Projeto e Análise de Algoritmos

Hamilton José Brumatto

Bacharelado em Ciência da Computação - UESC

21 de maio de 2019

Busca em Grafos

Buscas em Grafos

- Grafos são estruturas mais complexas do que listas, vetores ou árvores.
- Uma busca em grafos implica em uma visita a todos seus vértices ou possivelmente a todas suas arestas.
- Algoritmos eficientes de busca podem resolver problemas modelados na forma de grafos também de forma eficiente.
- Um método de busca deve ser capaz de percorrer/explorar um grafo seja ele orientado ou não.

Métodos de Busca

- Dois são os métodos de busca:
- Busca em Largura: ou BFS *Breadth-first search*

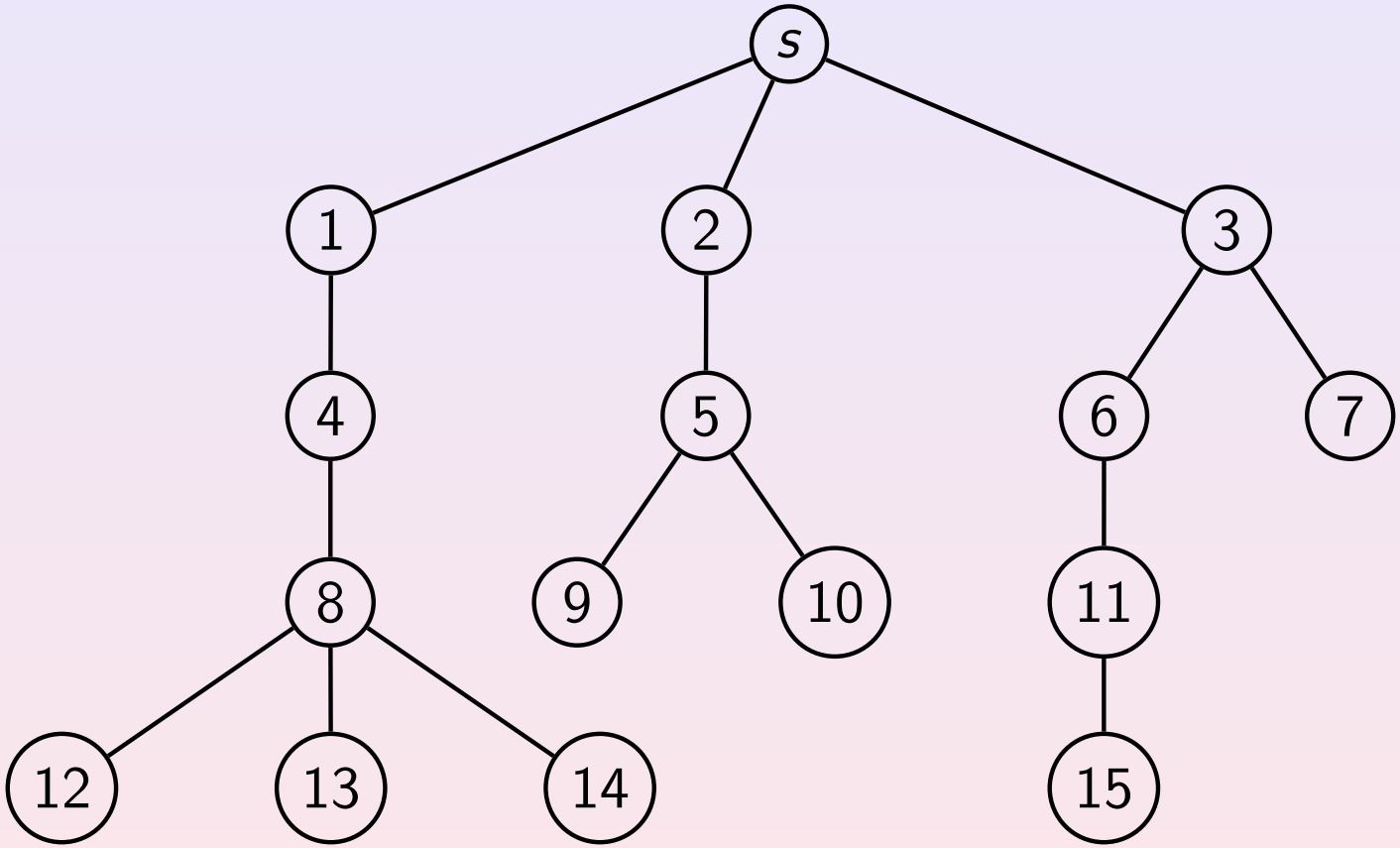
  Iniciada a busca por um vértice s, explora-se todos seus vizinhos: vértices cuja distância mínima é 1 e depois os vizinhos de cada vizinho, ou seja, vértices cuja distância é 2 e assim sucessivamente.
- Busca em Profundidade: ou DFS *Depth-first search*

  Iniciada a busca por um vértice s, explora-se um vizinho, o vizinho deste e assim sucessivamente. Ao concluir a busca dos vizinhos de um vértice, retorna-se ao vértice anterior para explorar outros vizinhos.

Exemplo da busca em largura em uma árvore

- Vamos considerar um grafo árvore enraizada com raiz s: $T(s)$.
- Nosso algoritmo tem início em s, em seguida visita todos os vizinhos de s.
- Terminados os vizinhos de s, visitamos todos os vizinhos dos vizinhos que já foram visitados.
- Na figura do próximo slide apresentamos uma árvore onde os nós são numerados de acordo com a ordem da visita.

Seqüência da busca em largura em uma árvore

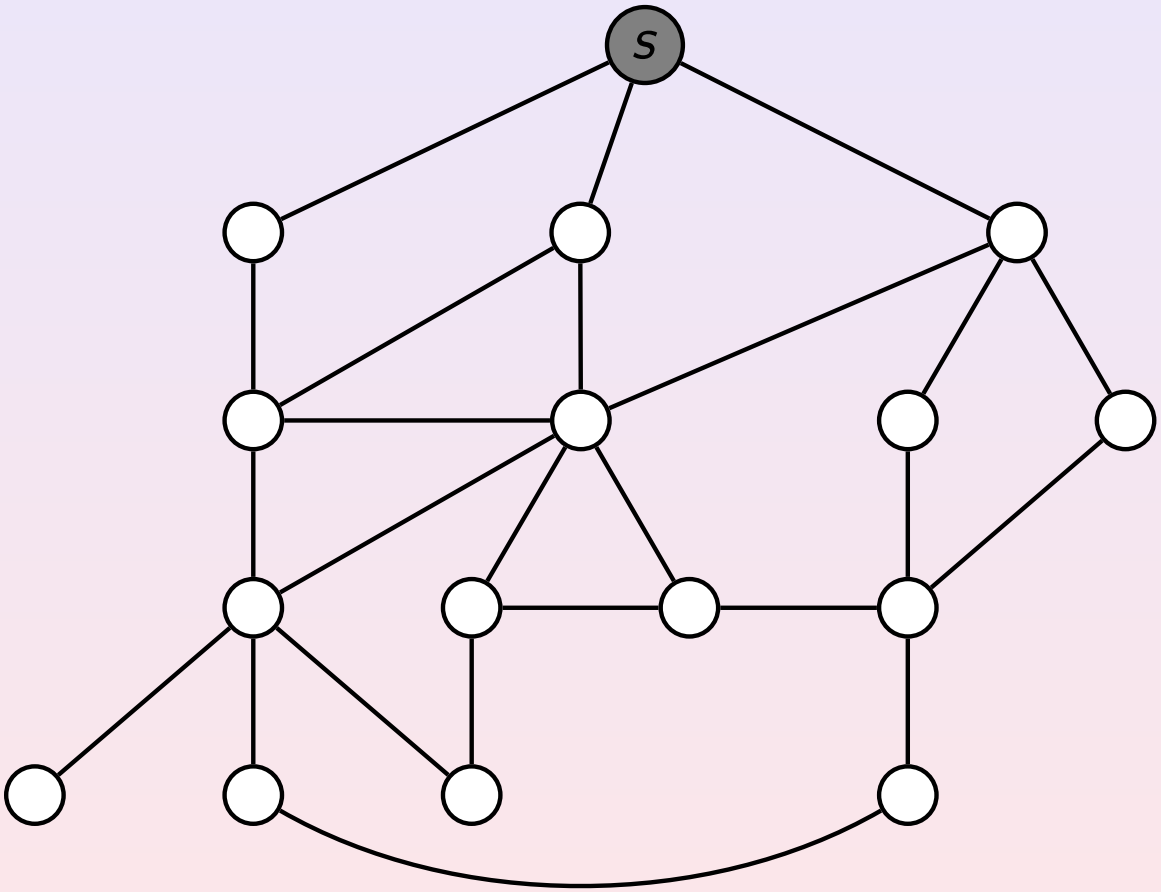


Os números indicam a ordem na qual os vértices são visitados. Ao visitar um vértice que tenha uma distância k de s, significa que todos os vértices com distância $< k$ já foram visitados.

Seqüência em um grafo mais geral

- O processo é o mesmo, somente é necessário organizar os vértices em uma fila, na ordem em que será visitado.
- A ordem é definida pela distância de s ao vértice. Sendo a distância o caminho mais curto de s até o vértice.
- Inicialmente o grafo está intocado, todos vértices são "brancos". Marco o vértice s de "cinza" e coloco na fila para ser visitado.
- Iterativamente, retiro um vértice da fila e visito ele.
- Ao visitar um vértice, coloco todos os seus vizinhos que ainda não foram marcados na fila, marcando-os de "cinza", e ao terminar com este vértice, pinto ele de "preto".

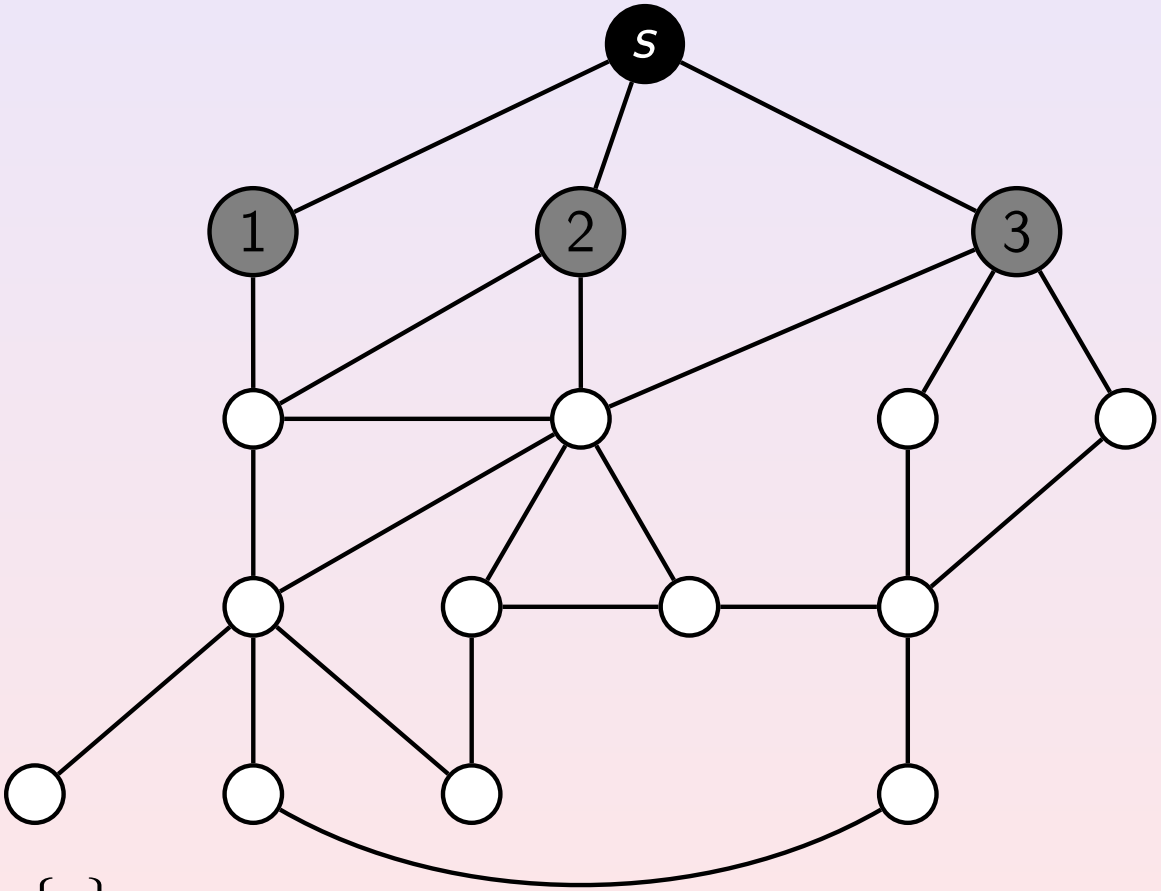
Busca em largura no grafo: Nó s inserido na fila



