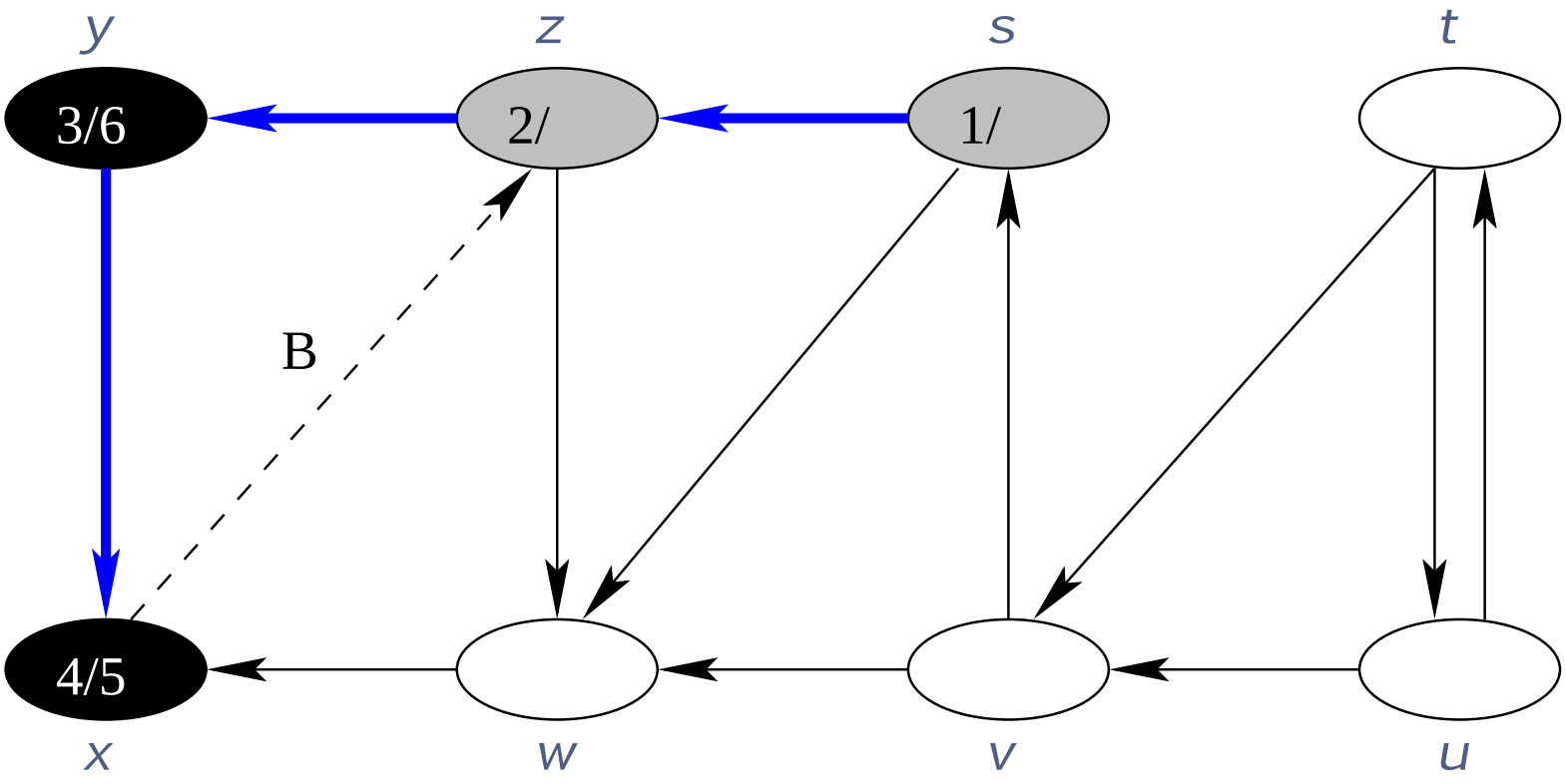
Exemplo de busca em profundidade



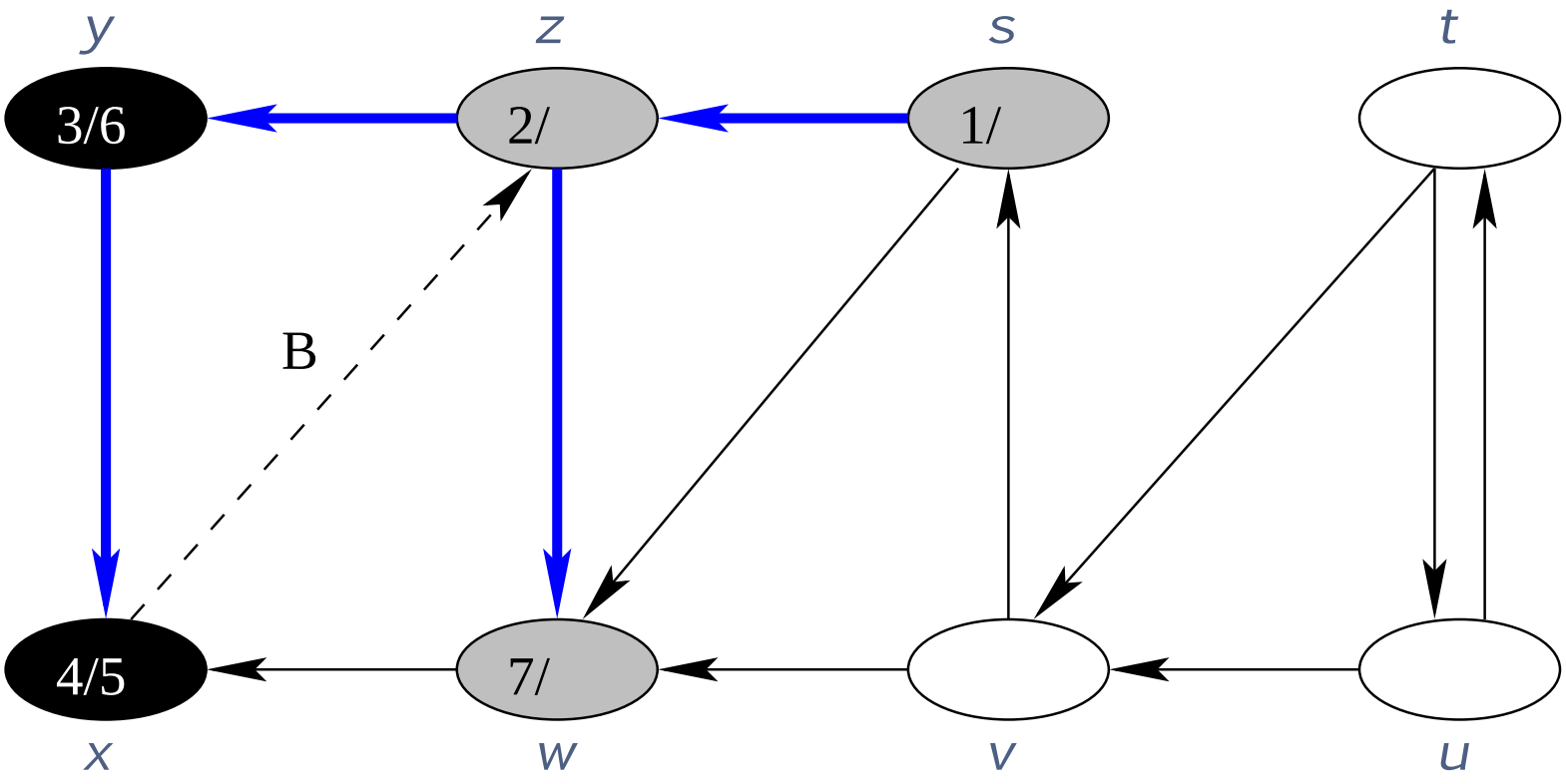
Exemplo de busca em profundidade



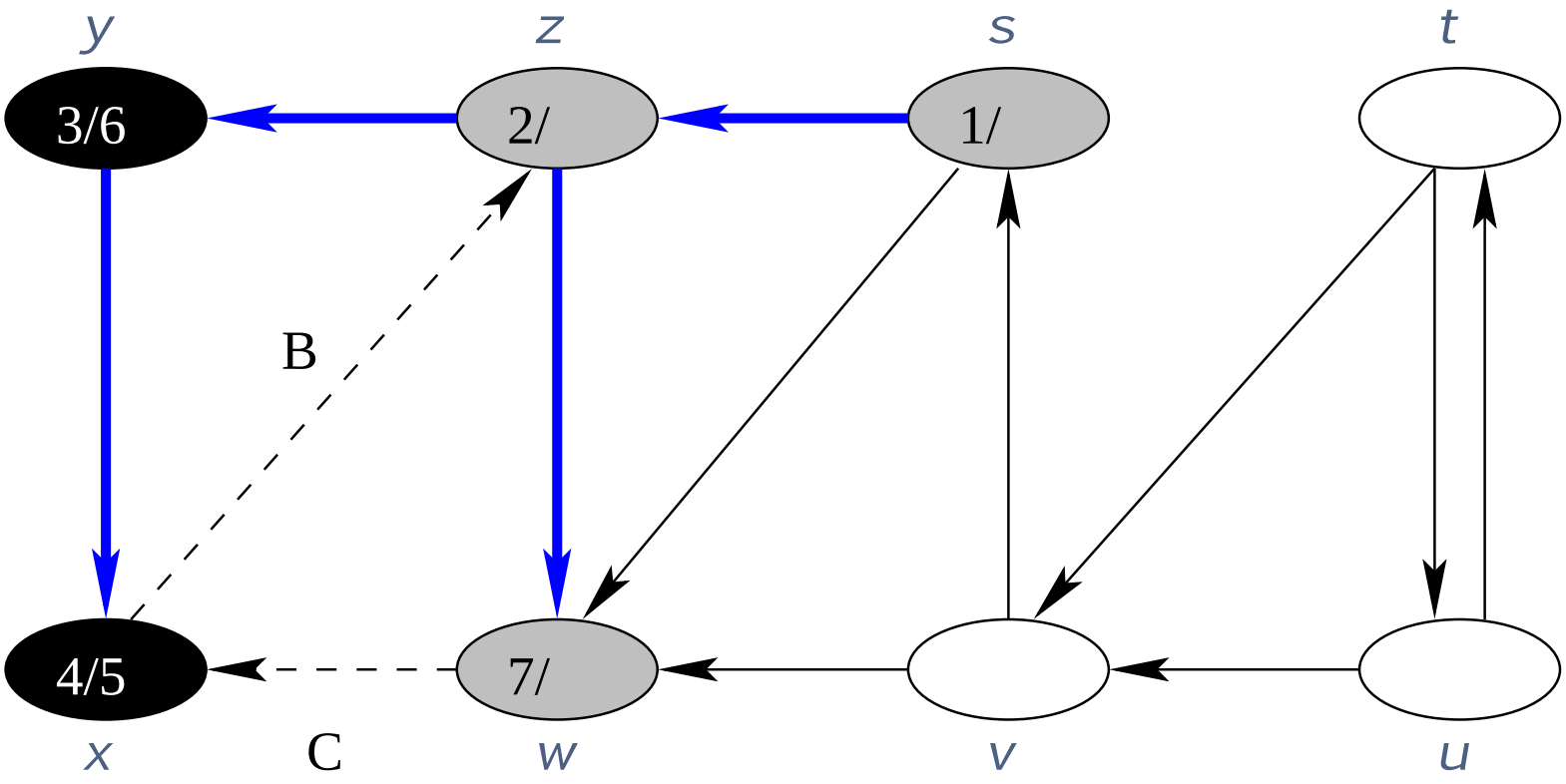
Exemplo de busca em profundidade



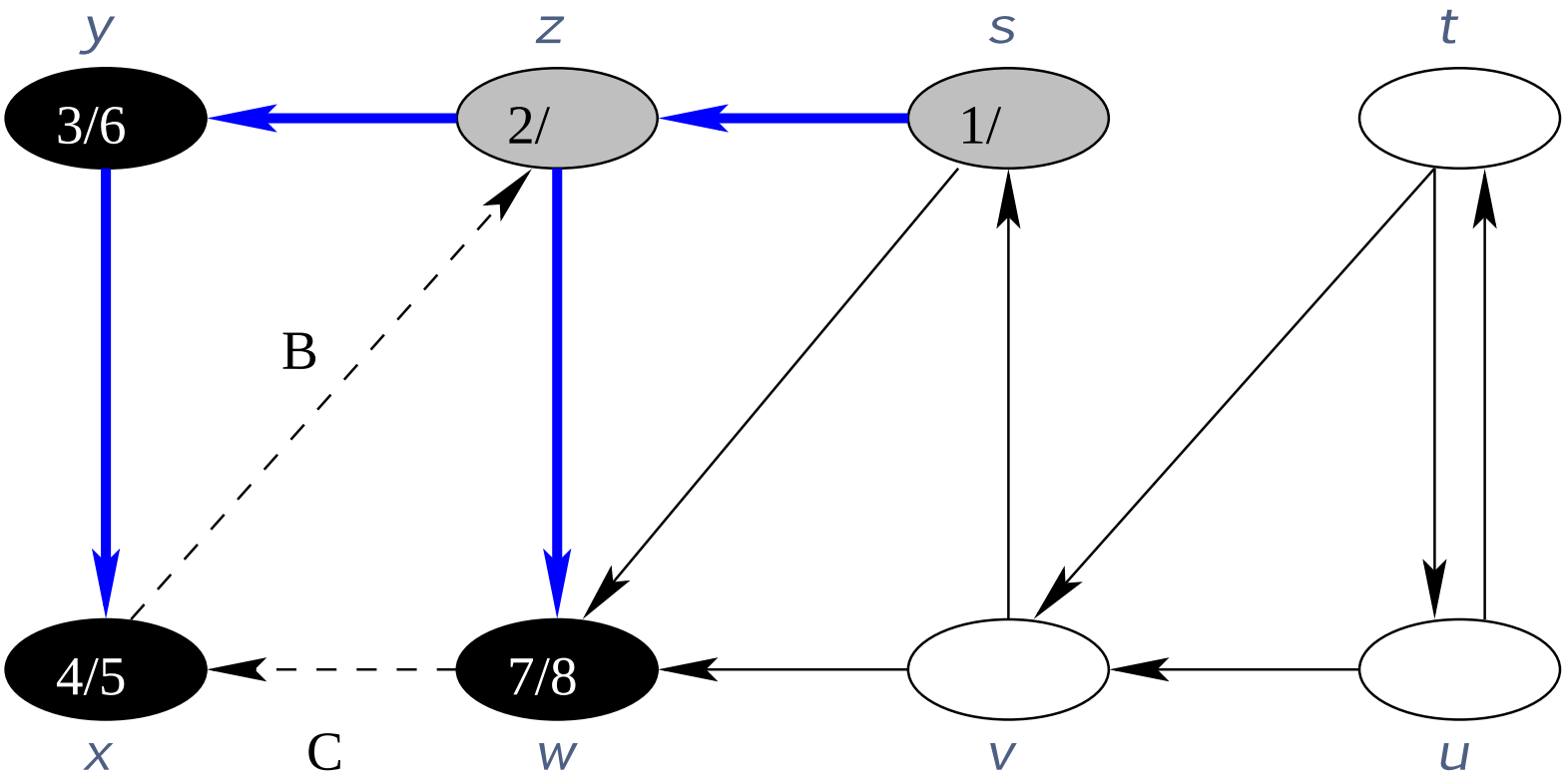
Exemplo de busca em profundidade



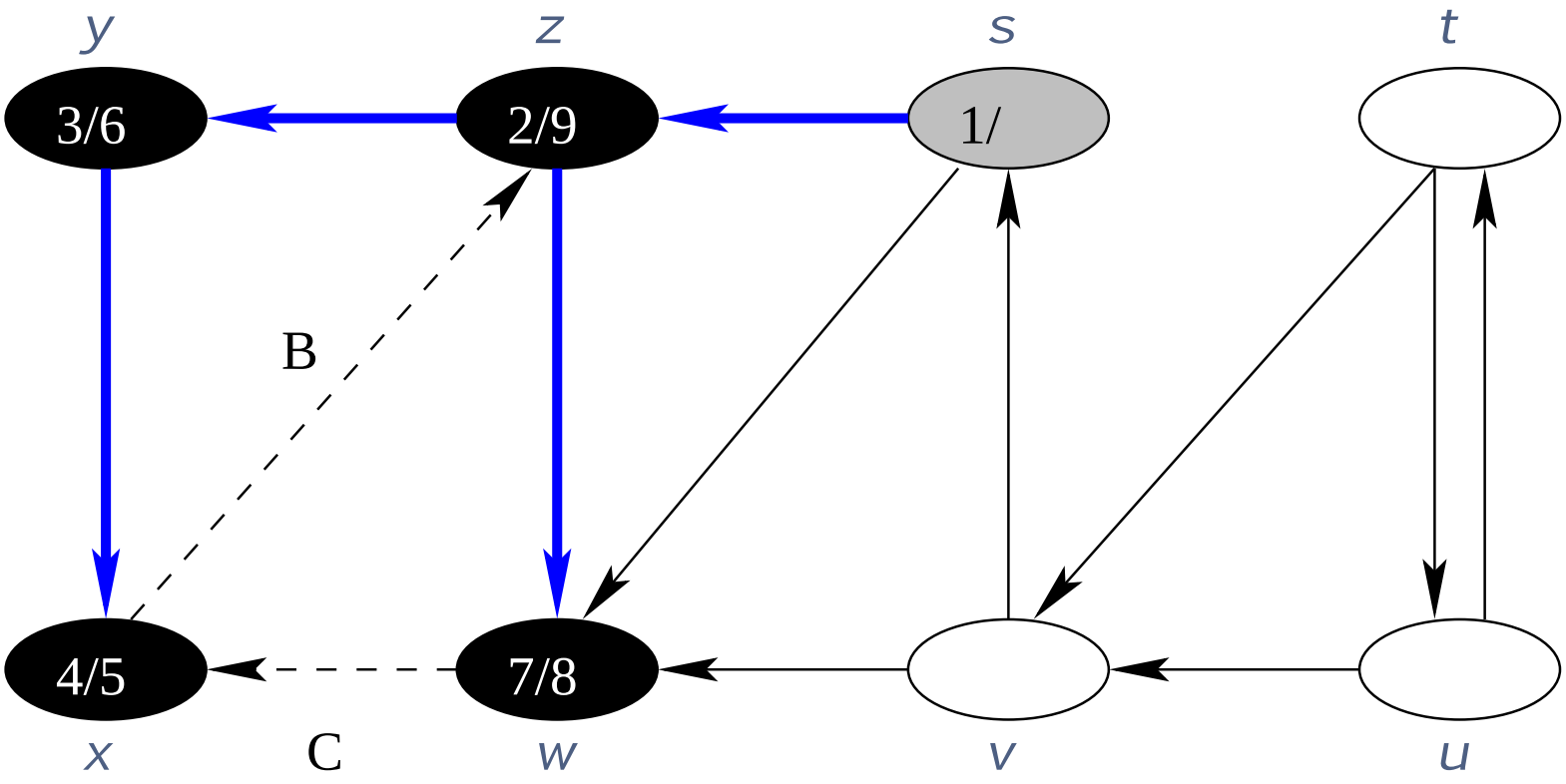
Exemplo de busca em profundidade



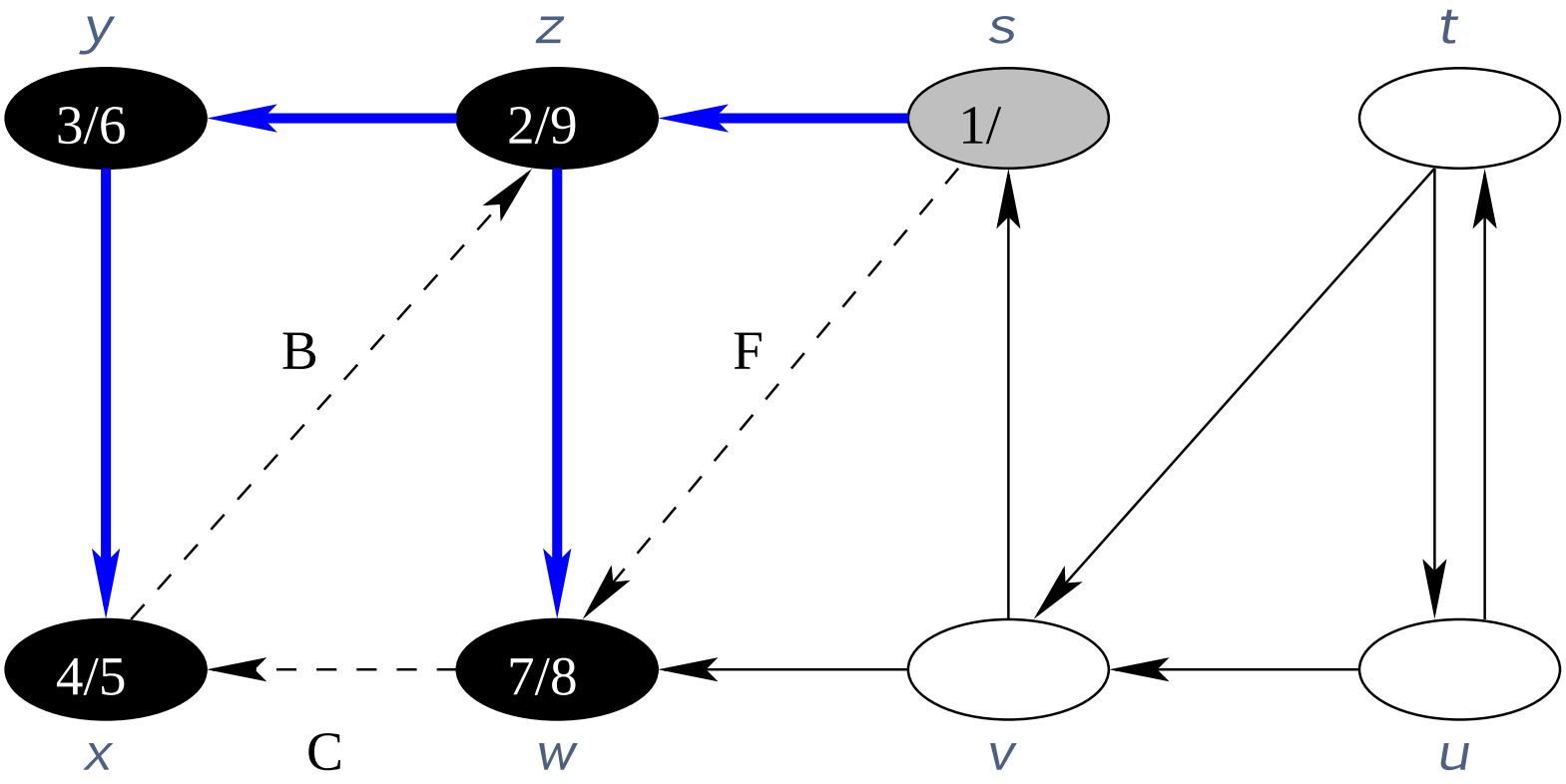
Exemplo de busca em profundidade



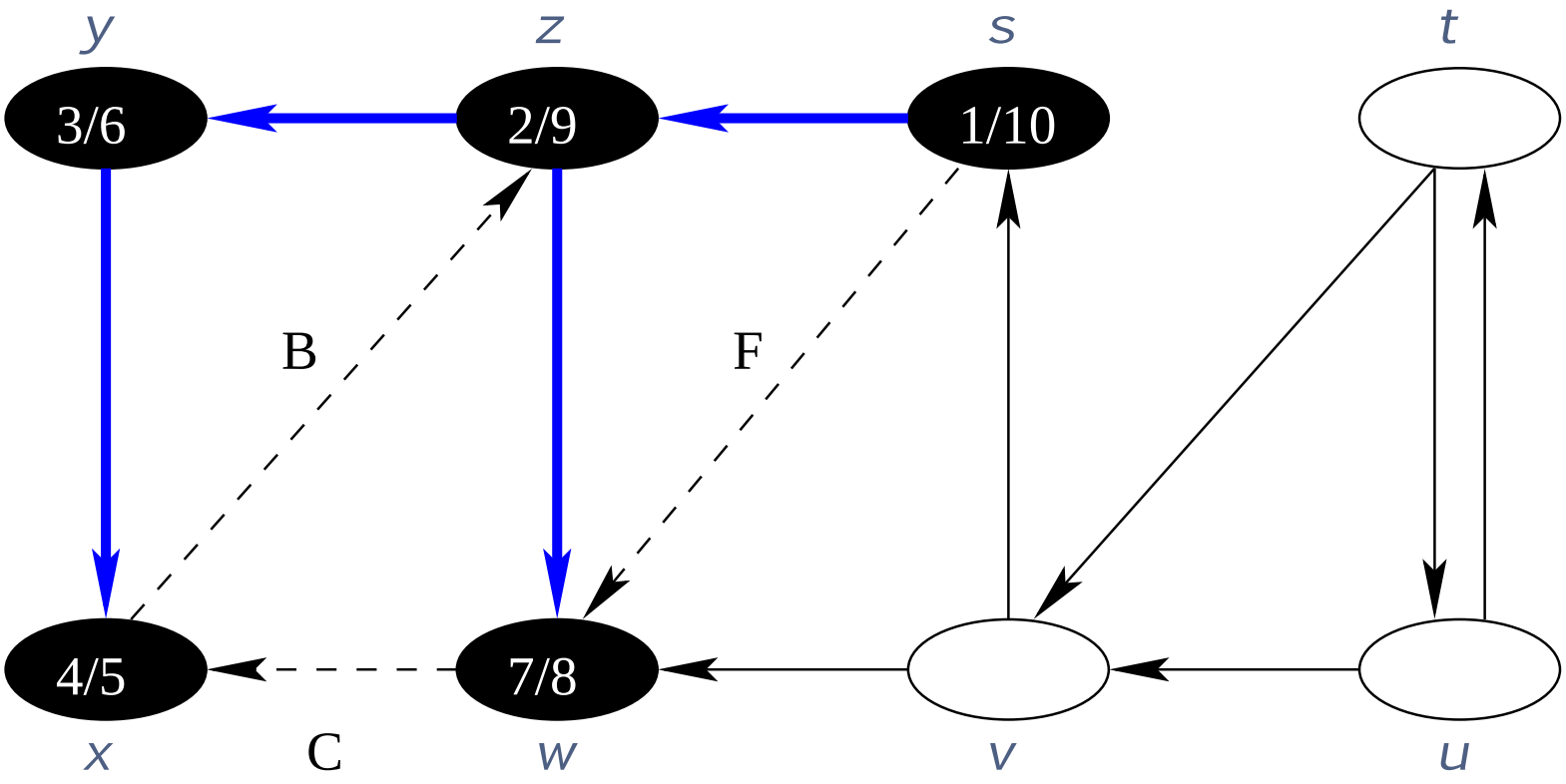