Visitados: $\emptyset$

Fila: $\{s\}$

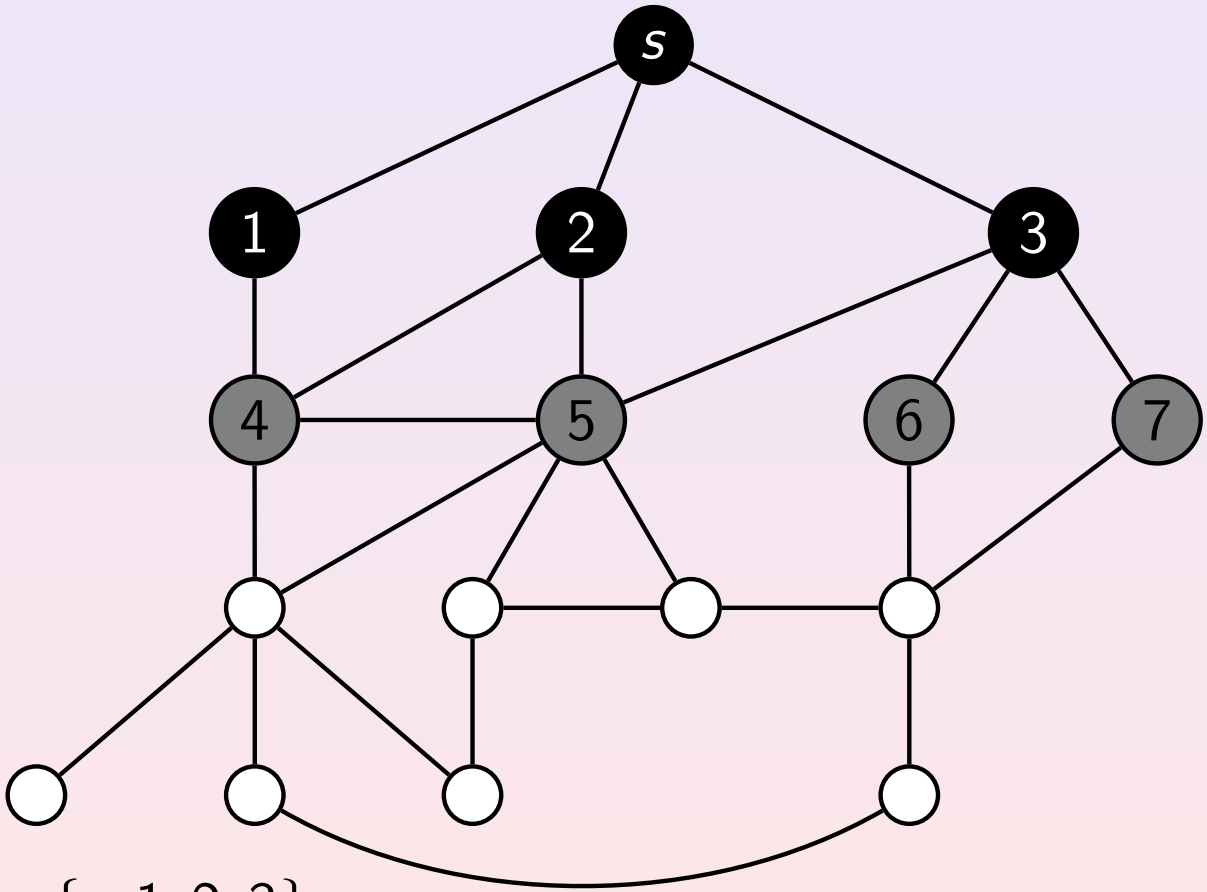
Busca em largura no grafo: Vizinhos de s inseridos na fila (distância: 1)



Visitados: $\{s\}$

Fila: $\{1, 2, 3\}$

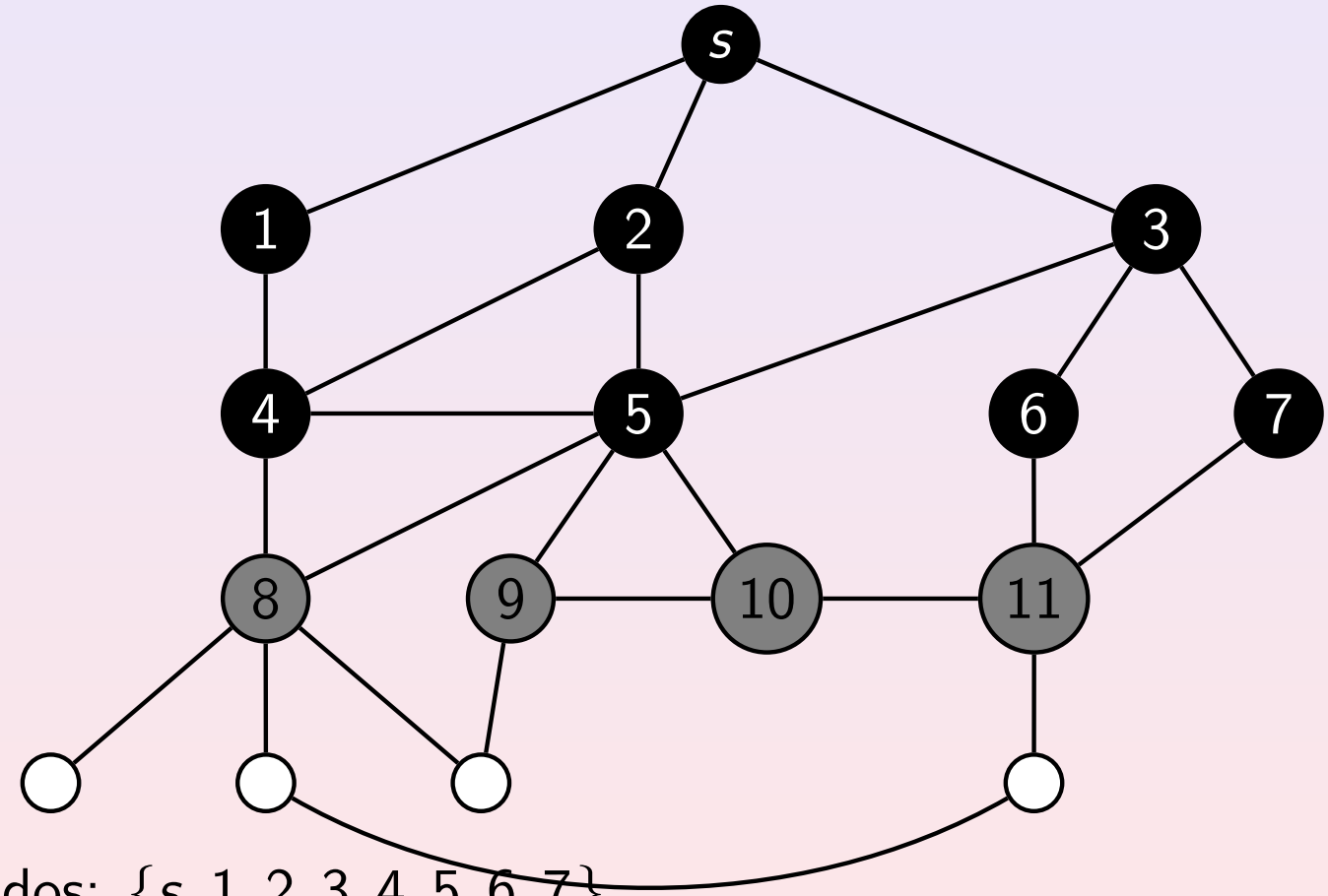
Busca em largura no grafo: (distância: 2)



Visitados: $\{s, 1, 2, 3\}$

Fila: $\{4, 5, 6, 7\}$

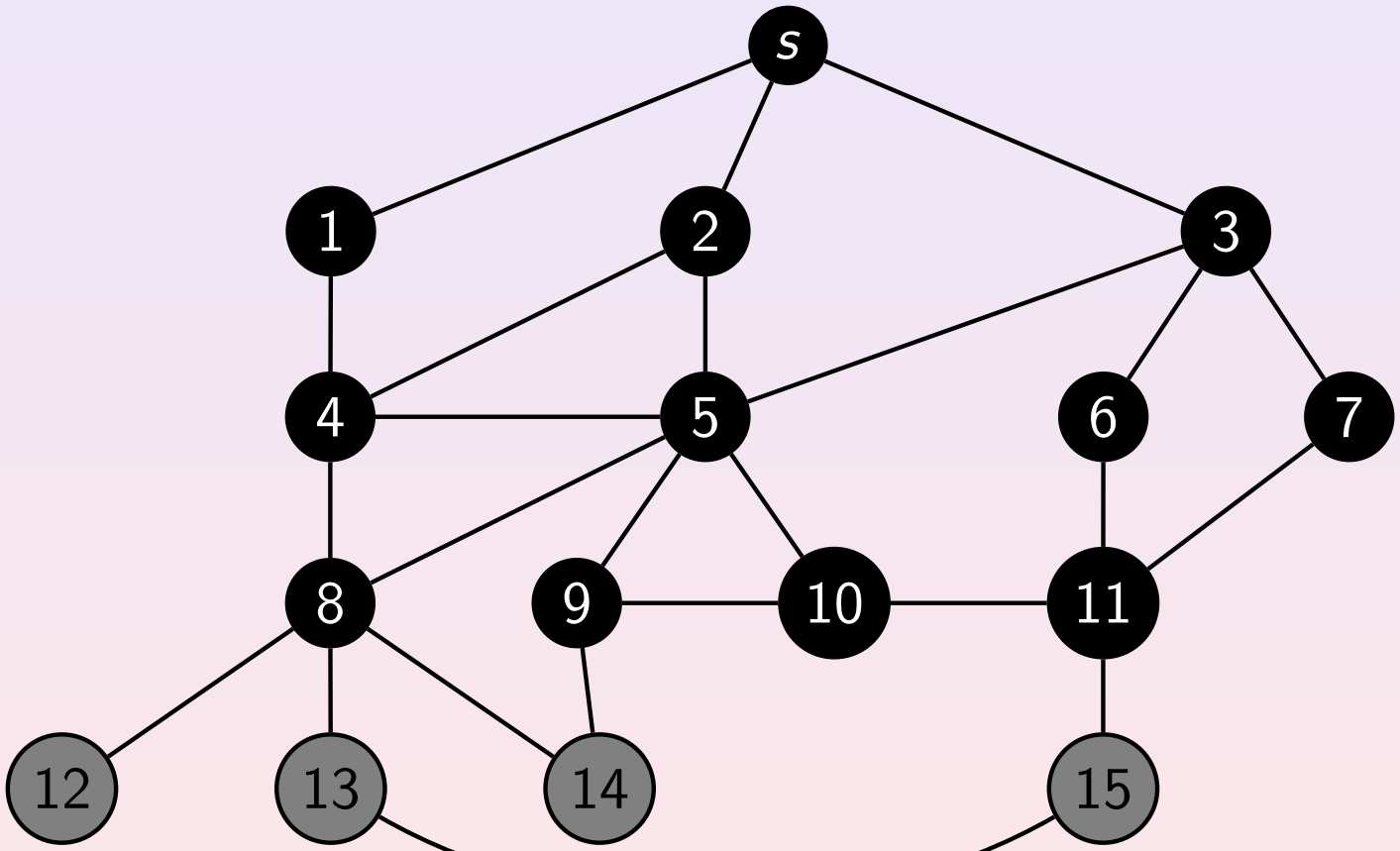
Busca em largura no grafo: (distância: 3)



Visitados: $\{s, 1, 2, 3, 4, 5, 6, 7\}$

Fila: $\{8, 9, 10, 11\}$

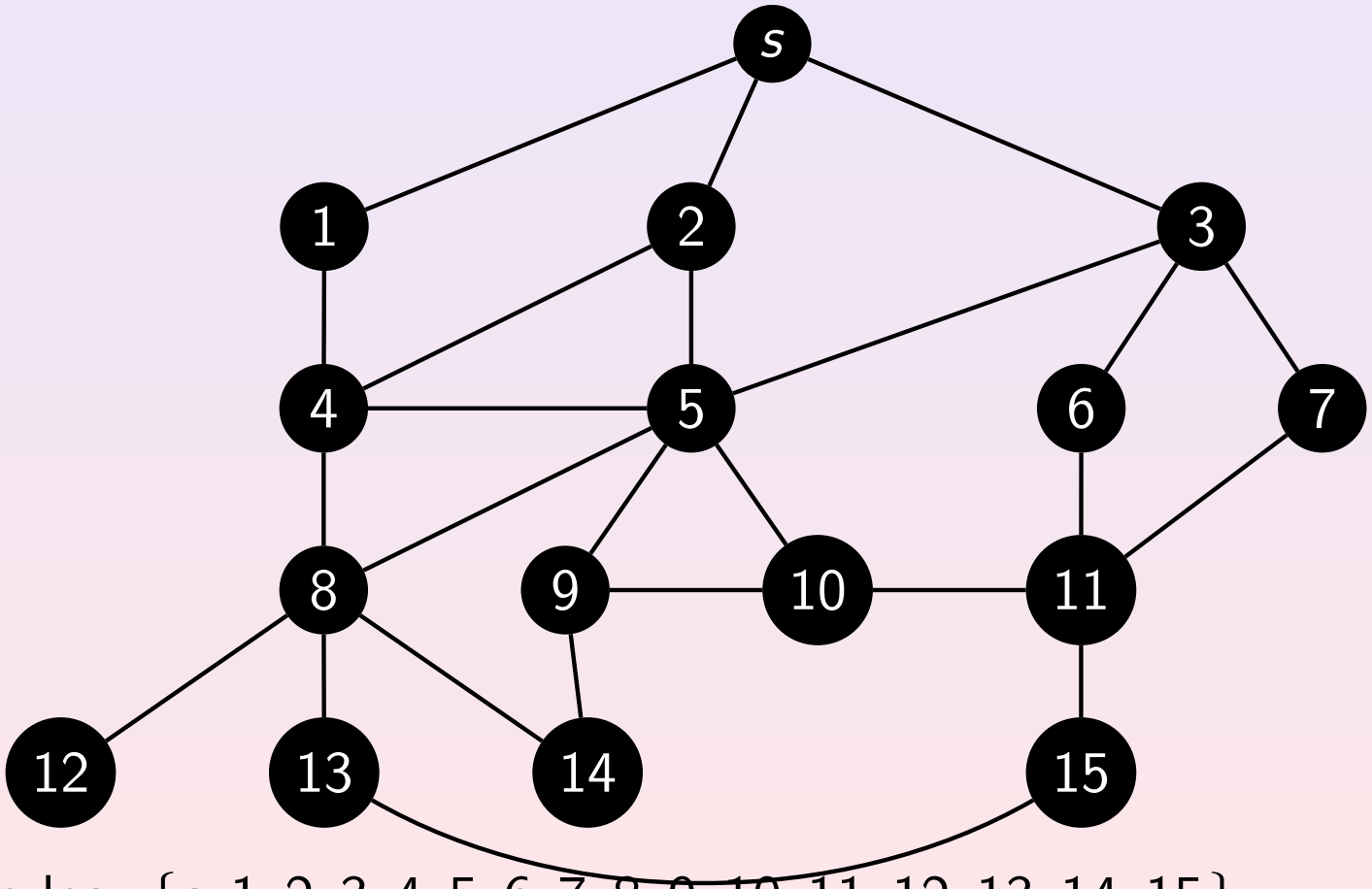
Busca em largura no grafo: (distância: 4)



Visitados: $\{s, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$

Fila: $\{12, 13, 14, 15\}$

Busca em largura no grafo: Algoritmo termina quando a fila está vazia



Visitados: $\{s, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$

Fila: $\emptyset$

Observações sobre o algoritmo

- Automaticamente é construída uma árvore de busca em largura no grafo.
- Cada vértice ao marcar outro marca-o como filho na árvore. Também é possível guardar a informação de que é seu pai.
- Vértices com a mesma distância podem ser percorridos em qualquer ordem. A árvore pode mudar mas não irá alterar a distância do vértice.
- Para efeito de implementação do algoritmo, marcar um vértice com cores distintas pode não ser necessário, porém facilita a análise.

Definições para o algoritmo de busca em largura

- Raiz da busca é s.
- Cada vértice v (diferente de s) possui um *pai*: $\pi[v]$
- O caminho de s a v na Árvore é dado por:

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

- A distância de s a v é dada por $d[v]$
- Este algoritmo foi escrito para um grafo representado na forma de lista de adjacência

Algoritmo de Busca em Largura: BFS

Algoritmo BuscaEmLargura(G,s)

para $u \in V[G] - \{s\}$ **faça**

$cor[u] \leftarrow branco$; $d[u] \leftarrow \infty$; $\pi[u] \leftarrow nulo$

$cor[s] \leftarrow cinza$; $d[s] \leftarrow 0$; $\pi[s] \leftarrow nulo$

$Q \leftarrow \emptyset$ ▷ Fila

$Enfileira(Q, s)$

enquanto $Q \neq \emptyset$ **faça**

$u \leftarrow Desenfileira(Q)$

para $v \in Adj[u]$ **faça**

se $cor[v] = branco$ **então**

$d[v] \leftarrow d[u] + 1; \pi[v] \leftarrow u$

$cor[v] \leftarrow cinza$

$Enfileira(Q, v)$

$cor[u] \leftarrow preto$

Análise do algoritmo: Complexidade do tempo

- A inicialização consome o tempo $\Theta(V)$.
- Depois que um vértice deixa de ser branco, ele não volta a ser branco. Cada vértice é inserido na fila uma vez. Operação na fila é $\Theta(1)$ total de $O(V)$
- Em uma lista de adjacência, cada vértice é percorrido apenas uma vez. A soma dos comprimentos das listas é $\Theta(E)$. O tempo para percorrer a lista $O(E)$.
- A complexidade em tempo do algoritmo BuscaEmLargura é $O(V + E)$.

Análise do algoritmo: Corretude

É necessário provar duas situações

- Seja $dist(u, v)$ a distância de u a v. É necessário mostrar que $d[v] = dist(s, v)$
- É necessário mostrar que a função "pai" $\pi[]$ define uma Árvore de Busca em Largura com raiz s.

Desta forma, o algoritmo percorre o grafo pesquisando os vértices de menor distância primeiro, tal qual uma busca em largura na árvore.

Prova da corretude: Lema 1

Lemma

Seja G um grafo e $s \in V[G]$.
Então para toda aresta (u, v) temos que:

$$dist(s, v) \leqslant dist(s, u) + 1$$

Demonstração.

Imediato, se $dist(s, v) \leqslant dist(s, u)$ não há o que provar. Então supomos que $dist(s, v) > dist(s, u)$. Como $dist(u, v) = 1$ então, no mínimo, $dist(s, v) = dist(s, u) + 1$. □

Prova da corretude: Lema 2 - $d[v]$ é uma estimativa superior de $dist(s, v)$

Lemma

Durante a execução do algoritmo vale o seguinte invariante: $d[v] \geqslant dist(s, v)$ *para todo* $v \in V[G]$

Demonstração.

Indução no número de operações *Enfileirar*

Base: quando s é inserido na fila temos $d[s] = 0 = dist(s, s)$ e $d[v] = \infty \geqslant dist(s, v)$ para $v \in V - \{s\}$.

Passo de Indução: v é descoberto enquanto a busca é feita em u (percorrendo $Adj[u]$). Então:

$d[v] = d[u] + 1 \geqslant dist(s, u) + 1$ (Hip. Ind.)

$d[v] \geqslant dist(s, v) + 1$ (Lema 1).

Como $d[v]$ nunca muda após ser inserido na fila, o invariante vale.

Prova da corretude: Lema 3

Lemma

Suponha que $(v_1, v_2, \ldots, v_r)$ *seja a disposição da fila* Q *na linha 6 em uma iteração qualquer, Então:*

$$d[v_r] \leqslant d[v_1] + 1, \; e$$

$$d[v_i] \leqslant d[v_i + 1]$$

Em outras palavras, os vértices são inseridos na fila em ordem crescente e há no máximo dois valores de $d[v]$ *para vértices na fila.*

Prova da corretude: Prova do Lema 3

Demonstração.

Indução no número de operações *Desenfileira* e *Enfileira*.
Base: $Q = \{s\}$. O lema vale trivialmente.
Hipótese da Indução: Antes das operações o Lema é válido, v_1 é removido de Q. Agora v_2 é o primeiro vértice de Q. Então

$$d[v_r] \leqslant d[v_1] + 1 \leqslant d[v_2] + 1$$

Passo da Indução: Novo elemento é enfileirado. $v = v_{r+1}$ é inserido em Q. Como v é vizinho de $u = v_1$, $d[v] = d[v_1] + 1$.

$$d[v_r] \leqslant d[u] + 1 = d[v] = d[v_{r+1}], \text{ e}$$

$$d[v_{r+1}] \leqslant d[v_1] + 1 \leqslant d[v_2] + 1$$

As outras desigualdades se mantém. □

Prova da corretude do algoritmo: Teorema da corretude

Theorem

Seja G um grafo e $s \in V[G]$.
Então após a execução de BuscaEmLargura,

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

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

Demonstração.

Note que se $dist(s, v) = \infty$ então $d[v] = \infty$ pelo Lema 2. Então vamos considerar o caso em que $dist(s, v) < \infty$. Vamos provar por indução em $dist(s, v)$ que $d[v] = dist(s, v)$ □

Prova da corretude do algoritmo: Prova do Teorema da corretude

Demonstração.

Base: Se $dist(s, v) = 0$ então $v = s$ e $d[s] = 0$.
Hipótese de Indução: $d[u] = dist(s, u)$ para u com $dist(s, u) < k$.
Passo da Indução: Seja v um vértice com $dist(s, v) = k$. No caminho mínimo de s a v em G u é o vértice que antecede v. $dist(s, u) = k - 1$.
u é removido da fila Q, v é branco, cinza ou preto.

- Se v é branco, então $d[v] = d[u] + 1 = k = dist(s, v)$
- Se v é cinza, então v foi visitado antes por algum vértice w. $d[w] \leqslant d[u] = k - 1$. Logo $d[v] \leqslant k$, mas pelo Lema 2, $d[v] \geqslant dist(s, v) = k$. Então $d[v] = k = dist(s, v)$.
- Se v é preto, então v já passou pela fila Q e pelo lema 3, $d[v] \leqslant d[u] = k - 1$, mas pelo lema 2, $d[v] \geqslant dist(s, v) = k$, contradição, este caso não é possível.

Caminho mais curto

Algoritmo ImprimeCaminho(G,s,v)

se $v = s$ **então**

imprime s

senão

se $\pi[v] = nulo$ **então**

imprime "não existe caminho de s a v

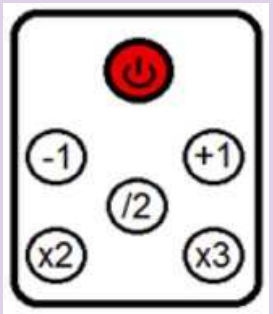
senão

$ImprimeCaminho[G, s, \pi[v])$

imprime v.

Problema URI - 1910 - Ajude Clotilde

Clotilde ama assistir novelas, porém o controle da sua televisão não é nada convencional. Ele possui apenas seis botões, o botão liga/desliga e cinco botões para mudar de canal.
Os botões de mudança de canal funcionam da seguinte maneira: +1 (avança um canal), -1(volta um canal), x2(vai para o dobro do canal atual), x3(vai para o triplo do canal atual) e /2(vai para a metade do canal atual, esse botão só funciona se o canal atual for par).



Eis o famoso controle de Clotilde.

Os vizinhos de Clotilde costumam visitar sua casa nos finais de semana, assistem televisão e não voltam ao canal de sua novela, assim fazendo com que Clotilde perca muito tempo tentando achar o canal de seu interesse novamente.

Problema URI - 1910 - Entrada/Saída

Sua tarefa é, dado o número do canal atual e o número do canal de interesse de Clotilde, você deve calcular a menor quantidade de cliques nos botões necessária para sair de um e chegar no outro. Lembrando que como Clotilde é uma pessoa séria, ela não gosta de passar por alguns canais específicos, mesmo que tenha que apertar mais botões para chegar no canal destino. Outra restrição é, não existe canal menor igual a 0 ou maior que 105. Ex: Se está no canal 55000, você não pode apertar o botão x2 nem o x3.

Entrada:
Haverá diversos casos de testes. Cada caso inicia com três inteiros, **O**, **D** e **K** ($1 \leqslant$**O**, **D**$\leqslant 10^5$, $0 \leqslant$**K**$\leqslant 100$), representando, respectivamente, o canal de origem, destino, e a quantidade de canais que Clotilde não quer passar. A segunda linha conterá os **K** canais proibidos por Clotilde. É garantido que o canal de origem e destino nunca serão proibidos.
A entrada termina com $\mathbf{O} = \mathbf{D} = \mathbf{K} = 0$, a qual não deve ser processada.

Saída:
Para cada caso, exiba uma única linha, a menor quantidade de cliques nos botões necessária para ir do canal de origem ao destino ou -1 caso seja impossível chegar ao canal de destino devido as restrições de Clotilde.