Exemplo de busca em profundidade



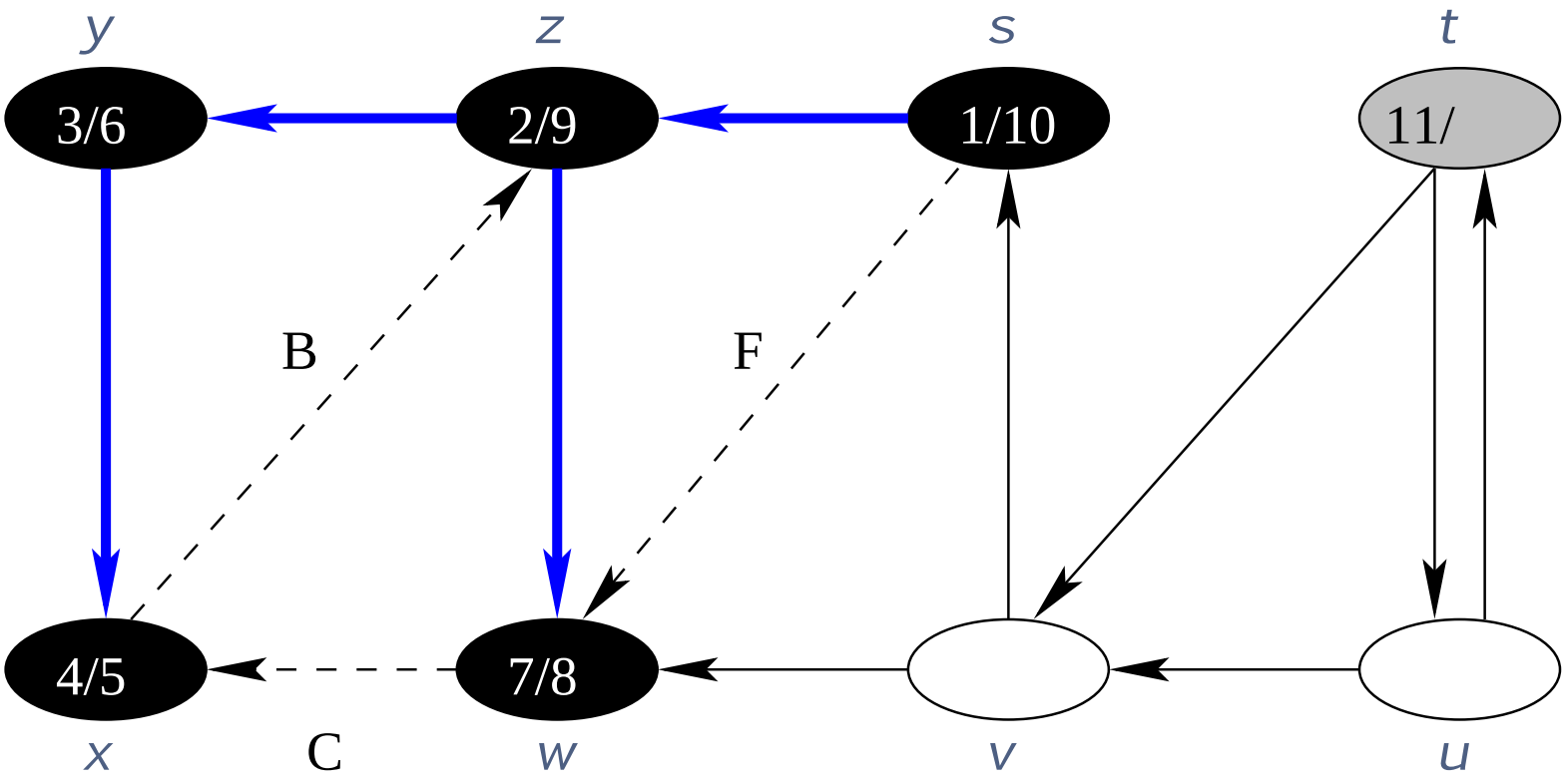
Exemplo de busca em profundidade



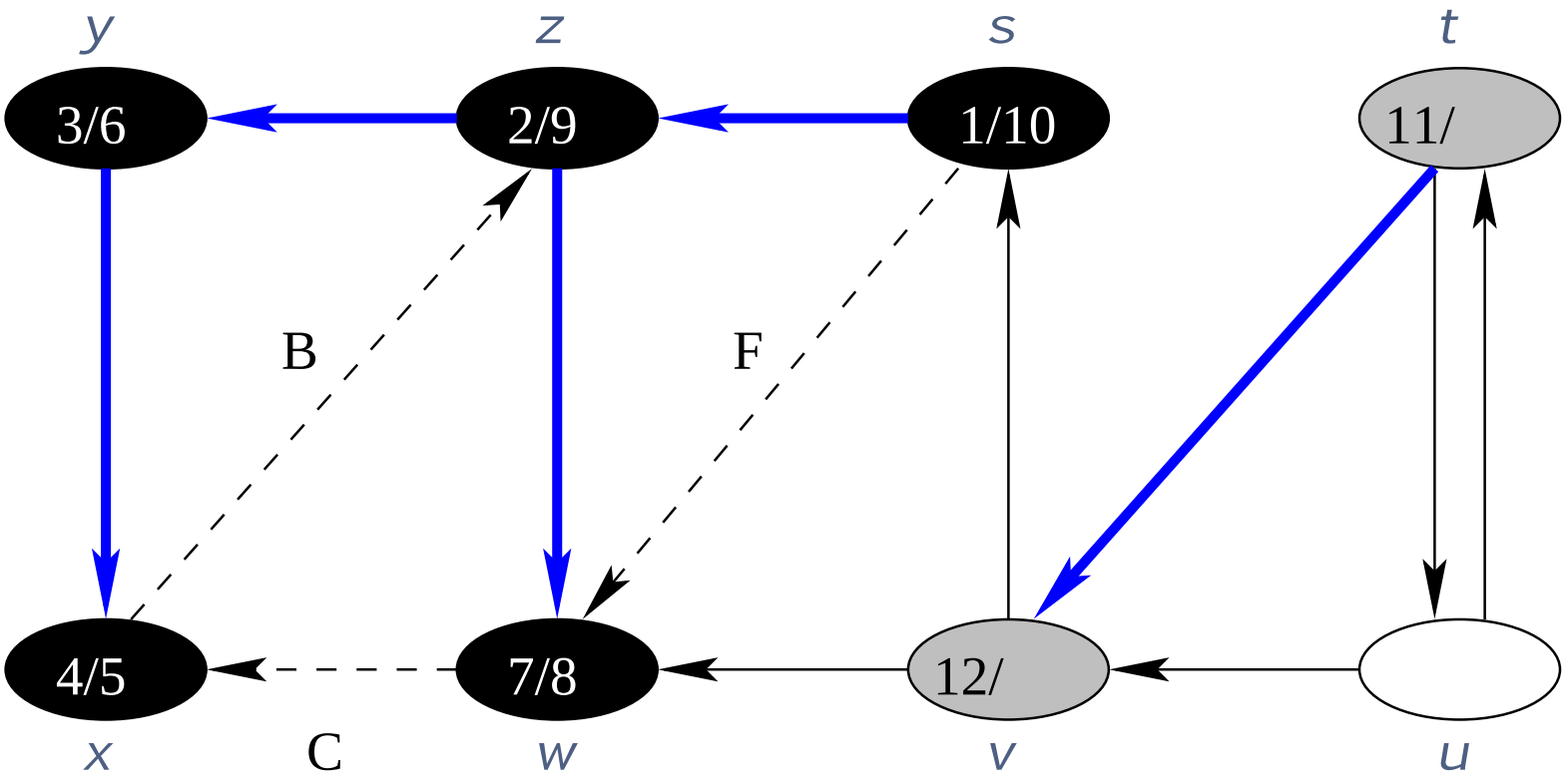
Exemplo de busca em profundidade



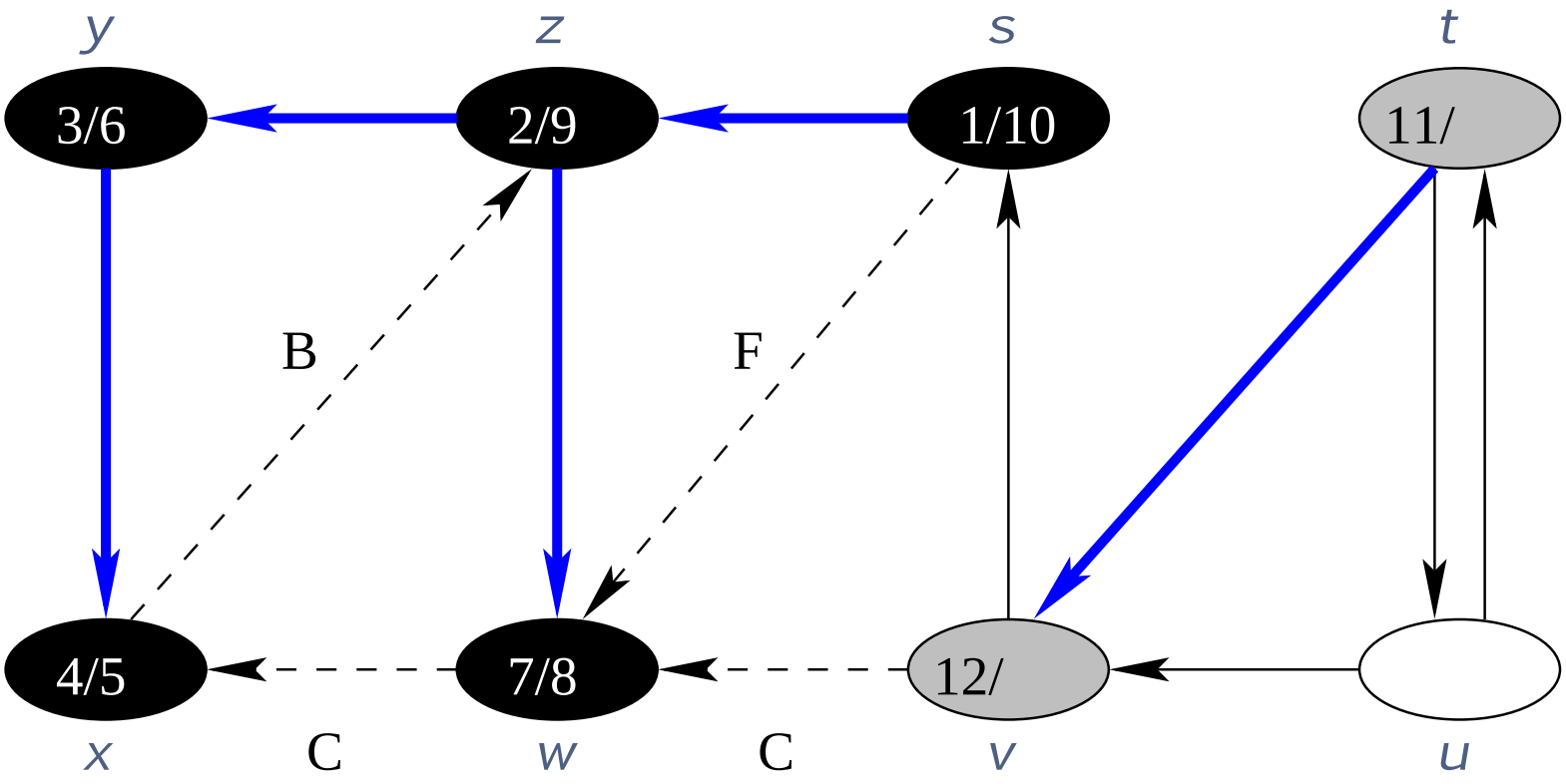
Exemplo de busca em profundidade



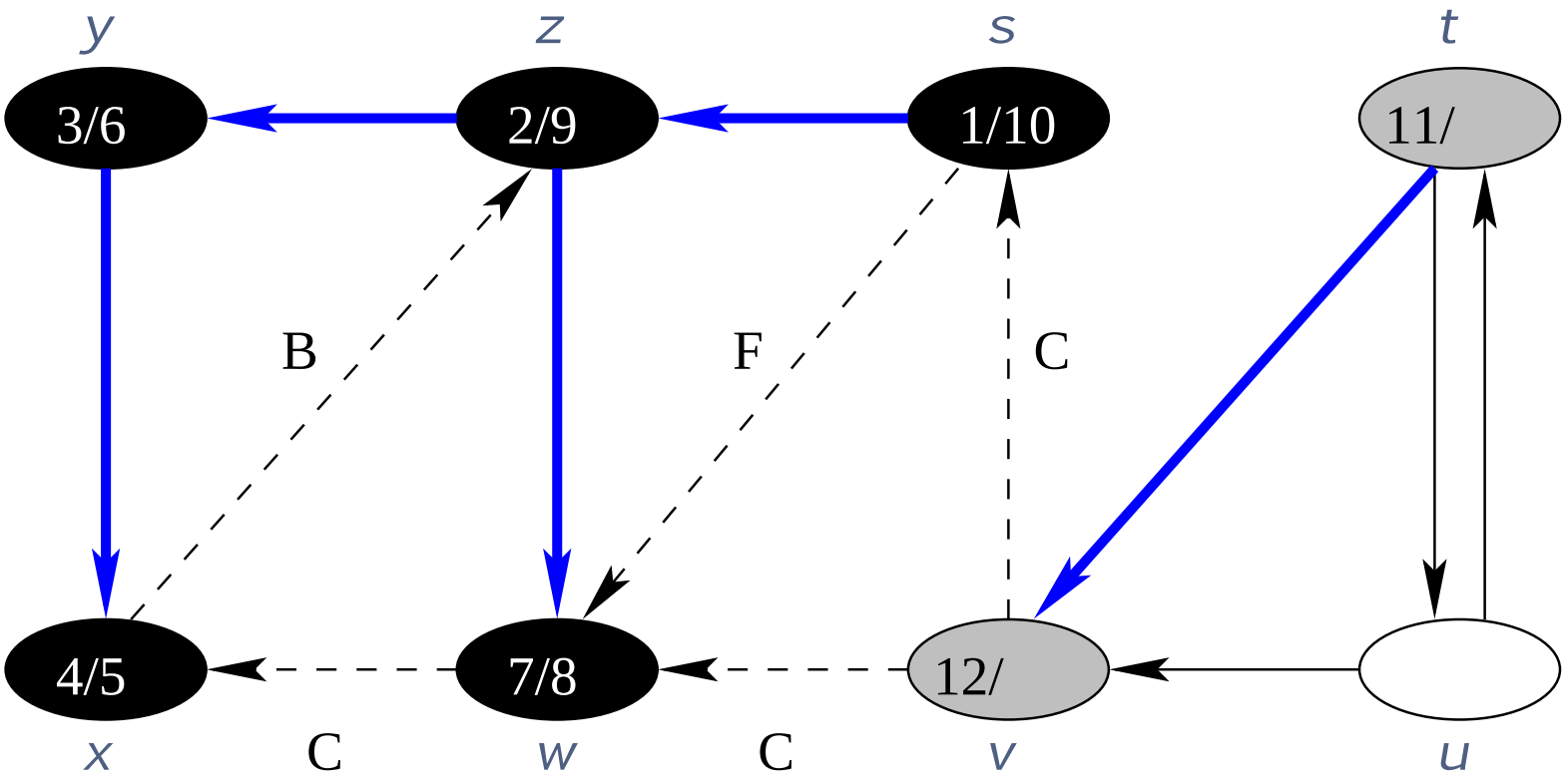
Exemplo de busca em profundidade



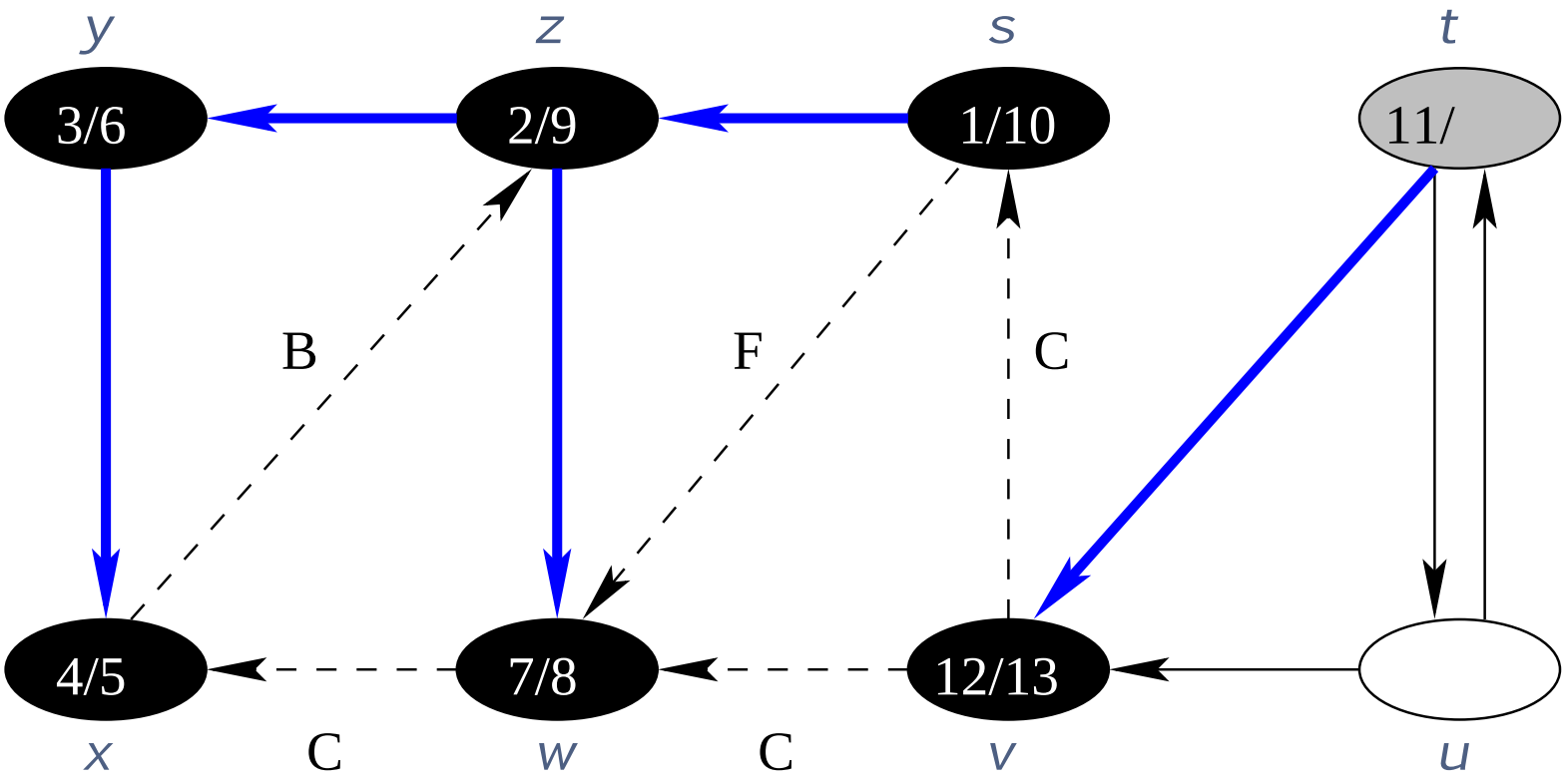
Exemplo de busca em profundidade



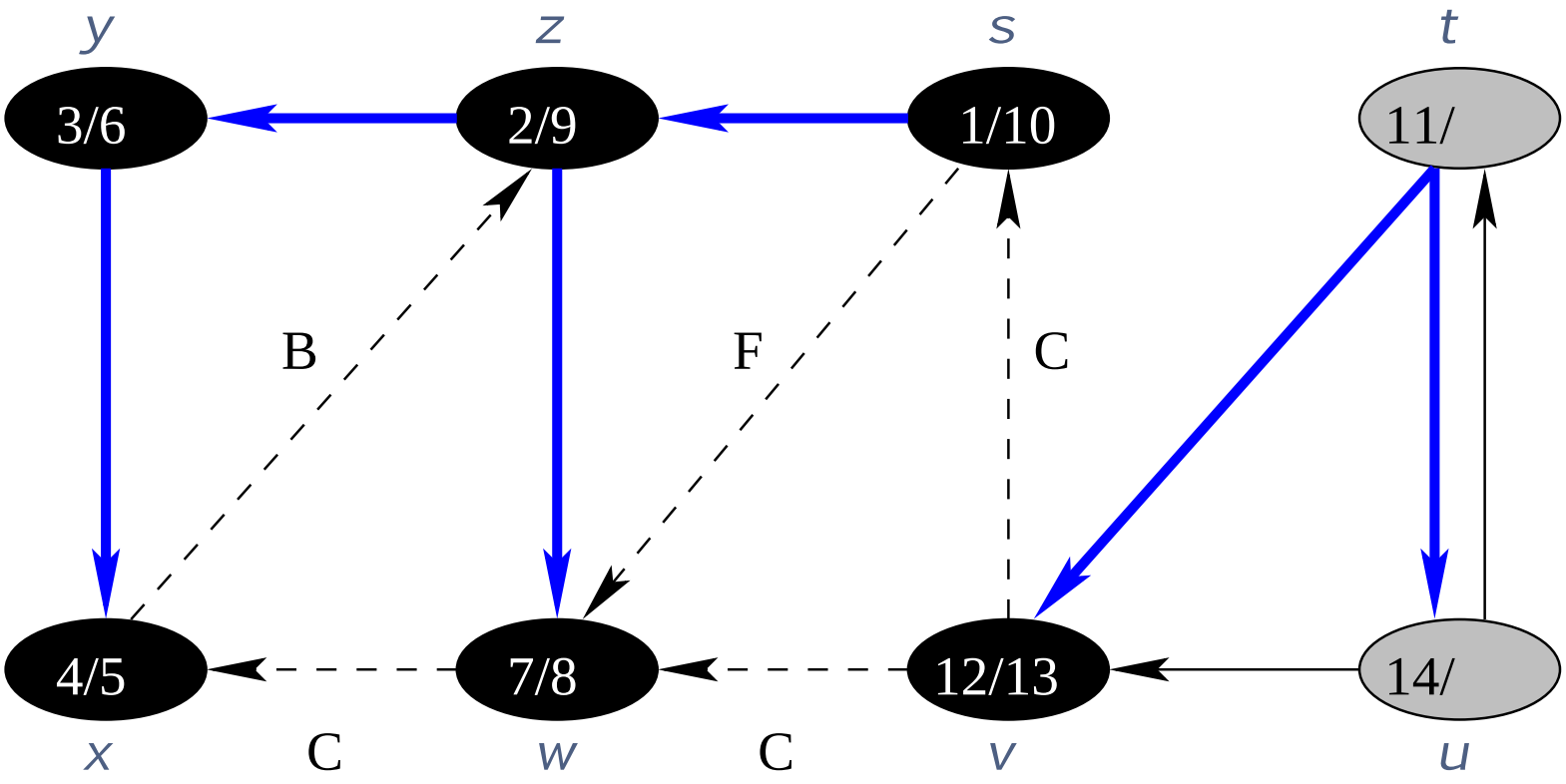
Exemplo de busca em profundidade