Problema URI - 1910 - Proposta de solução

- Podemos construir um grafo com o vértice de origem dado por **O**.
- Os vizinhos de um vértice i são: $i/2$, $i+1$, $i-1$, $i*2$ e $i*3$.
- Para saber a menor quantidade de cliques no controle entre o canal de **O** e **D** fazemos uma busca em largura, a partir de **O** até atingir **D**.
- Se não for possível, a saída é -1, se houver forma de alcançar, então o número de cliques é dado pela distância n (para não confundir com **D** o canal de destino).
- O nosso grafo tem no máximo 100.000 vértices numerados de 1 a 100.000 (que poderão ter ligações ou não, já que as ligações partem de **O**)

Parte da solução: BFS - início

```
pair<long long int,bool> c[100001];
bool found=false;
for(int i=1; i< 100001; i++) c[i]=make_pair(0,true);
for(int i=0; i<k; i++) {
  cin >> v;
  c[v].second=false;
}
queue<int> q;
c[o].second=false;
n = 0;
q.push(o);
```

Parte da solução: BFS - fila

```
while(!q.empty() && !found) {
 u = q.front(); q.pop();
 if(u==d) found=true;
 n = c[u].first;
 if(!found) { n++;
  if(!(u&1)) { v = u >> 1;
   if(c[v].second) { c[v].first=n; c[v].second=false; q.push(v); }
  }
  if(u>1) { v = u-1;
   if(c[v].second) { c[v].first=n; c[v].second=false; q.push(v); }
  }
  if(u<100000) { v = u+1;
   if(c[v].second) { c[v].first=n; c[v].second=false; q.push(v); }
  }
  if(u<=50000) { v = u*2;
   if(c[v].second) { c[v].first=n; c[v].second=false; q.push(v); }
  }
  if(u<33334) { v = u*3;
   if(c[v].second) { c[v].first=n; c[v].second=false; q.push(v); }
  }
 }
}
```

Exemplo de Aplicações da Busca em Profundidade

- Determinar os componentes conexos de um grafo.
- Ordenação Topológica.
- Determinar Componentes Fortemente Conexos.
- Subrotina de busca para outros algoritmos.

Busca em Profundidade

- A busca inicia-se em um vértice s
- Escolhe-se um *vizinho* v não visitado de s
- Recursivamente a busca continua a partir de v.
- Concluída a busca em v, tenta-se prosseguir a busca a partir de outro vizinho de s, se não foi possível ela retorna ao nível anterior de recursão. (*backtracking*)

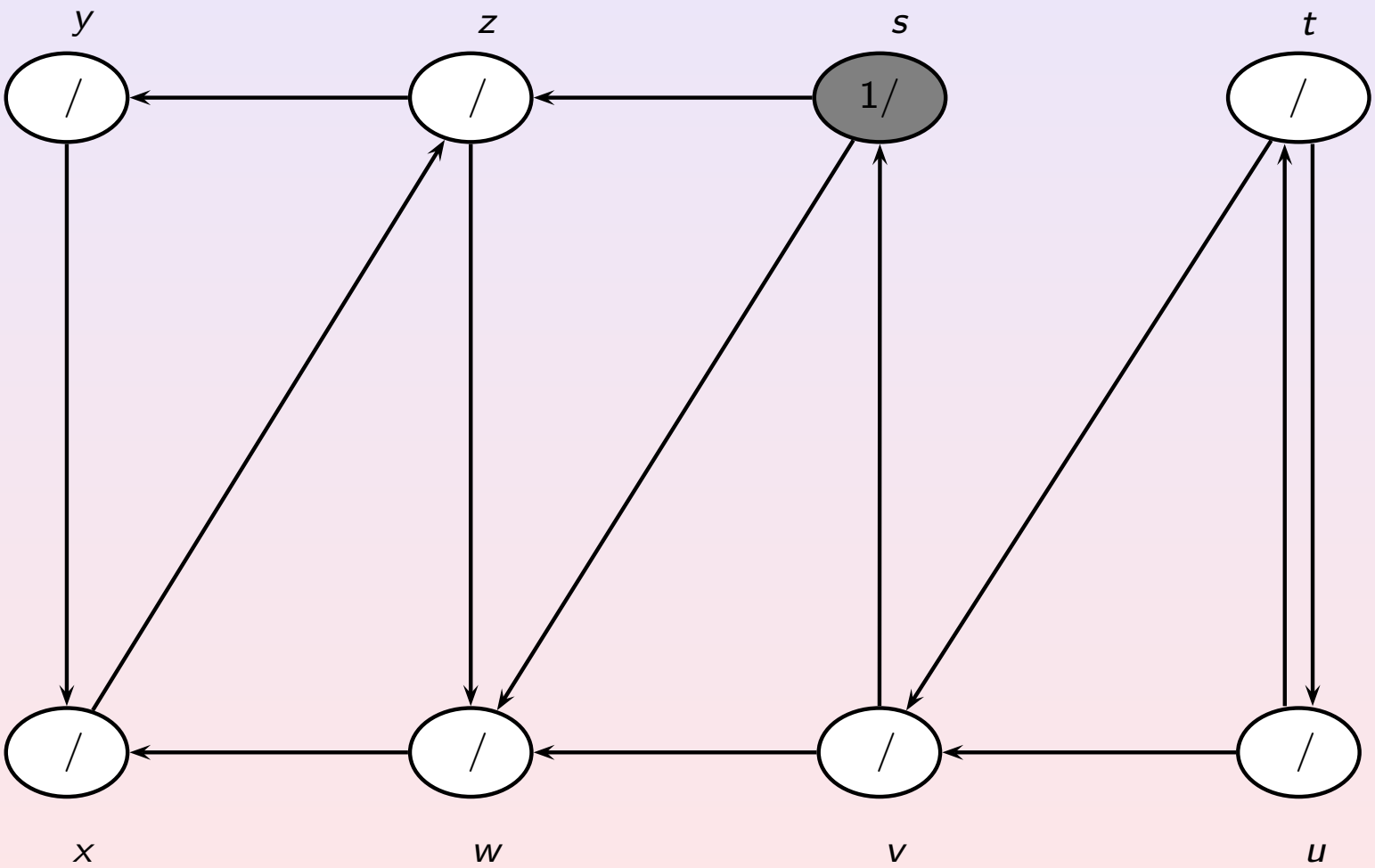
Algoritmo para Busca em Profundidade

- O algoritmo para busca em profundidade pode ser construído de forma iterativa semelhante ao algoritmo de busca em largura.
- Ao invés de se utilizar uma estrutura de FILA para colocar os vértices que serão investigados, utiliza-se uma estrutura de PILHA:
  - A busca atingiu um vértice u.
  - Escolhe-se um vizinho não visitado de u, por exemplo v e o coloque na pilha.
  - Quando acabar os vizinhos de u, pega-se novo vértice da pilha.

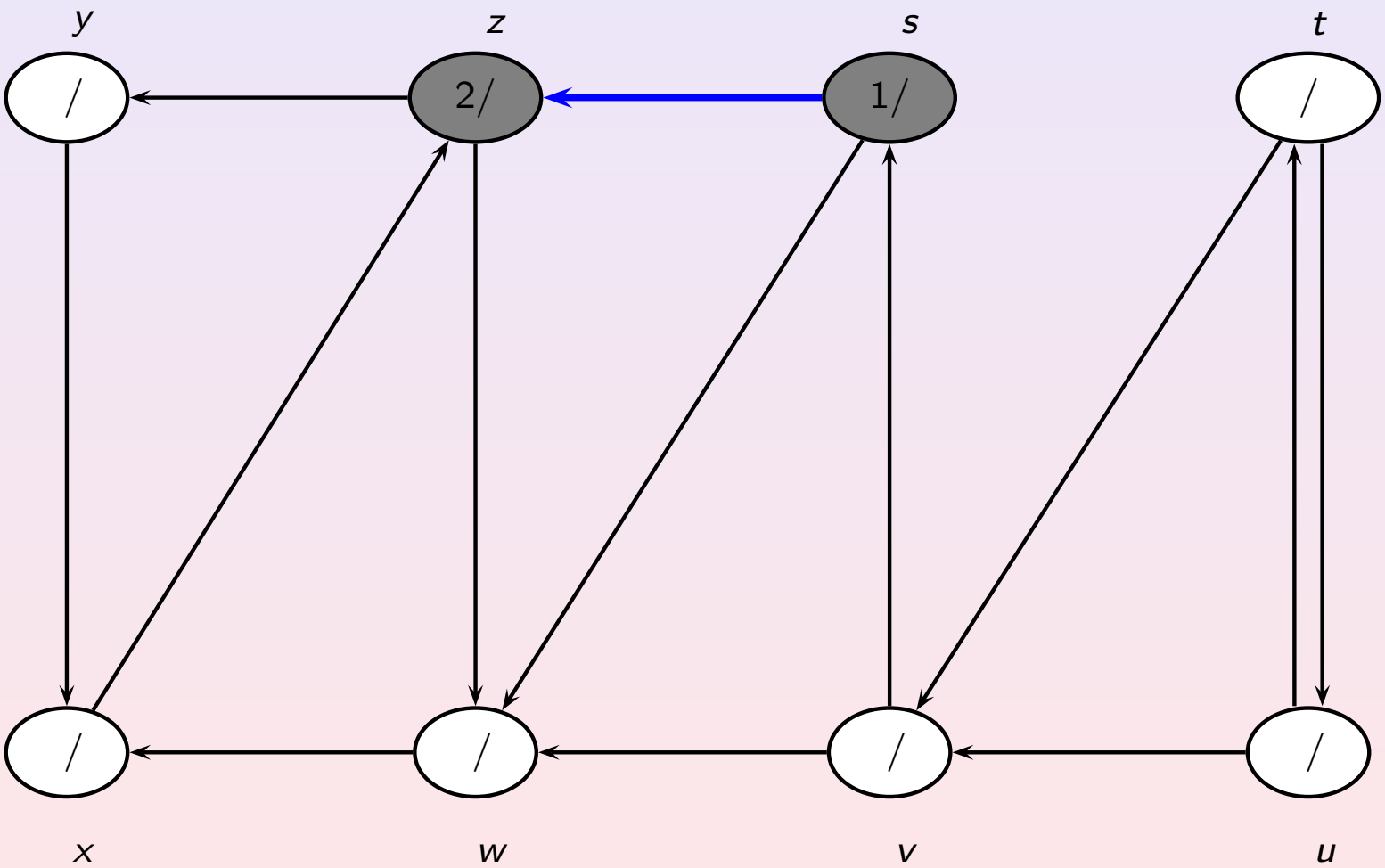
Algoritmo para Busca em Profundidade: Versão Recursiva

- Cada vértice v é associado a um predecessor: $\pi[v]$.
- O subgrafo induzido pelas arestas $\{(\pi[v], v] : v \in V[G] \text{ e } \pi[v] \neq nulo\}$ é a *Floresta de Busca em Profundidade*
- Os vértices são coloridos durante a busca:
  - Branco: Não visitado.
  - Cinza: Visitado mas ainda não finalizado.
  - Preto: Visitado e Finalizado.
- Dois rótulos (inteiros) de *tempo* são definidos:
  - $d[v]$: Instante da descoberta de v (virou cinza)
  - $f[v]$: Instante de finalização de v (virou preto)

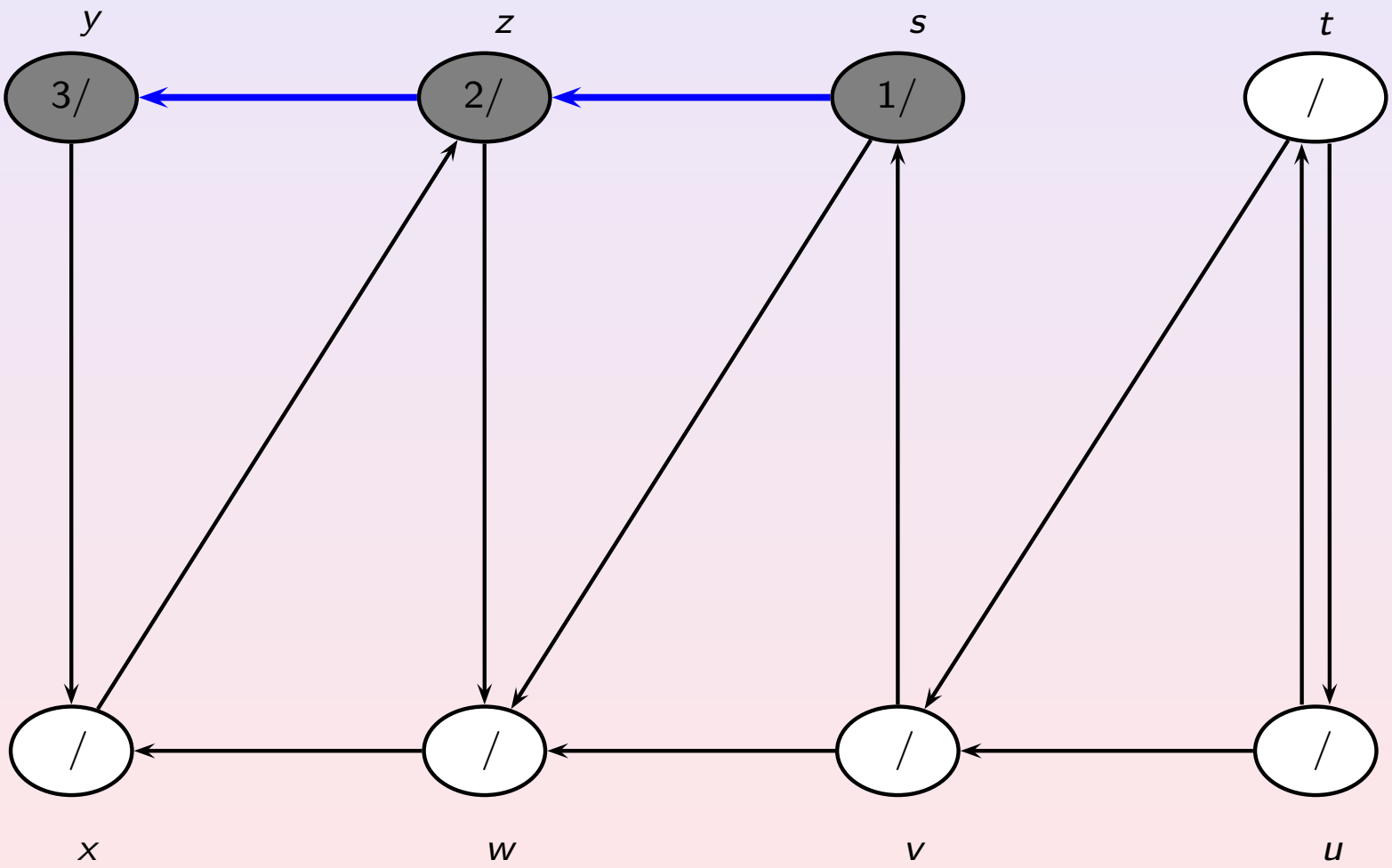
Exemplo de Aplicação do Algoritmo



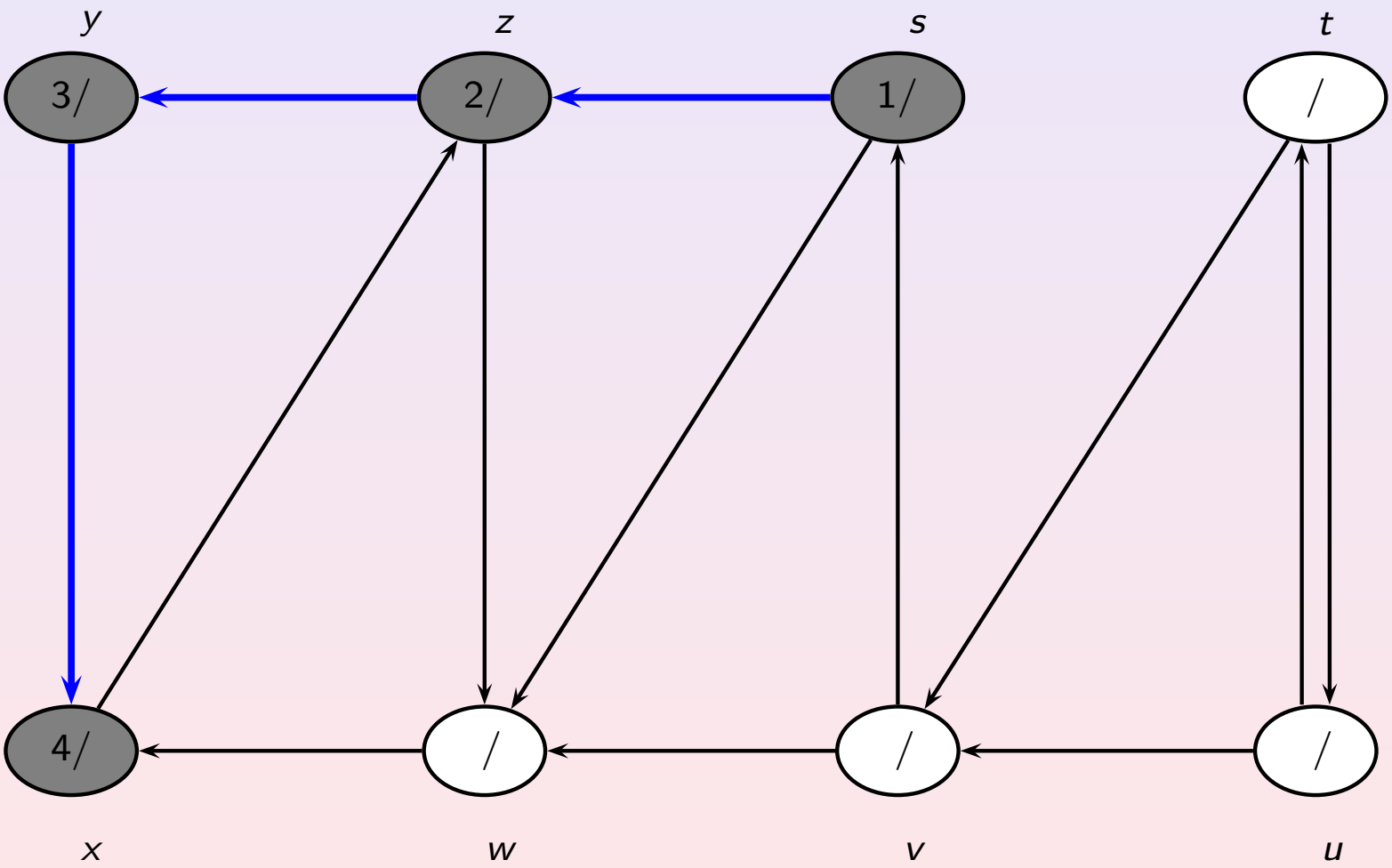
Exemplo de Aplicação do Algoritmo



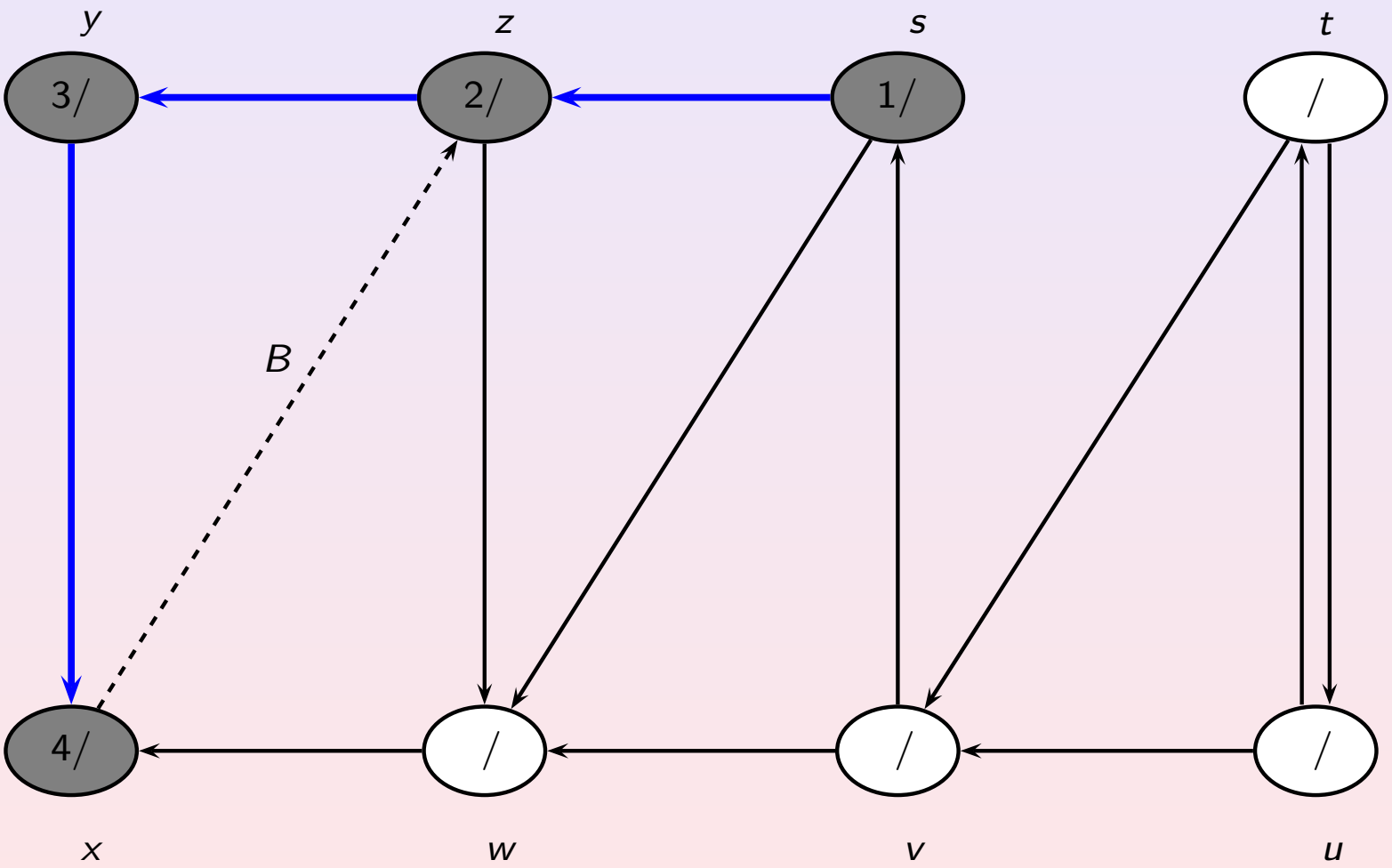
Exemplo de Aplicação do Algoritmo



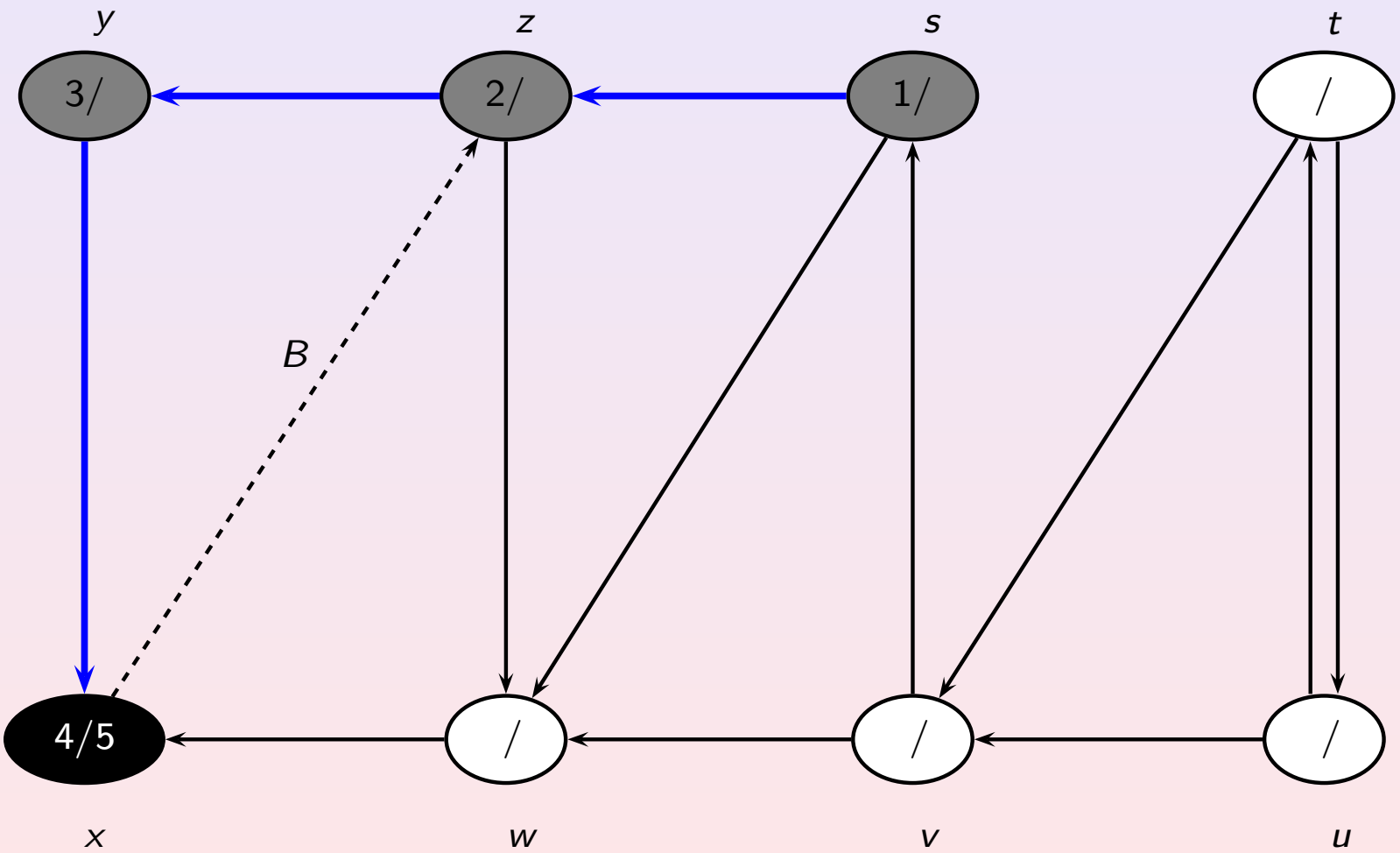