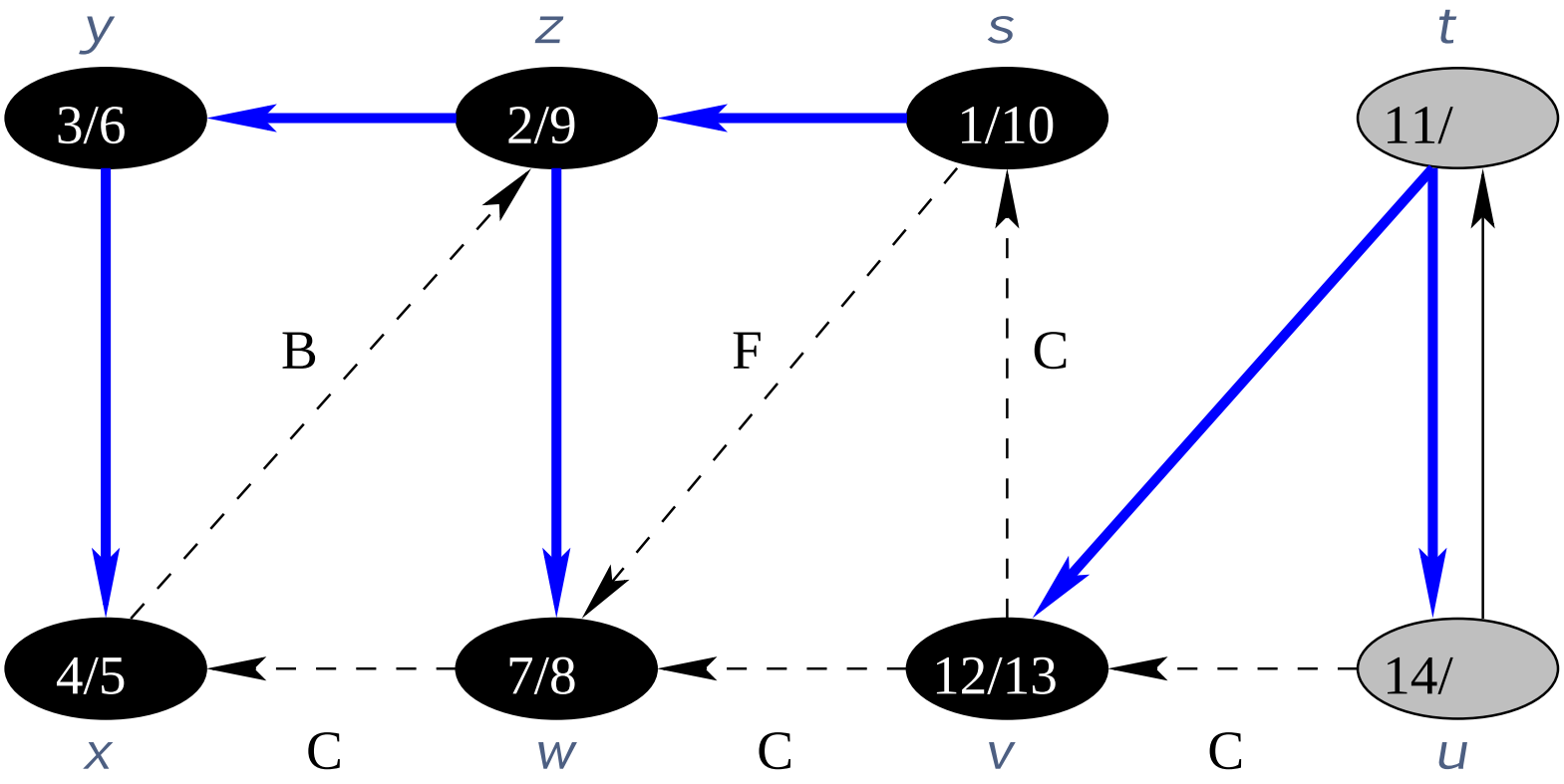
Exemplo de busca em profundidade



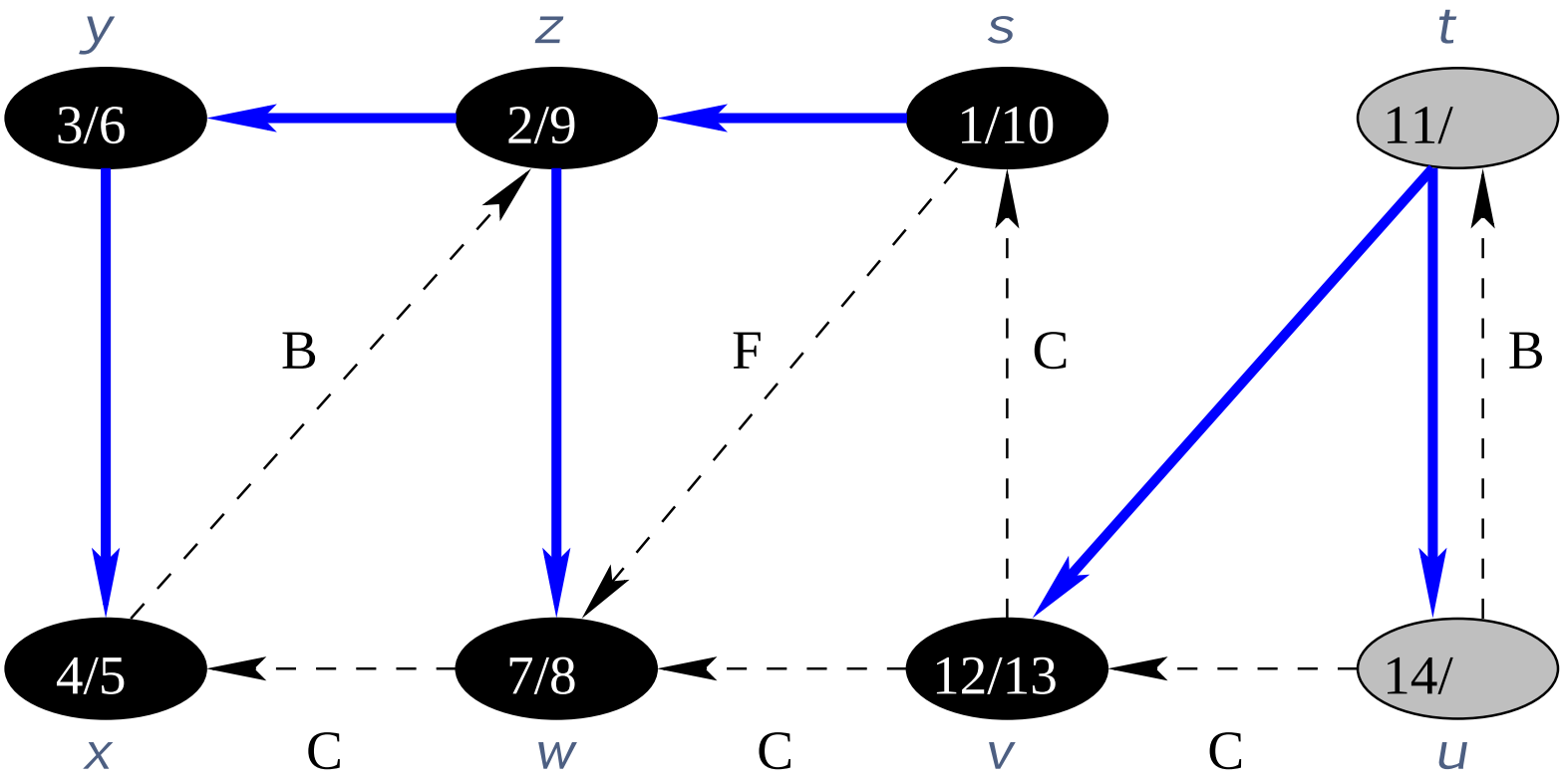
Exemplo de busca em profundidade



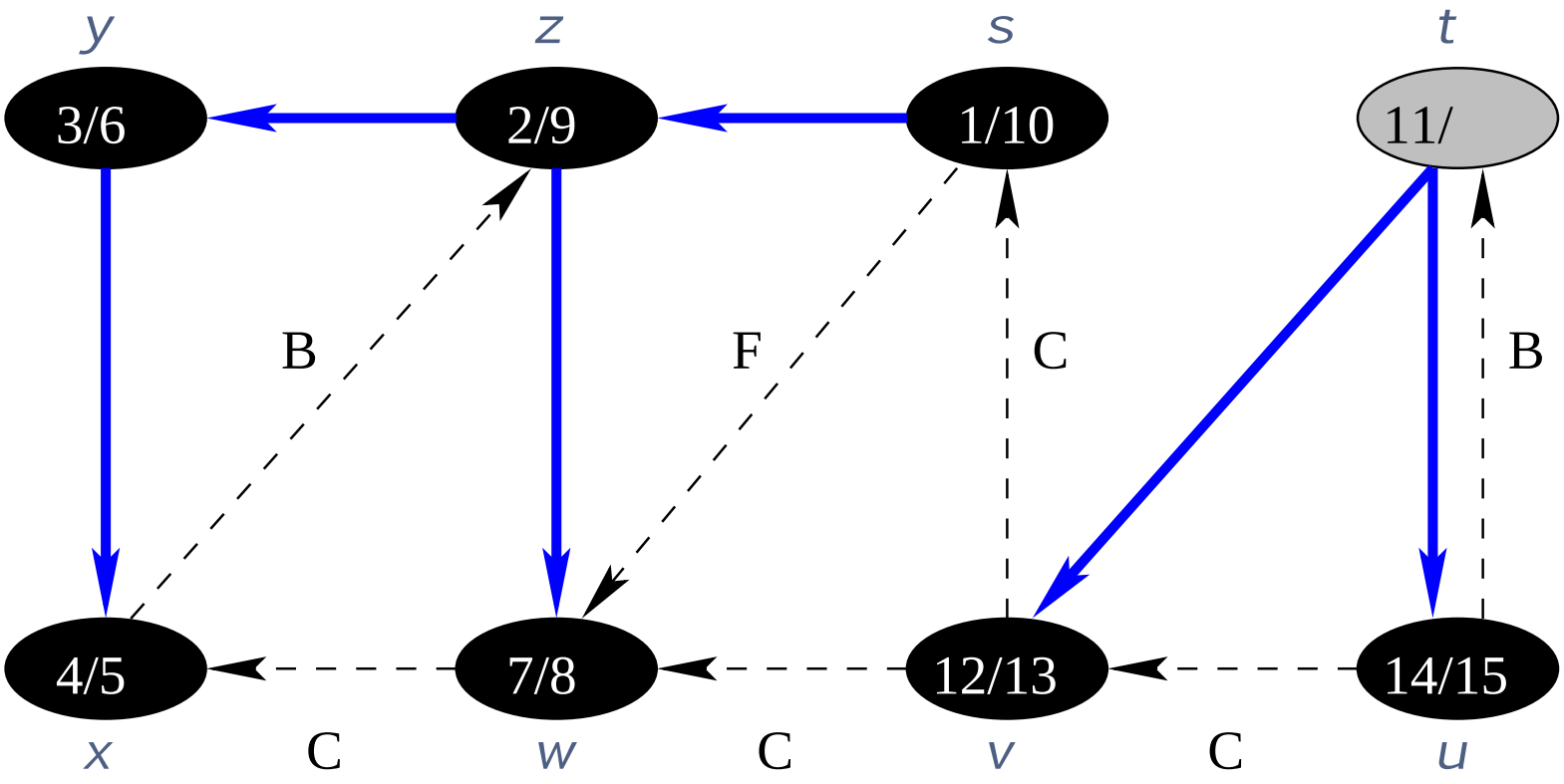
Exemplo de busca em profundidade



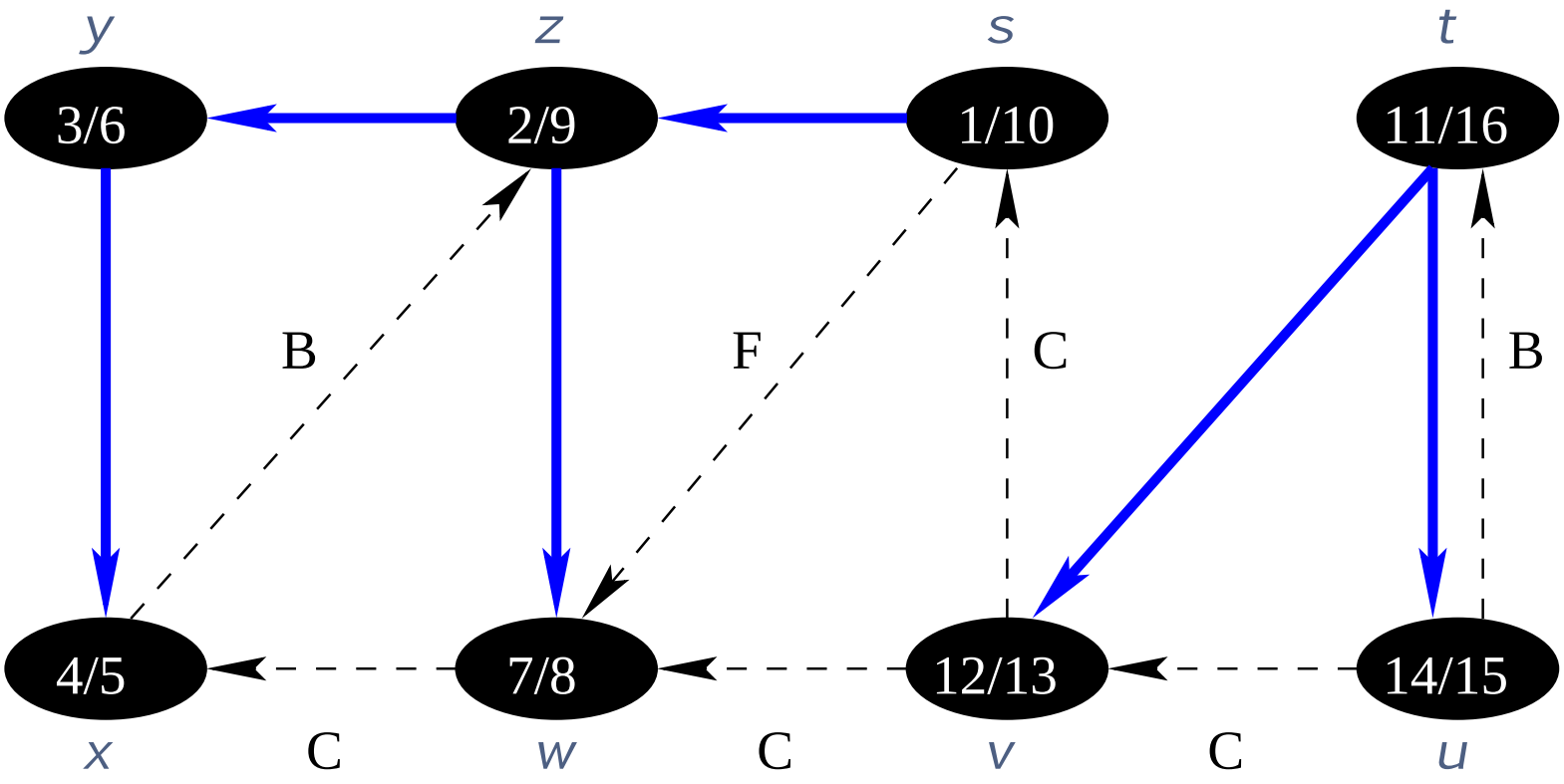
Exemplo de busca em profundidade



Exemplo de busca em profundidade

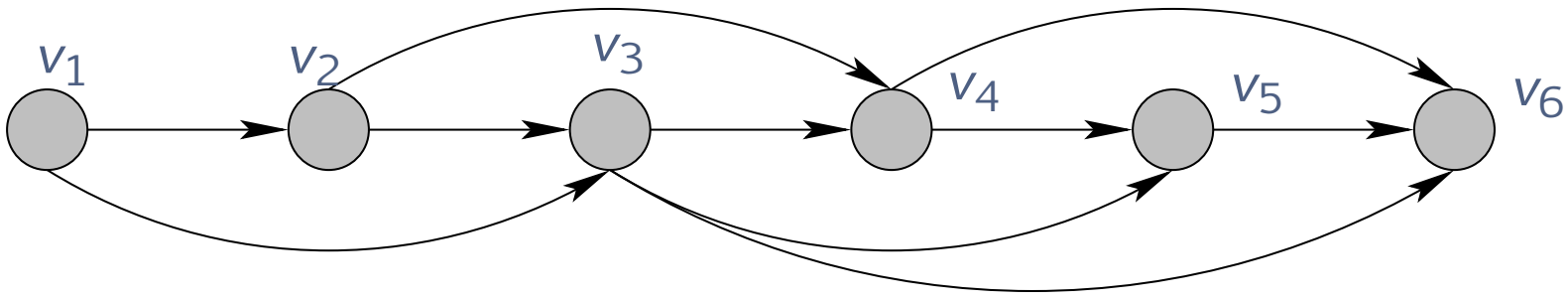


Exemplo de busca em profundidade



Ordenação Topológica

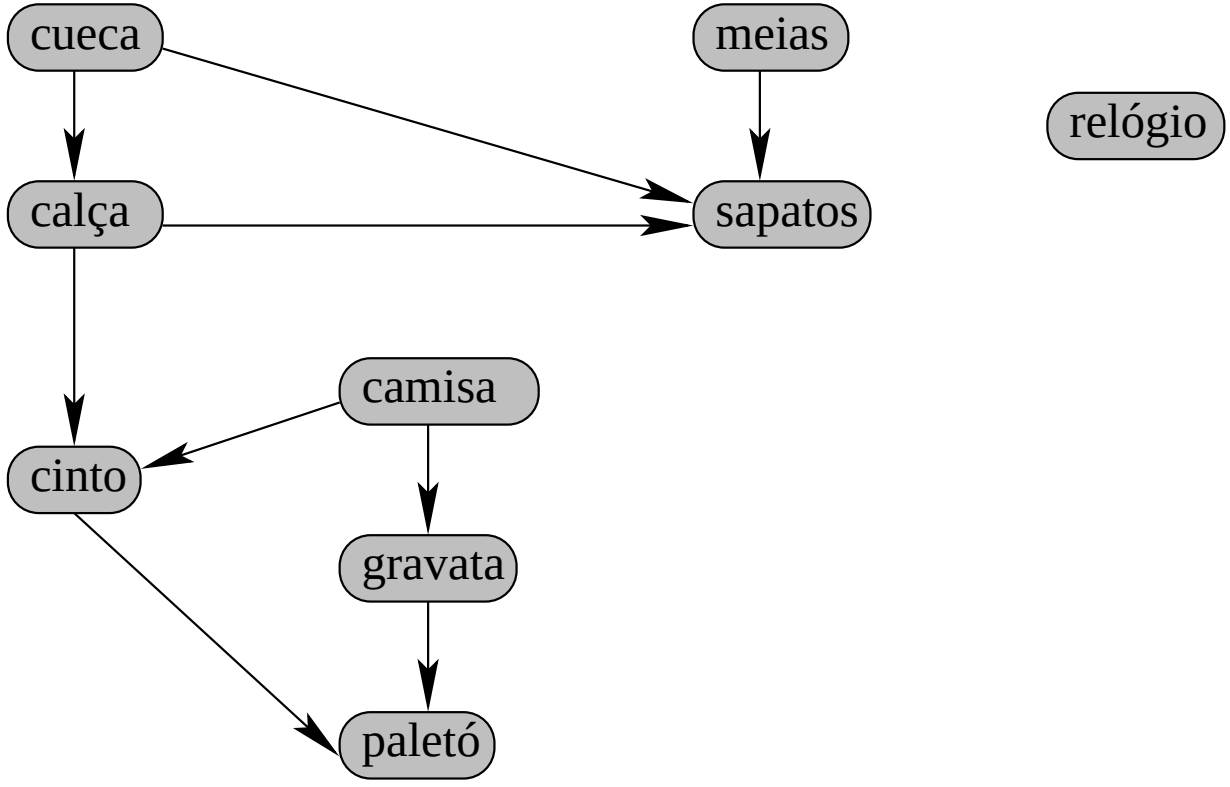
Uma ordenação topológica de um grafo direcionado é arranjo dos vértices $v_1, v_2, \ldots, v_n$ tal que se (v_i, v_j) é uma aresta do grafo, então $i < j$.



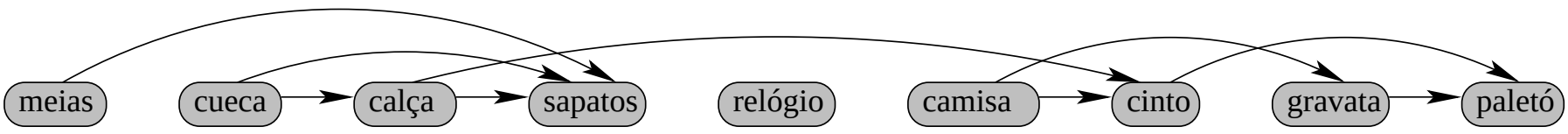
Exemplo de aplicação

Representando dependências

- um grafo pode representar precedências entre tarefas
- queremos um ordem que respeita as precedências



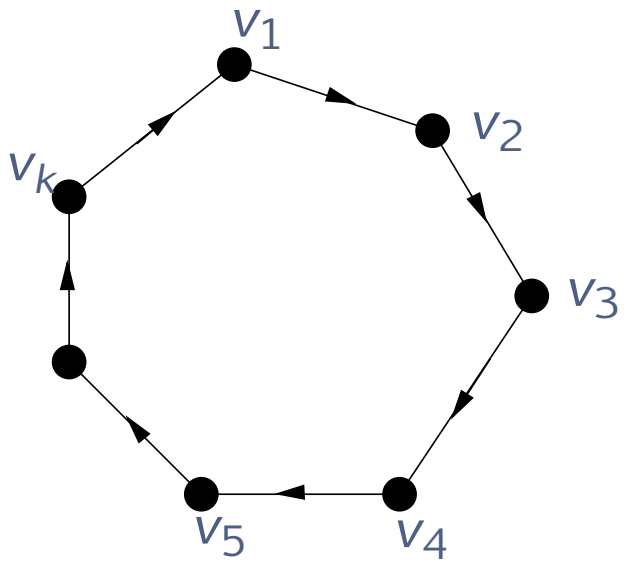
Exemplo de ordenação topológica



Codições de existência

Todo grafo direcionado possui ordenação topológica?

- não, um ciclo direcionado não possui
- nem outro grafo que contém um ciclo



Um grafo direcionado é acíclico se não contiver um ciclo direcionado.

Codições de existência

Teorema

Um grafo direcionado é acíclico se e somente se possui uma ordenação topológica.

Demonstração

- se G tem uma ordenação topológica, então ele é acíclico
- em seguida, vamos mostrar a recíproca

Um lema auxiliar

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

Lema

Todo grafo direcionado acíclico G com pelo menos um vértice possui uma fonte e um sorvedouro.