Exemplo de Aplicação do Algoritmo



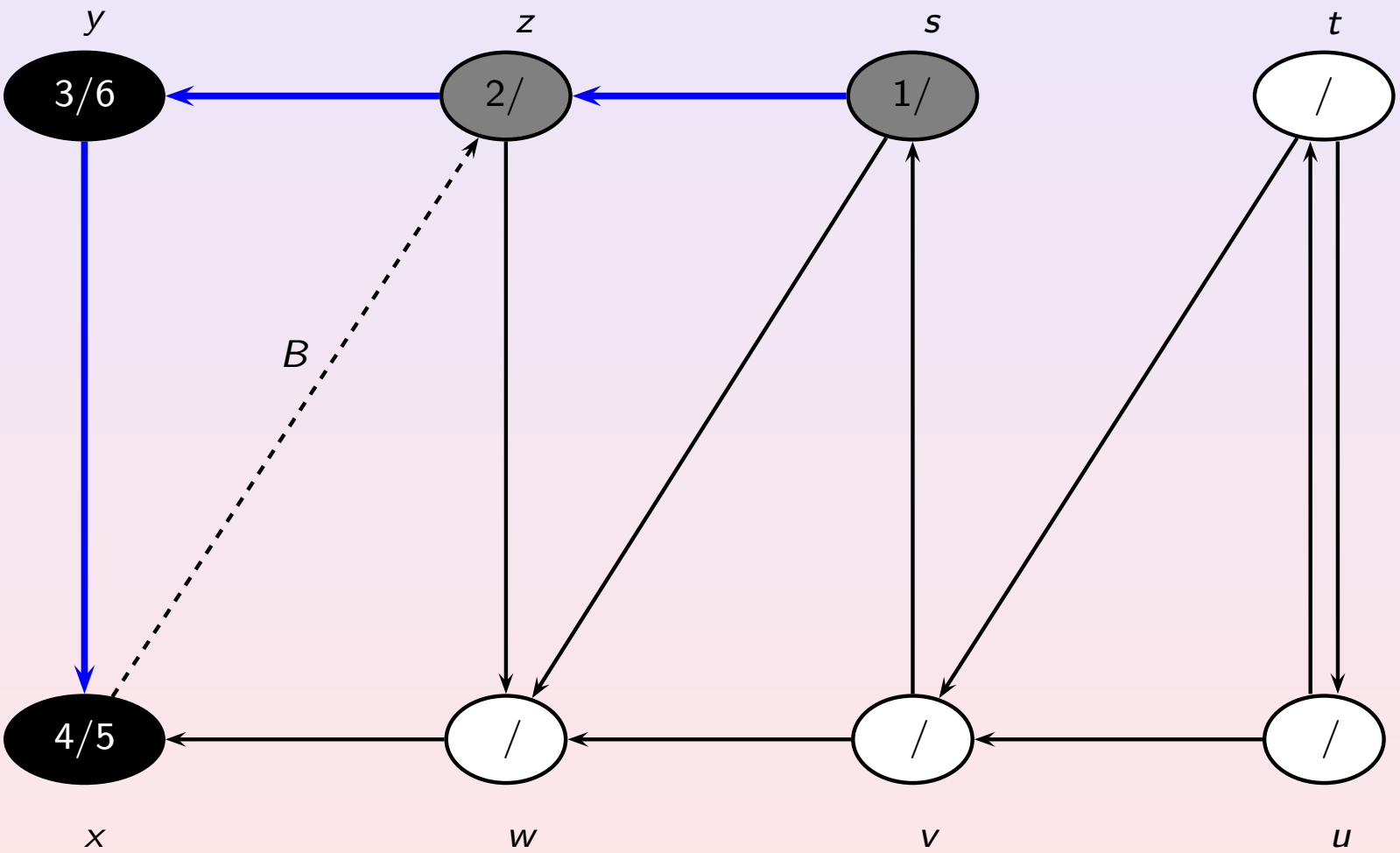
Exemplo de Aplicação do Algoritmo



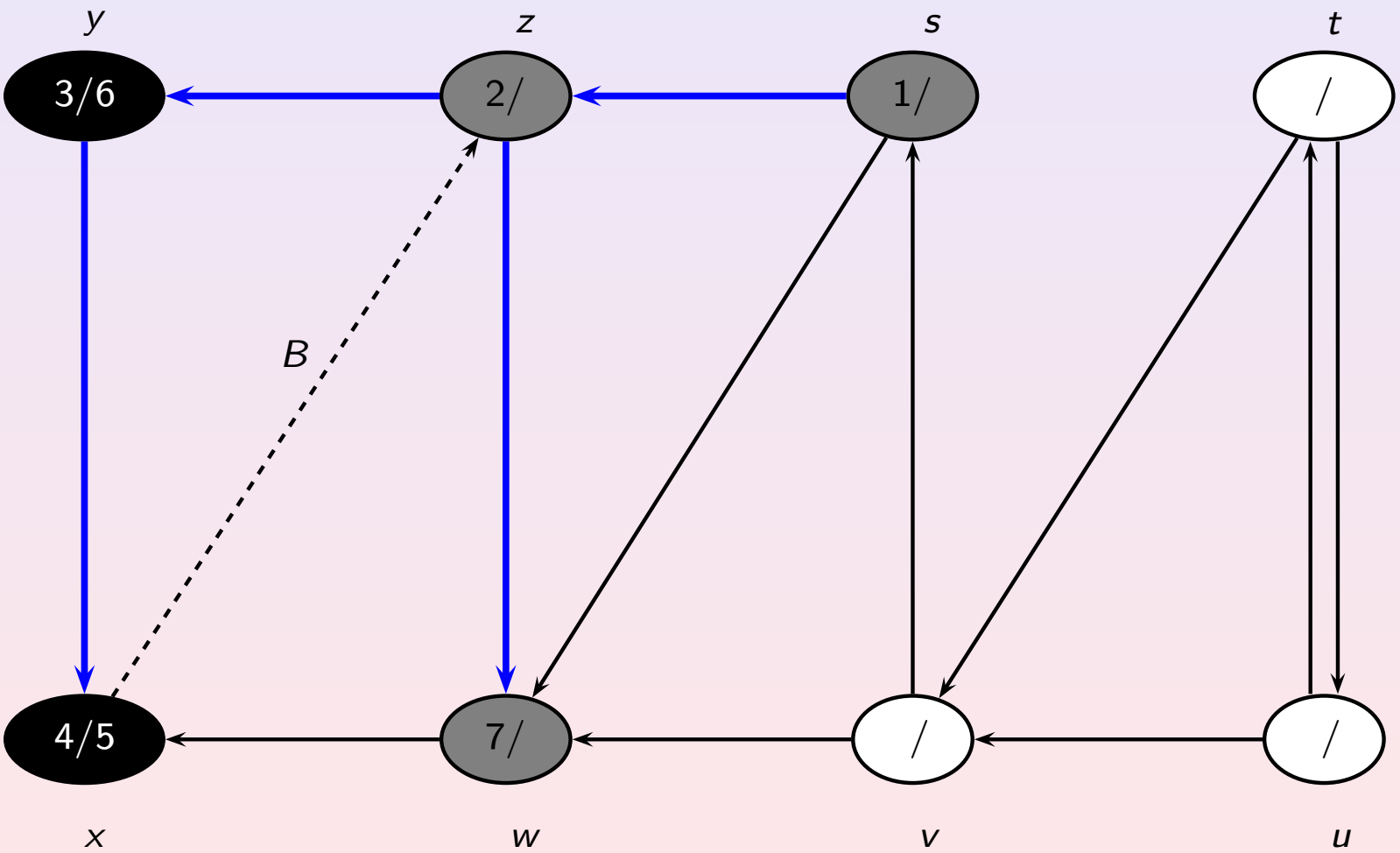
Exemplo de Aplicação do Algoritmo



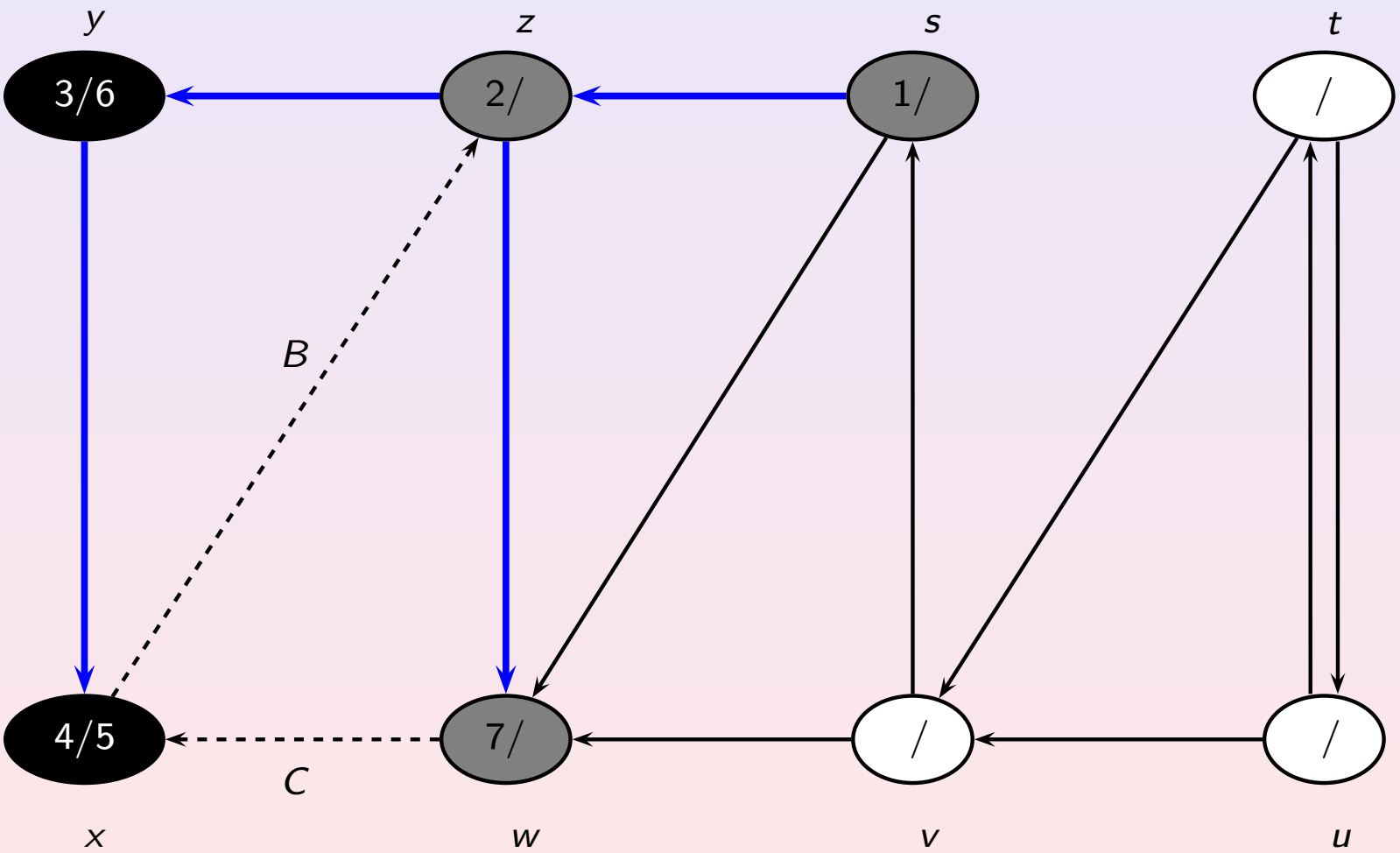
Exemplo de Aplicação do Algoritmo



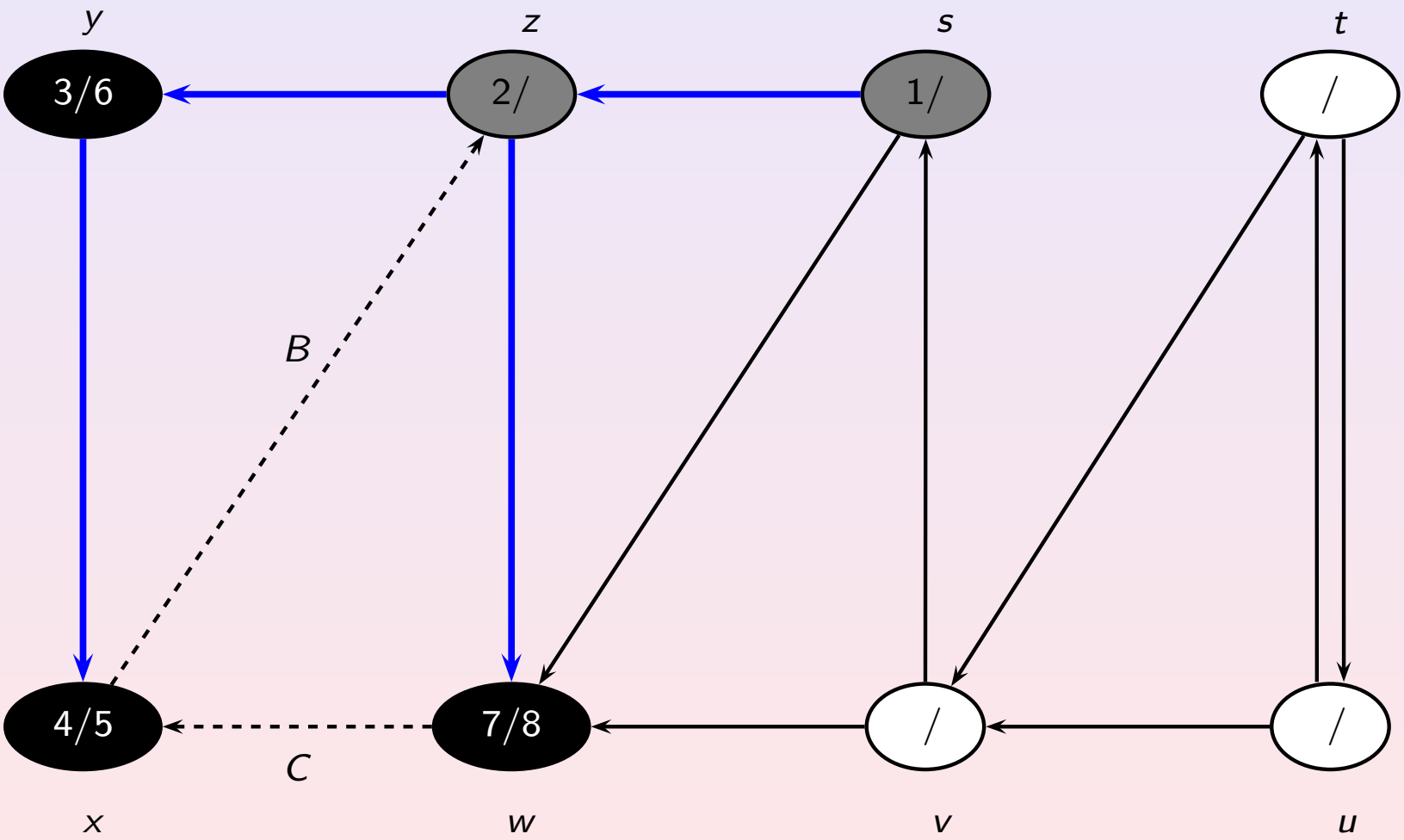
Exemplo de Aplicação do Algoritmo



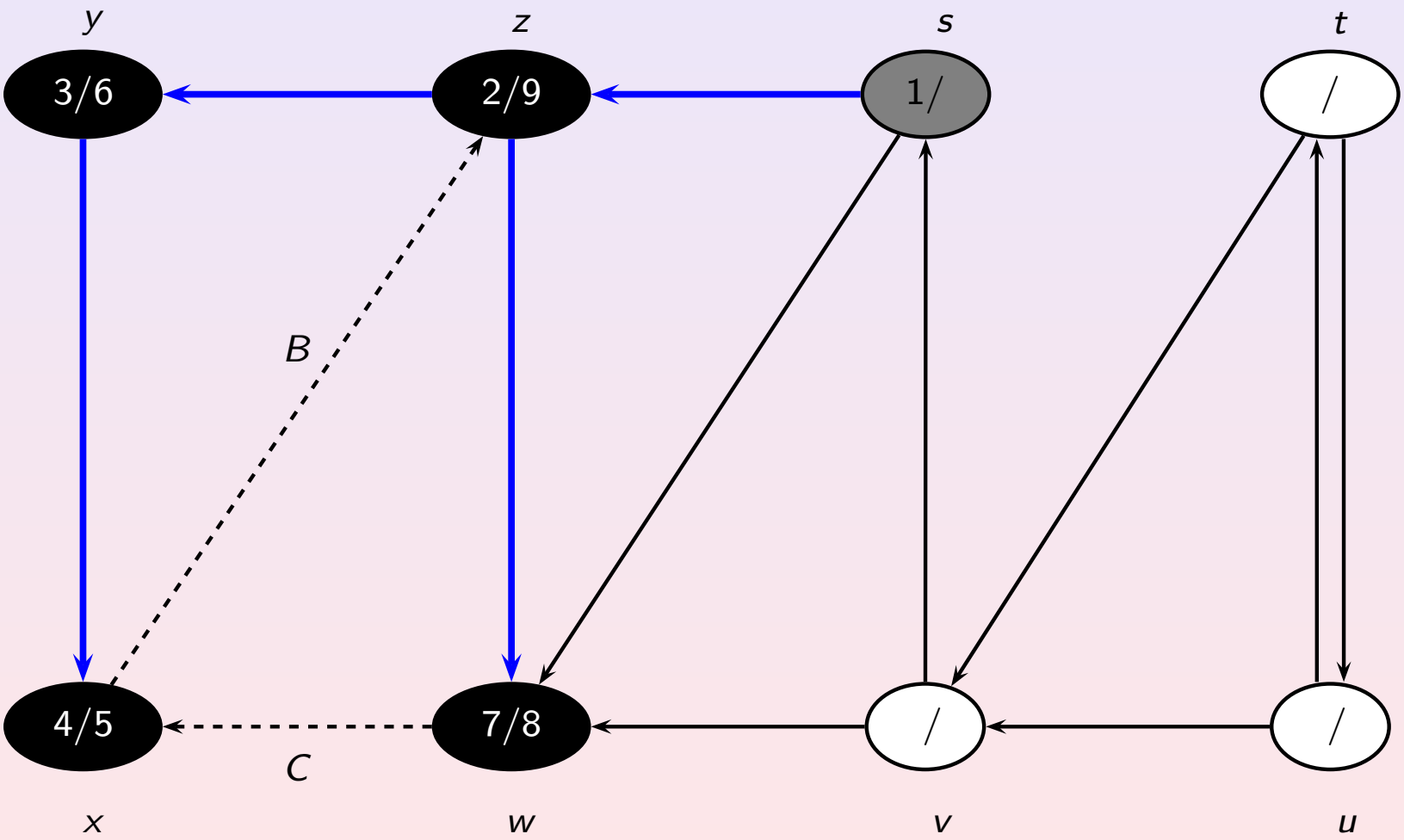
Exemplo de Aplicação do Algoritmo



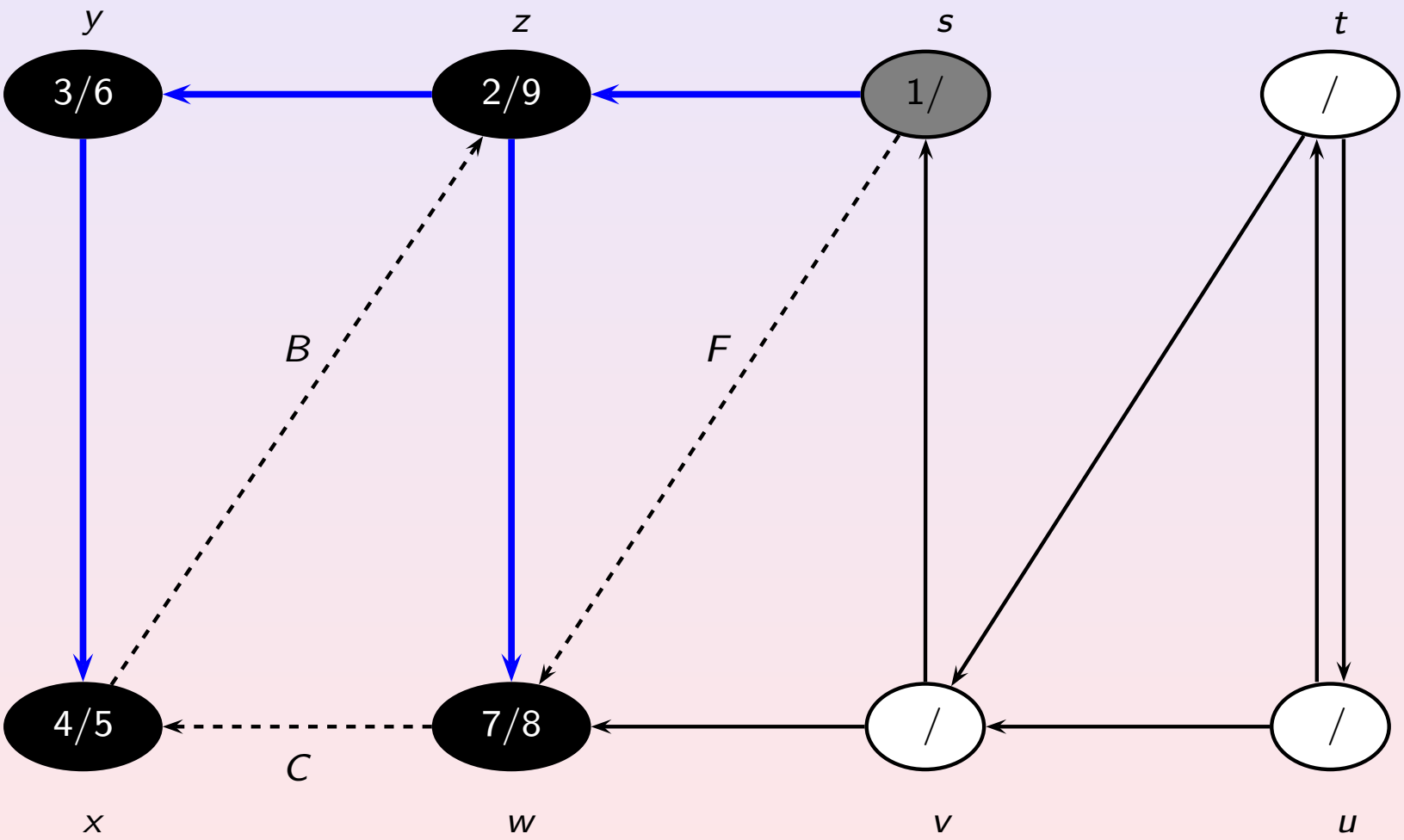
Exemplo de Aplicação do Algoritmo



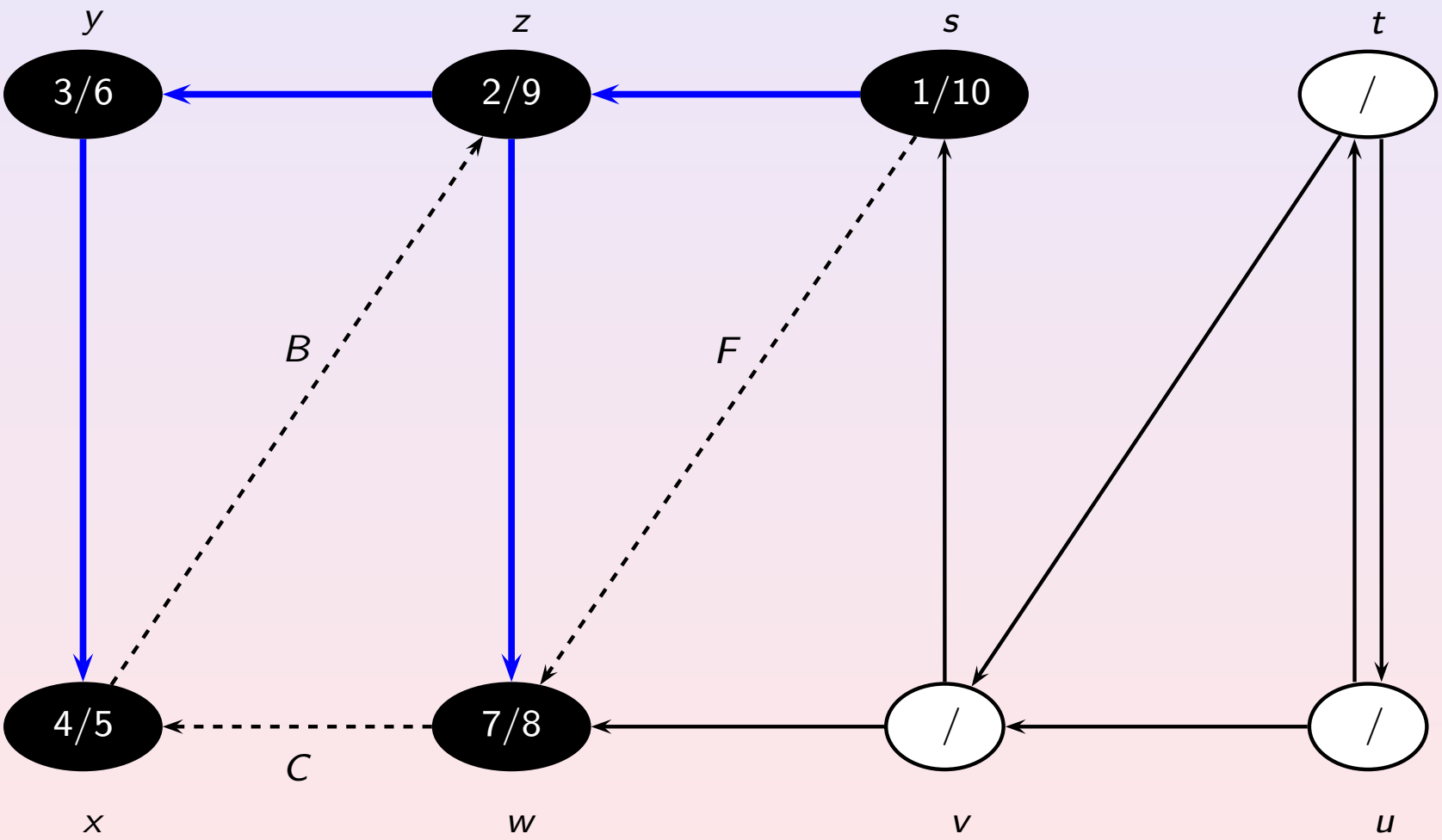