Demonstração

- tome um caminho mais longo no grafo P que vai de s até t
- observe que s é uma fonte e t é um sorvedouro

Prova do teorema

Agora podemos terminar a demonstração

- considere um grafo acíclico $G = (V, E)$
- afirmamos que G possui uma ordenação topológica
- vamos mostrar por indução em $|V|$
- se $|V| = 1$, então a afirmação é clara

Considere um grafo com pelo menos dois vértices

- pelo lema anterior, G possui um sorvedouro v_n
- pela hipótese de indução, o grafo $G - v_n$ possui uma ordenação topológica $v_1, \ldots, v_{n-1}$
- logo $v_1, v_2, \ldots, v_n$ é uma ordenação topológica de G

Encontrando uma ordenação topológica

A demonstração anterior é construtiva

- é baseada em exibir uma ordenação topológica
- ela sugere um algoritmo recursivo

Algoritmo para ordenação topológica

1. encontre um sorvedouro v_n de G
2. recursivamente, obtenha ordenação $v_1, \ldots, v_{n-1}$ de $G - v_n$
3. devolva $v_1, v_2, \ldots, v_n$

A complexidade desse algoritmo é $O(V^2)$

- encontrar um sorvedouro leva tempo $O(V)$
- há $|V|$ chamadas recursivas
- pode-se fazer em tempo $O(V + E)$ (exercício)

Algoritmo baseado em DFS

Considere um grafo direcionado acíclico

- como não há ciclo, não existe aresta de retorno
- considere o instante em que v fica preto
- nesse instante todos seus vizinhos são pretos
- isso sugere considerar os vértices na ordem de término

Ideia para o algoritmo

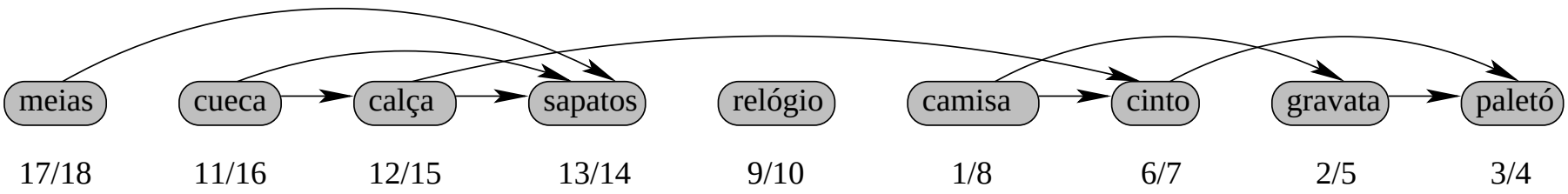
- o primeiro vértice a ficar preto não tem arestas saindo
- o segundo só pode ter arestas para o primeiro
- o terceiro só pode ter arestas para os dois primeiros
- etc.

Algoritmo Topological-Sort

Topological-Sort(G)
1 execute DFS(G) e calcule $f[v]$ para cada vértice v
2 quando um vértice finalizar, insira-o no início de uma lista
3 devolva a lista resultante

- inserir cada um dos $|V|$ vértices leva tempo $O(1)$
- além disso, executamos DFS uma vez
- portanto, a complexidade de tempo é $O(V + E)$

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

Correção

Teorema

TOPOLOGICAL-SORT(G) devolve ordenação topológica de G.

Demonstração

- a lista devolvida está em ordem decrescente de $f[v]$
- considere uma aresta arbitrária (u, v)
- basta mostrar que $f[u] > f[v]$
- considere o instante em que (u, v) foi examinada
- como (u, v) não é aresta de retorno, v não pode ser cinza
  1. se v for branco, ele será descendente de u e $f[u] > f[v]$
  2. se v for preto, então ele já foi finalizado e $f[u] > f[v]$
- em qualquer caso, obtemos o que desejávamos

Contagem de caminhos

Contagem de caminhos

Vamos resolver o seguinte problema

- **Entrada:** um grafo direcionado acíclico G e vértices s, t
- **Saída:** o número de caminhos de s a t

Observações

- queremos apenas contar os caminhos, não exibi-los
- vamos supor que o grafo não possui arestas múltiplas
- esse caso pode ser tratado de modo similar

Usando programação dinâmica

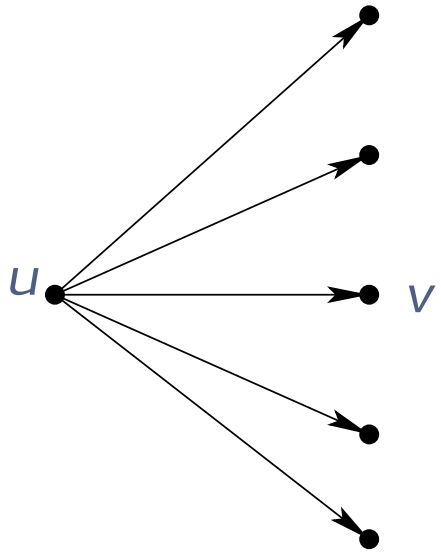
Considere o seguinte subproblema

- considere um vértice v qualquer de G
- denote por $p(v)$ o número de caminhos de v a t

Como calcular $p(s)$?

- é a soma do número de caminhos de cada vizinho v a t
- assim, existe sobreposição de problemas
- mas precisamos de uma ordem para calculá-los
- usamos programação dinâmica e ordenação topológica

Recorrência



- temos a seguinte recorrência

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

- esta recorrência vale se G contiver ciclos?

Algoritmo Conta-Caminhos

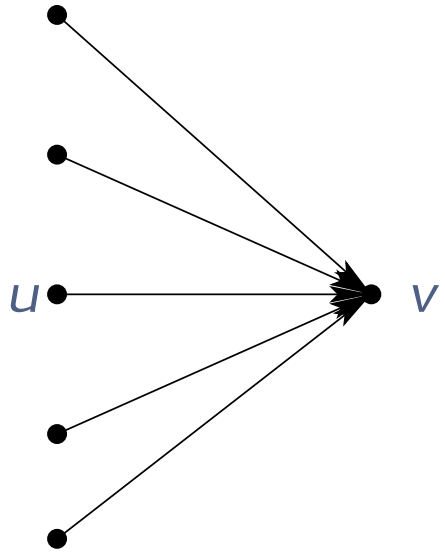
```
CONTA-CAMINHOS(G, s, t)
1  para cada u ∈ V[G] \ {s} faça
2      p[u] ← 0
3  p[t] ← 1
4  obtenha uma ordenação topológica v_1, v_2, ..., v_n de G
5  para i ← n até 1 faça
6      para cada v ∈ Adj[v_i] faça
7          p[v_i] ← p[v_i] + p[v]
8  devolva p
```

- o número de caminhos de s até t está em $p[s]$
- a complexidade de tempo é $O(V + E)$
- para demonstrar a correção, basta provar seguinte invariante de laço $p[v_i] = p(v_i)$

Uma outra maneira

Considere um subproblema alternativo

- considere um vértice v qualquer de G
- denote por $q(v)$ o número de caminhos de s a v



- temos a seguinte recorrência

$$p(v) = \sum_{u : v \in \text{Adj}[u]} p(u)$$

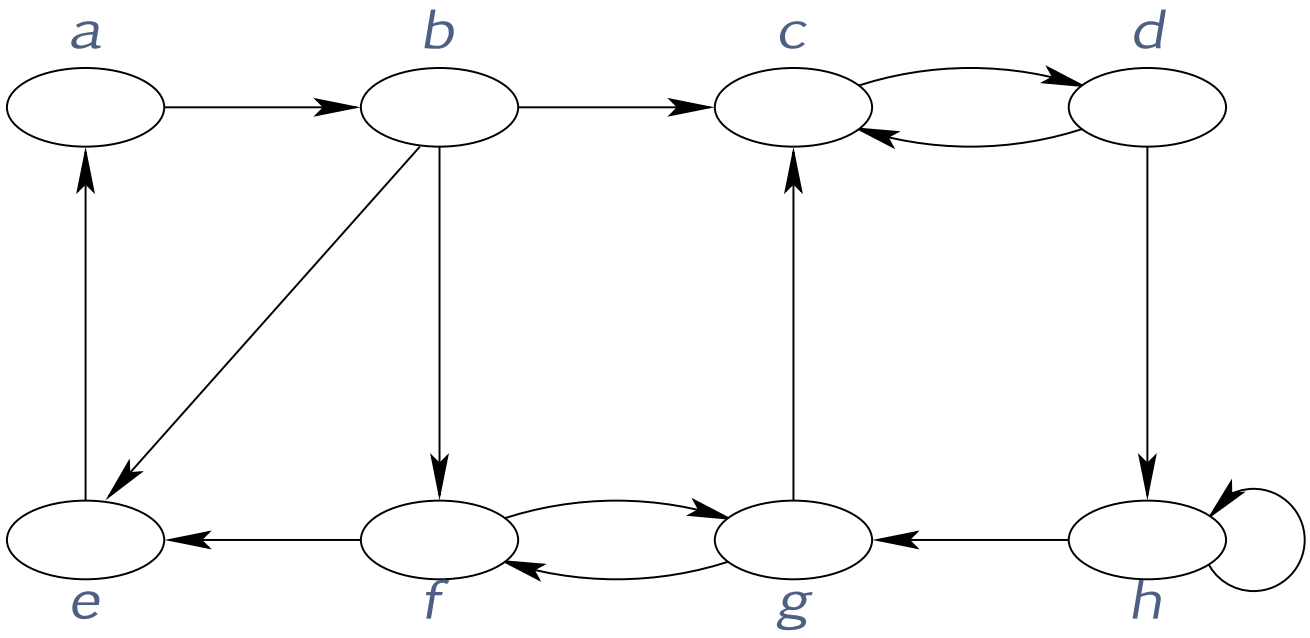
Algoritmo Conta-Caminhos alternativo

```
CONTA-CAMINHOS(G, s, t)
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 faça
6      para cada v ∈ Adj[v_i] faça
7          q[v] ← q[v] + q[v_i]
8  devolva q
```

- note a diferença no modo em que $q[v]$ é calculado

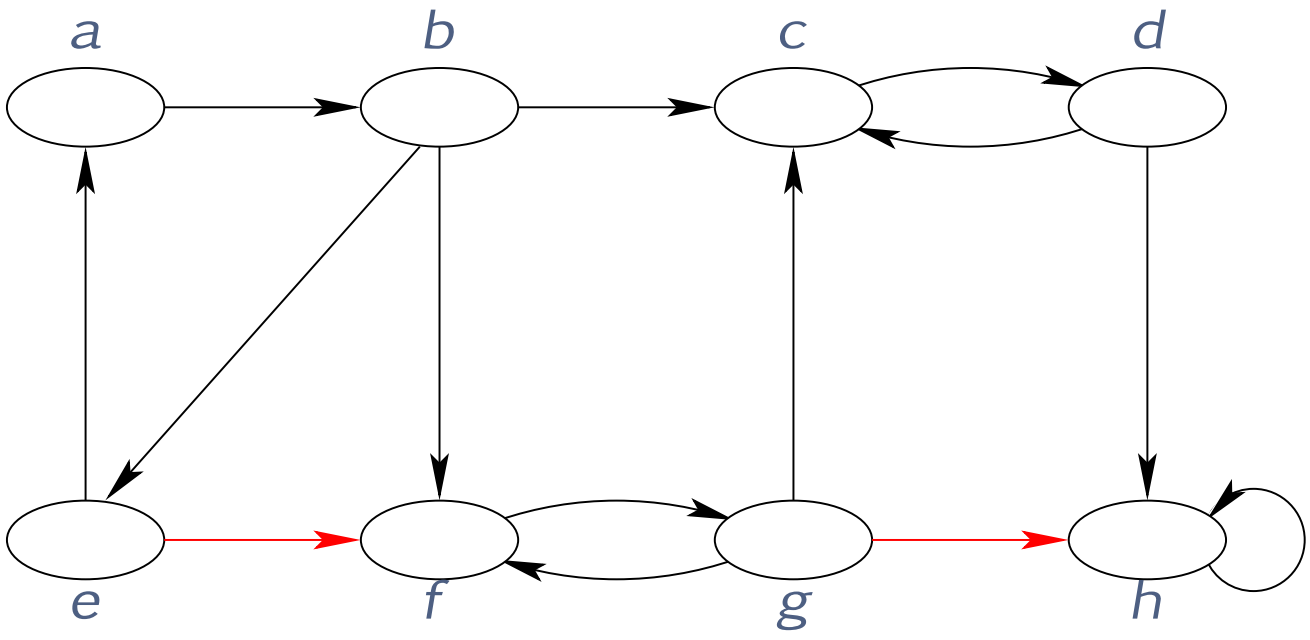
Componentes fortemente conexas

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

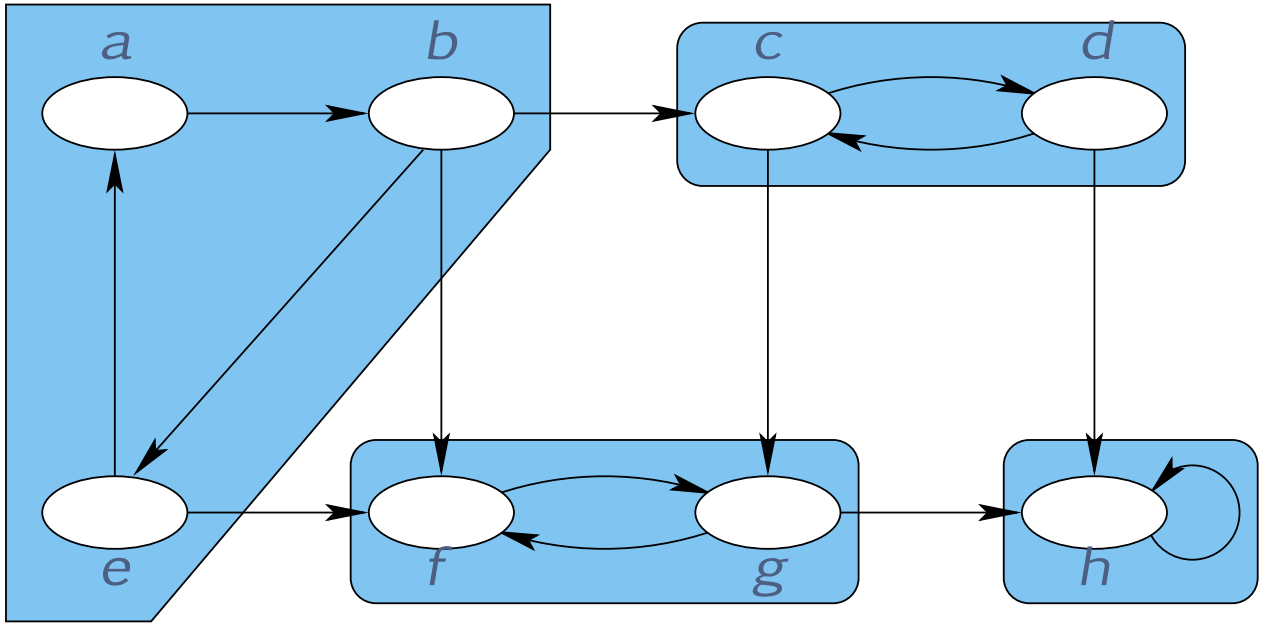
Componente fortemente conexa

Uma componente fortemente conexa 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 e

(2) C é maximal com respeito à propriedade (1).

Componente fortemente conexa

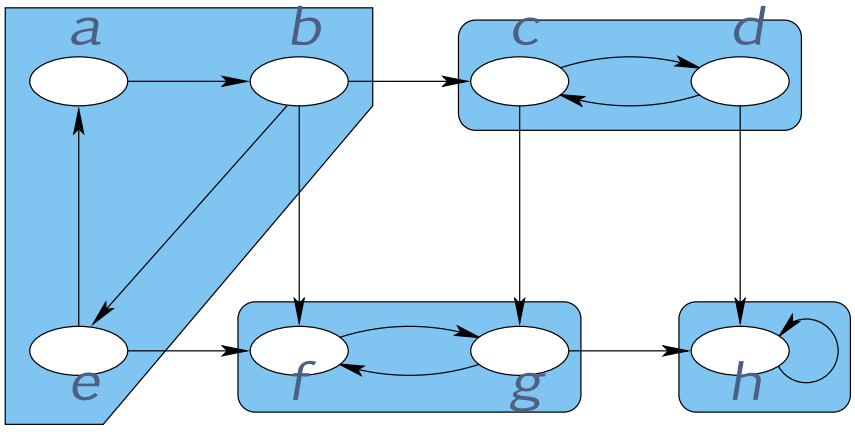


- podemos particionar um grafo direcionado em componentes fortemente conexas
- como encontrar as componentes fortemente conexas?

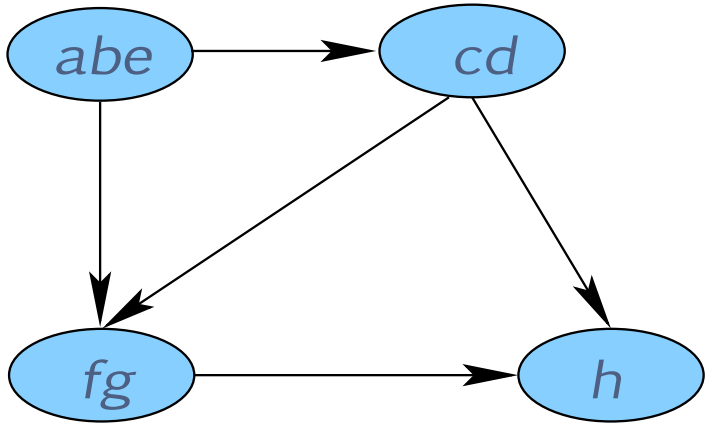
Grafo componente

O grafo componente de um grafo direcionado $G = (V, E)$ é um grafo direcionado em que

- cada vértice é uma componente fortemente conexa
- existe aresta (C, D) se houver $(u, v) \in E$ com $u \in C$ e $v \in D$



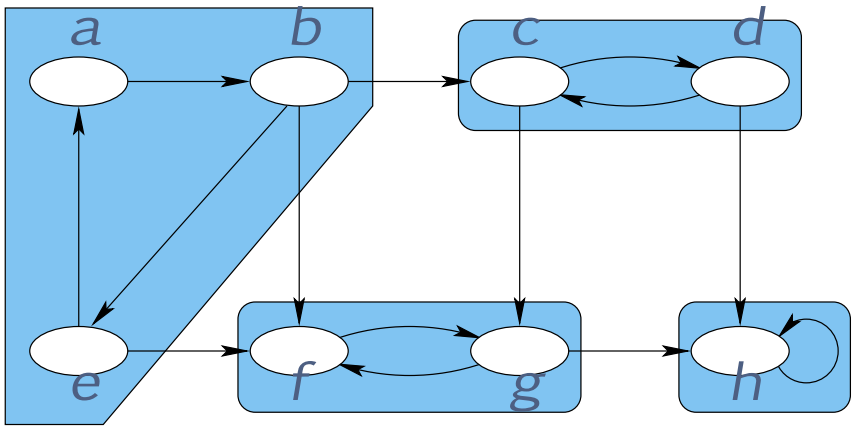
G



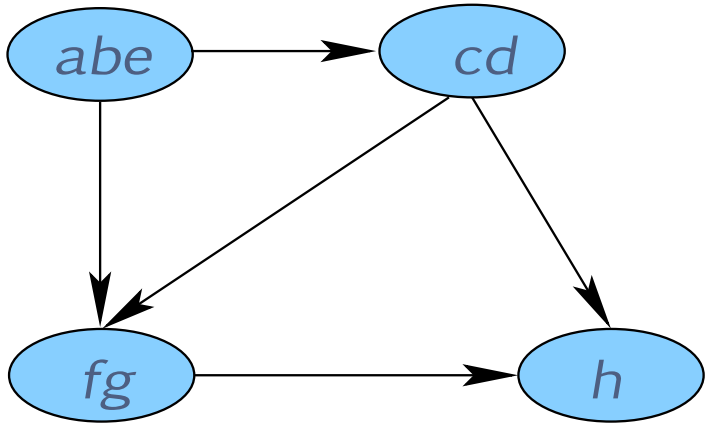
G^{CFC}

- denotamos o grafo componente por G^{CFC}
- note que G^{CFC} é acíclico (por quê?)

Grafo componente



G



G^{CFC}

Considere uma busca em profundidade sobre G

- seja u o último vértice finalizado
- então u deve pertencer a uma fonte de G^{CFC} (por quê?)
- visitados as componentes de G^{CFC} em ordem topológica

Grafo transposto

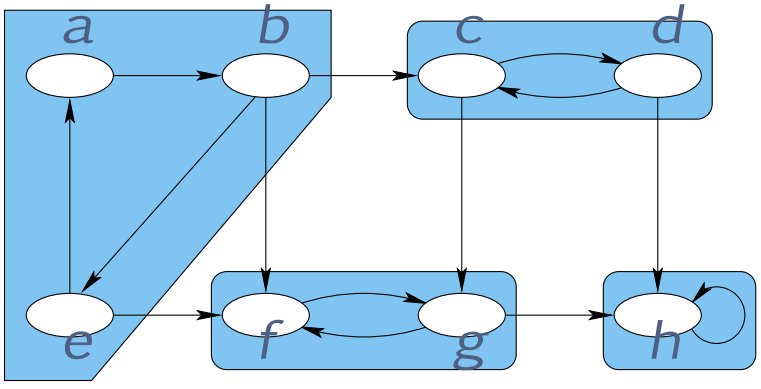
O grafo transposto de um grafo direcionado $G = (V, E)$ é um grafo direcionado que

- tem o mesmo conjunto de vértices
- tem uma aresta (u, v) se houver aresta (v, u) em G

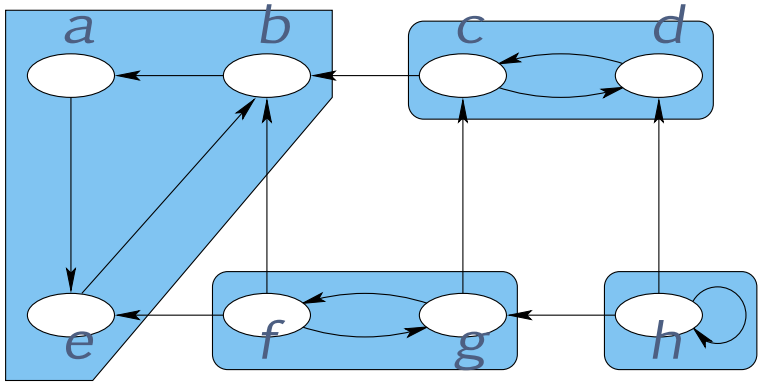
Observações

- denotamos o grafo transposto por G^T
- ele é obtido invertendo-se as arestas de G
- podemos calcular G^T em tempo $O(V + E)$

Grafo transposto



G



G^T

Como encontrar uma componente fortemente conexa?

- note que G e G^T têm as mesmas componentes
- mas componentes fontes para G são sorvedouros para G^T
- suponha que temos um vértice u de uma fonte em G^{CFC}
- os vértices alcançáveis de u formam uma componente!

Algoritmo

COMPONENTES-FORTEMENTE-CONEXAS(G)

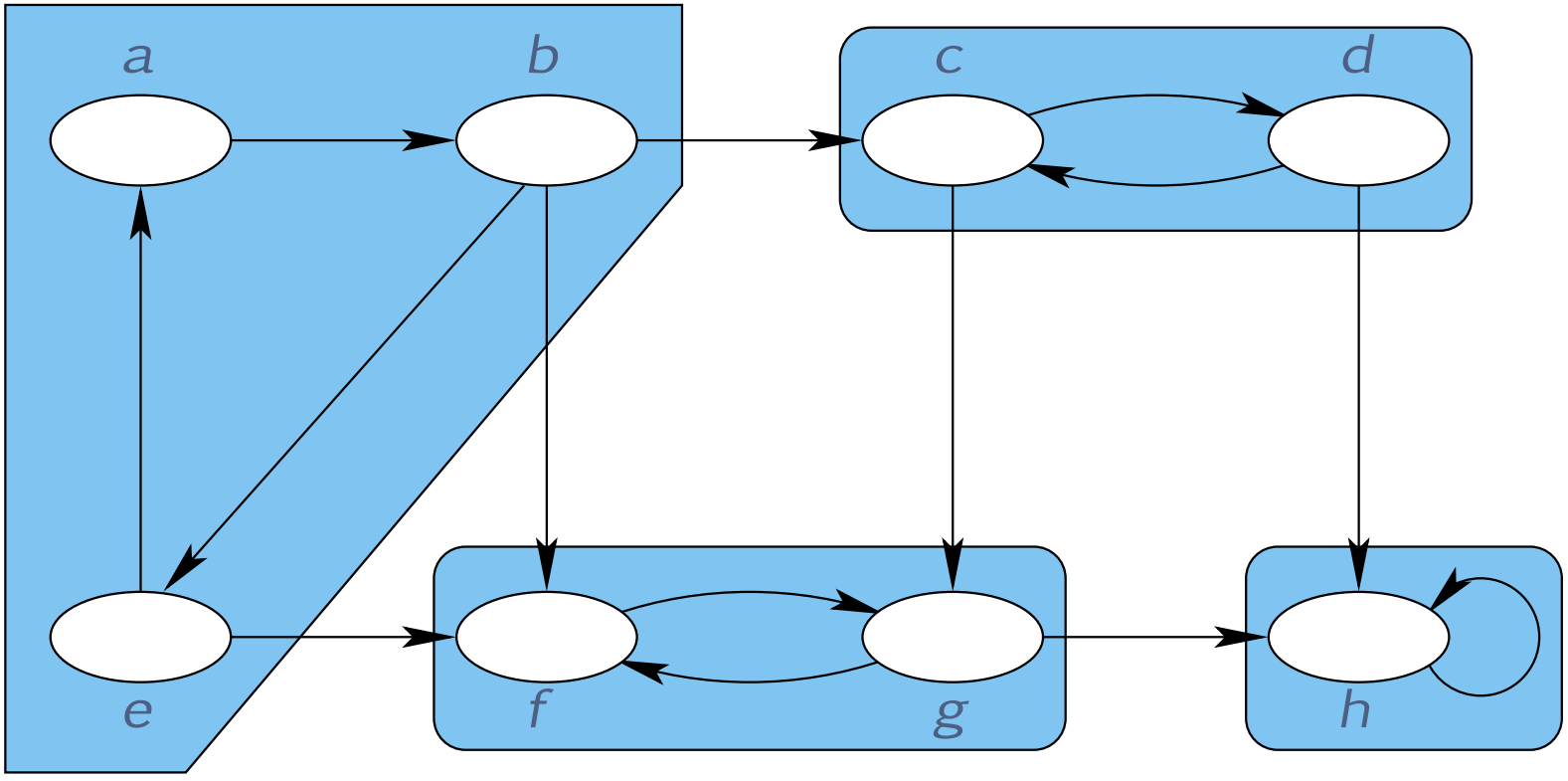
1 execute DFS(G) e calcule $f[v]$ para cada $v \in V$

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

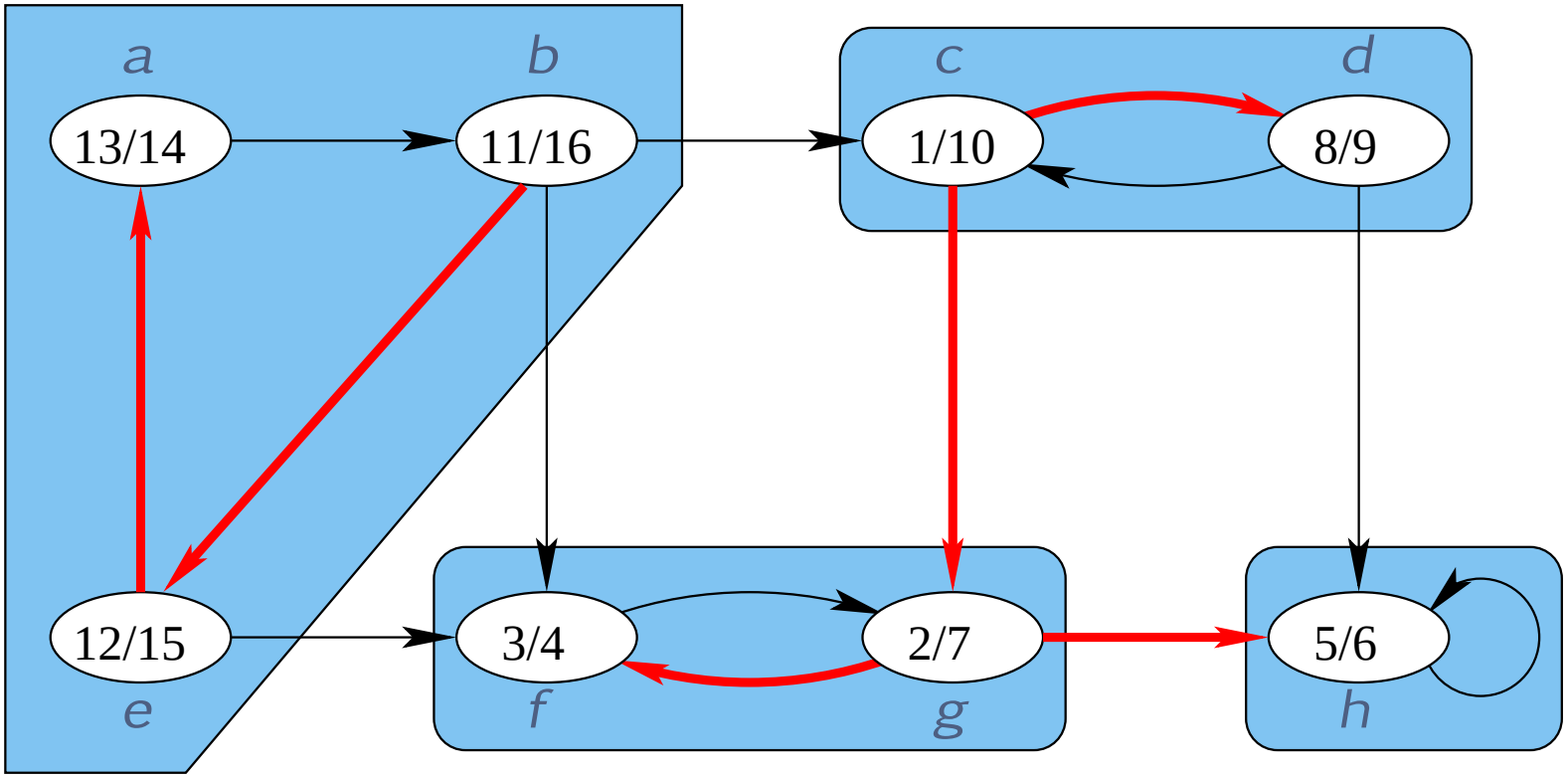
3 devolva os conjuntos de vértices de cada árvore da floresta de busca encontrada

- a complexidade de tempo é $O(V + E)$

Exemplo

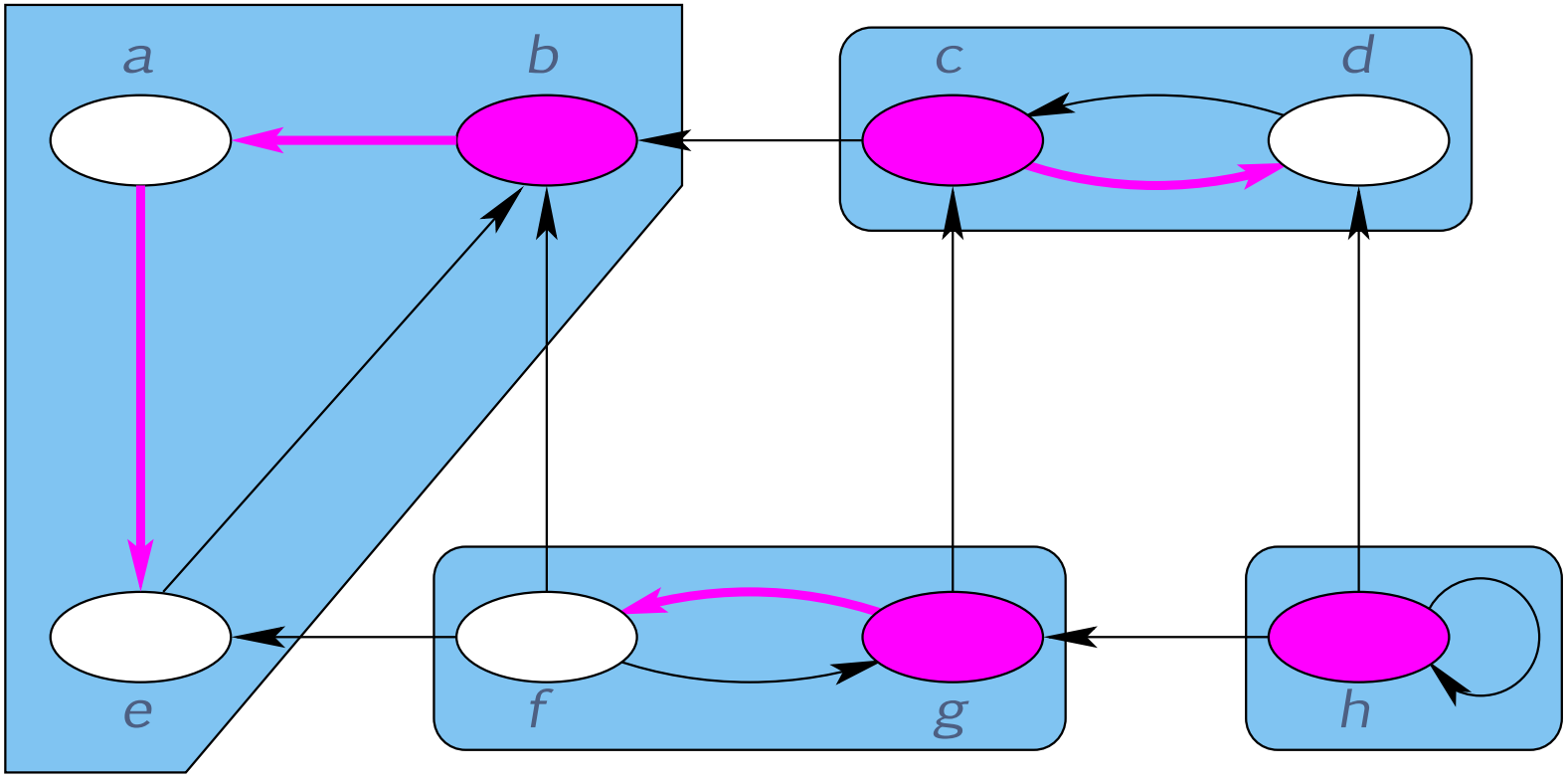


Exemplo



1 execute DFS(G) e calcule $f[v]$ para cada $v \in V$

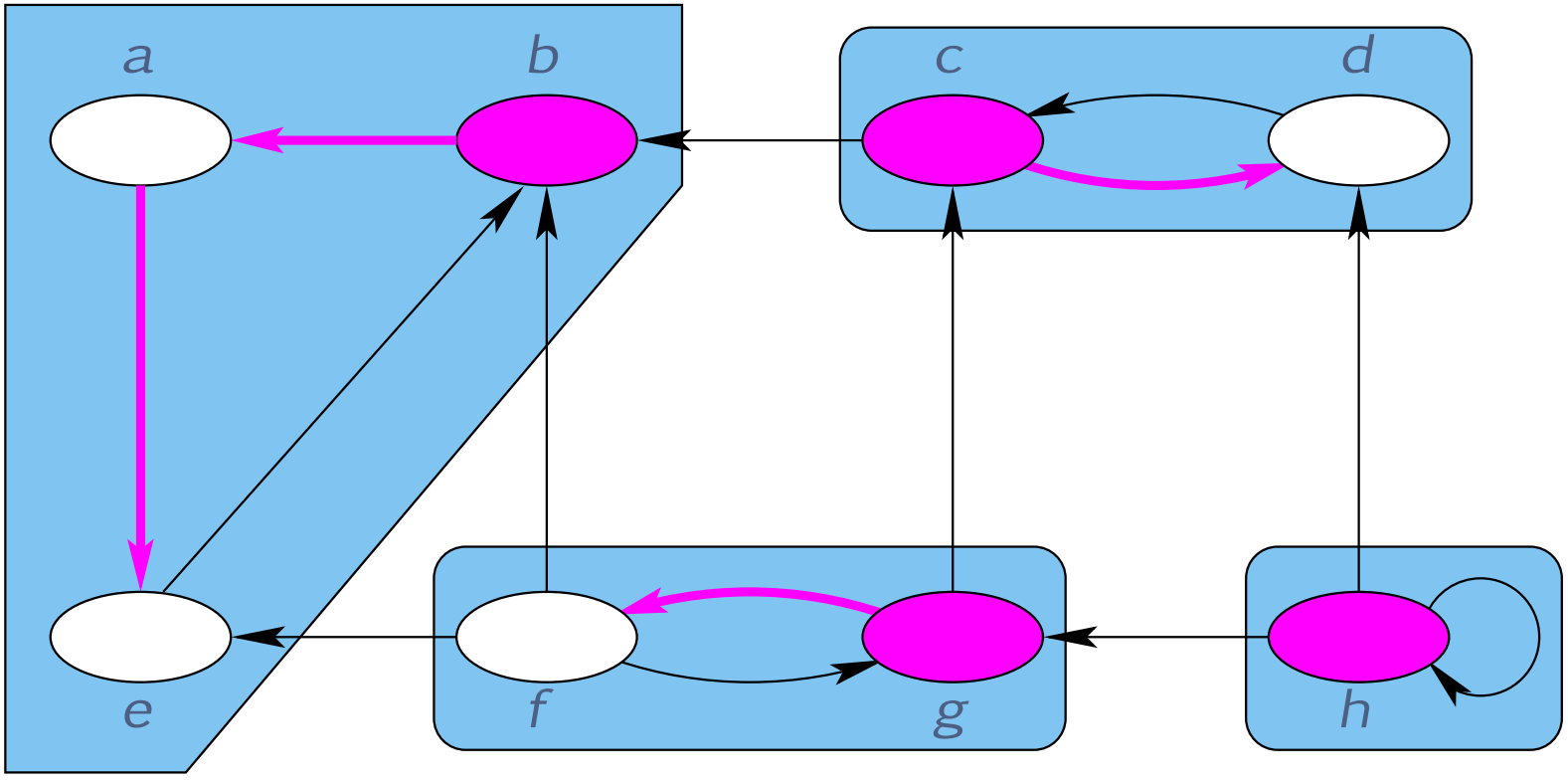
Exemplo



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 encontrada

Exemplo



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 encontrada

Correção

COMPONENTES-FORTEMENTE-CONEXAS(G)

1 execute DFS(G) e calcule $f[v]$ para cada $v \in V$

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 encontrada

Teorema

O algoritmo COMPONENTES-FORTEMENTE-CONEXAS determina as componentes fortemente conexas de G em tempo $O(V + E)$.

- antes da demonstração, precisamos de uma preparação

Lema auxiliar

Lema 1

Sejam C e D duas componentes fortemente conexas e considere vértices $u, v \in C$ e $u', v' \in D$.

- Se existe algum caminho $u \rightsquigarrow u'$,
- então não existe um caminho $v' \rightsquigarrow v$.

- segue da maximalidade de C e D
- o lema significa que G^{CFC} é acíclico

Definições auxiliares

Vamos adotar alguma convenção

- vamos considerar uma execução do algoritmo
- d e f referem-se à busca em profundidade da linha 1

Para cada subconjunto U de vértices, defina

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

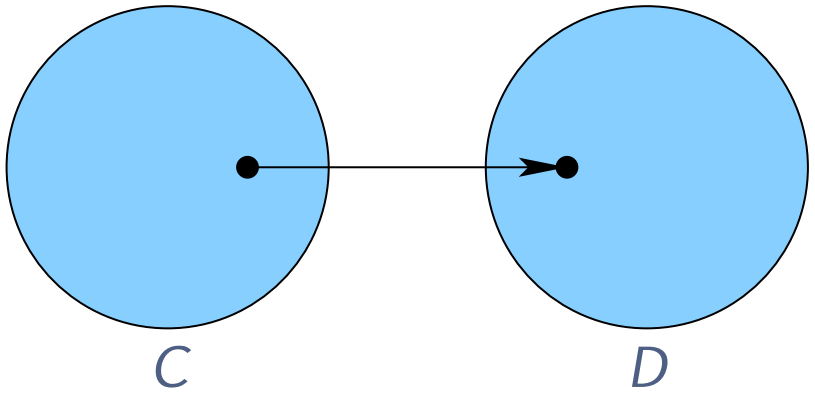
Em outras palavras

- $d(U)$ é o primeiro instante em que um vértice é descoberto
- $f(U)$ é o último instante em que um vértice é finalizado

Outro lema auxiliar

Lema 2

Sejam C e D duas componentes fortemente conexas. Se existe aresta (u, v) tal que $u \in C$ e $v \in D$, então $f(C) > f(D)$.

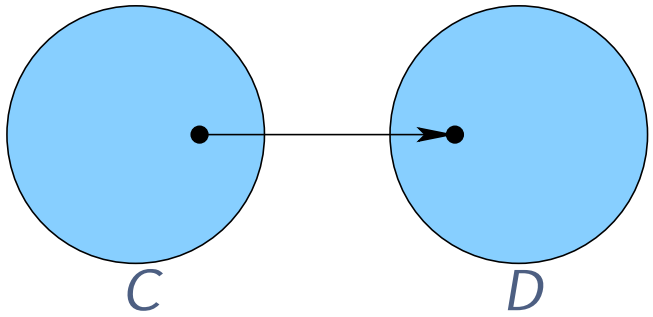


Corolário

Se G^T tem aresta (u, v) tal que $u \in C$ e $v \in D$, então $f(C) < f(D)$.

- segue do fato de que G e G^T têm as mesmas componentes

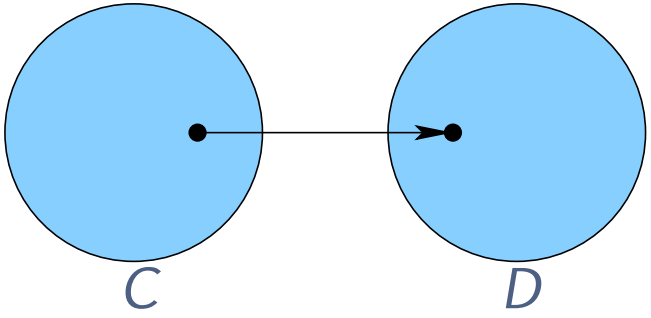
Prova do lema



Demonstração

- primeiro suponha que $d(C) < d(D)$
- isso é, suponha que descobrimos C antes de D
- seja x um vértice de C tal que $d[x] = d(C)$
- assim, x é o primeiro vértice de C a ser descoberto
- no instante $d[x]$, existia um caminho branco de x a cada um dos vértices em $C \cup D$
- então todos os vértices de $C \cup D$ são descendentes de x
- e portanto $f(D) < f[x] \leq f(C)$

Prova do lema (cont)



Continuando

- agora suponha que $d(C) > d(D)$
- assim, o primeiro vértice a ser descoberto está em D
- logo, cada um dos vértices de D é finalizado antes de qualquer vértice de C ser descoberto
- portanto $f(C) > f(D)$

Prova do teorema

Teorema

O algoritmo Componentes-Fortemente-Conexas determina as componentes fortemente conexas de G em tempo $O(V + E)$.

Demonstração

- vamos provar que as k primeiras árvores produzidas na linha 3 correspondem a componentes fortemente conexas
- a prova é por indução em k
- quando $k = 0$, a afirmação é trivial, então tome $k \geq 1$
- suponha que as primeiras $k - 1$ primeiras árvores produzidas correspondem a componentes

Prova do teorema (cont)

Considere a k-ésima árvore produzida pelo algoritmo

- seja u a raiz dessa árvore de busca
- e seja C a componente fortemente conexa que contém u
- vamos mostrar que a árvore produzida contém todos os vértices de C e somente os vértices de C
- isso completará a indução e a prova do teorema

Prova do teorema (cont)

A árvore contém todos vértices de C

- considere o instante em que u é descoberto
- por indução nenhum vértice de C foi finalizado
- então nesse instante $d[u]$ os vértices de C são brancos
- assim, todos os vértices de C tornam-se descendentes de u na árvore de busca de G^T

A árvore contém somente vértices de C

- suponha que existe aresta (u, v) que sai de C
- seja D a componente fortemente conexa que contém D
- pelo corolário do Lema 2, temos $f(C) < f(D)$
- então descobrimos vértices de D antes de u
- por indução, todos vértices de D já foram finalizados
- portanto, a árvore só contém vértices de C