Exemplo de Aplicação do Algoritmo



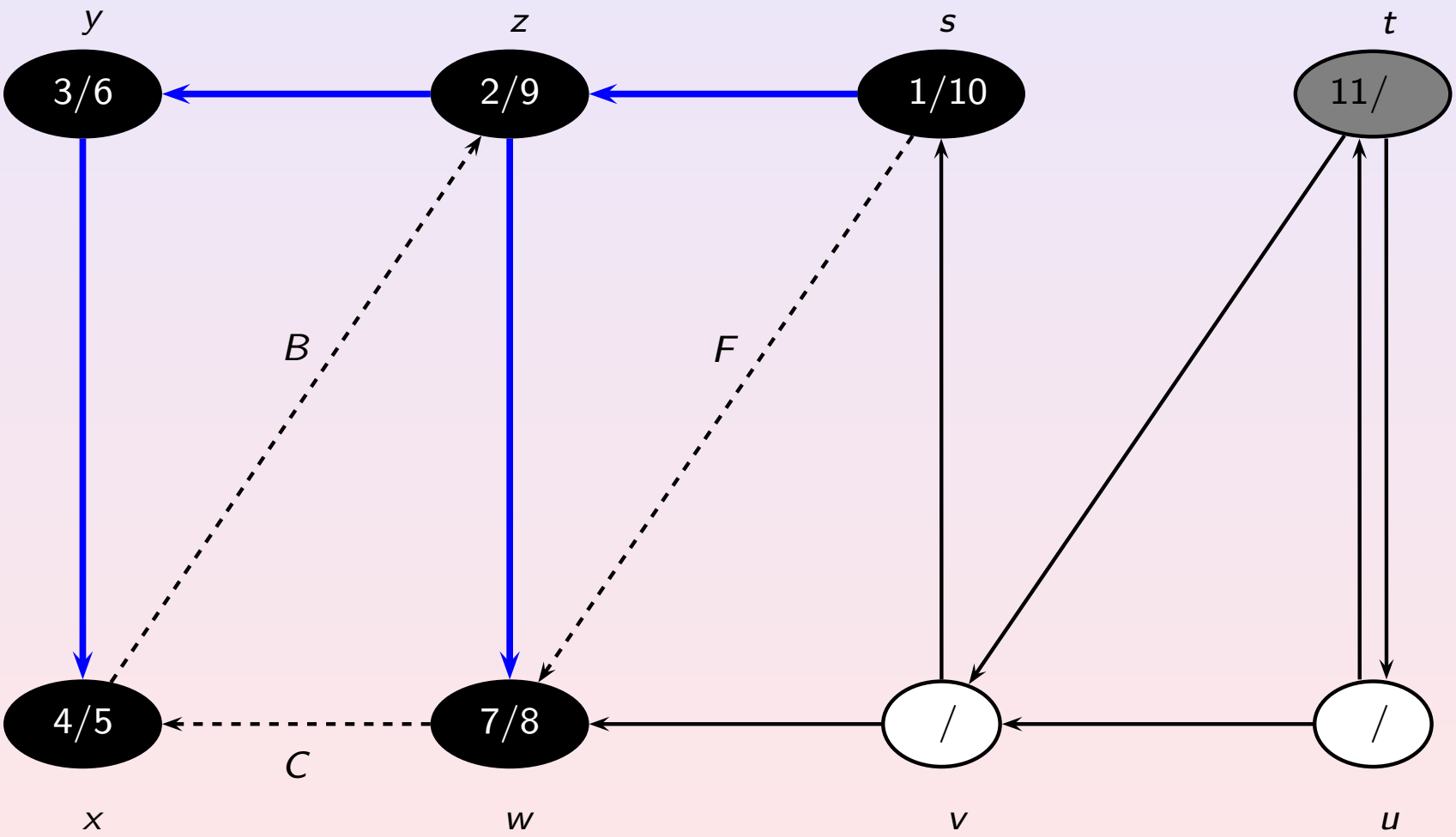
Exemplo de Aplicação do Algoritmo



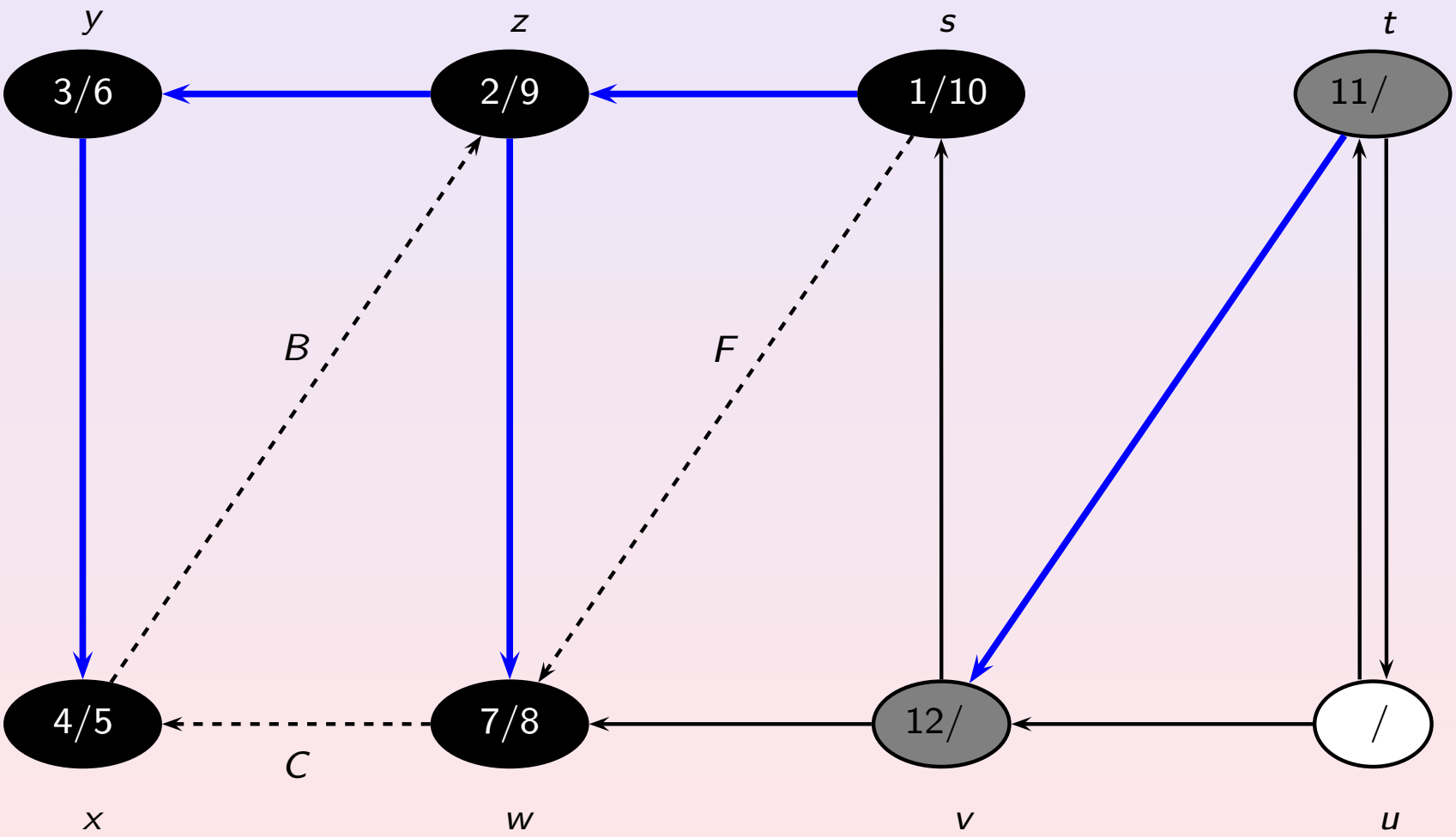
Exemplo de Aplicação do Algoritmo



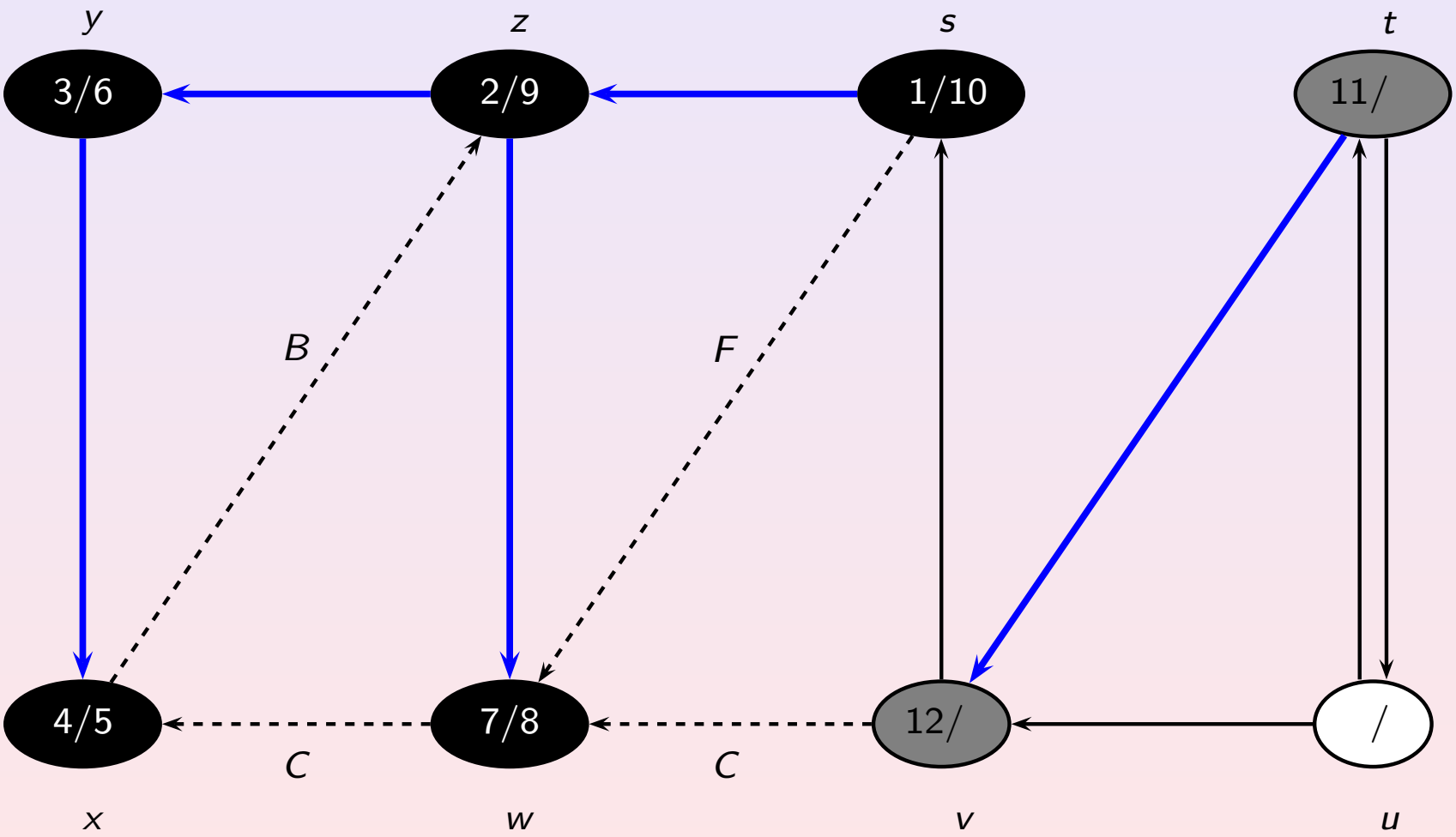
Exemplo de Aplicação do Algoritmo



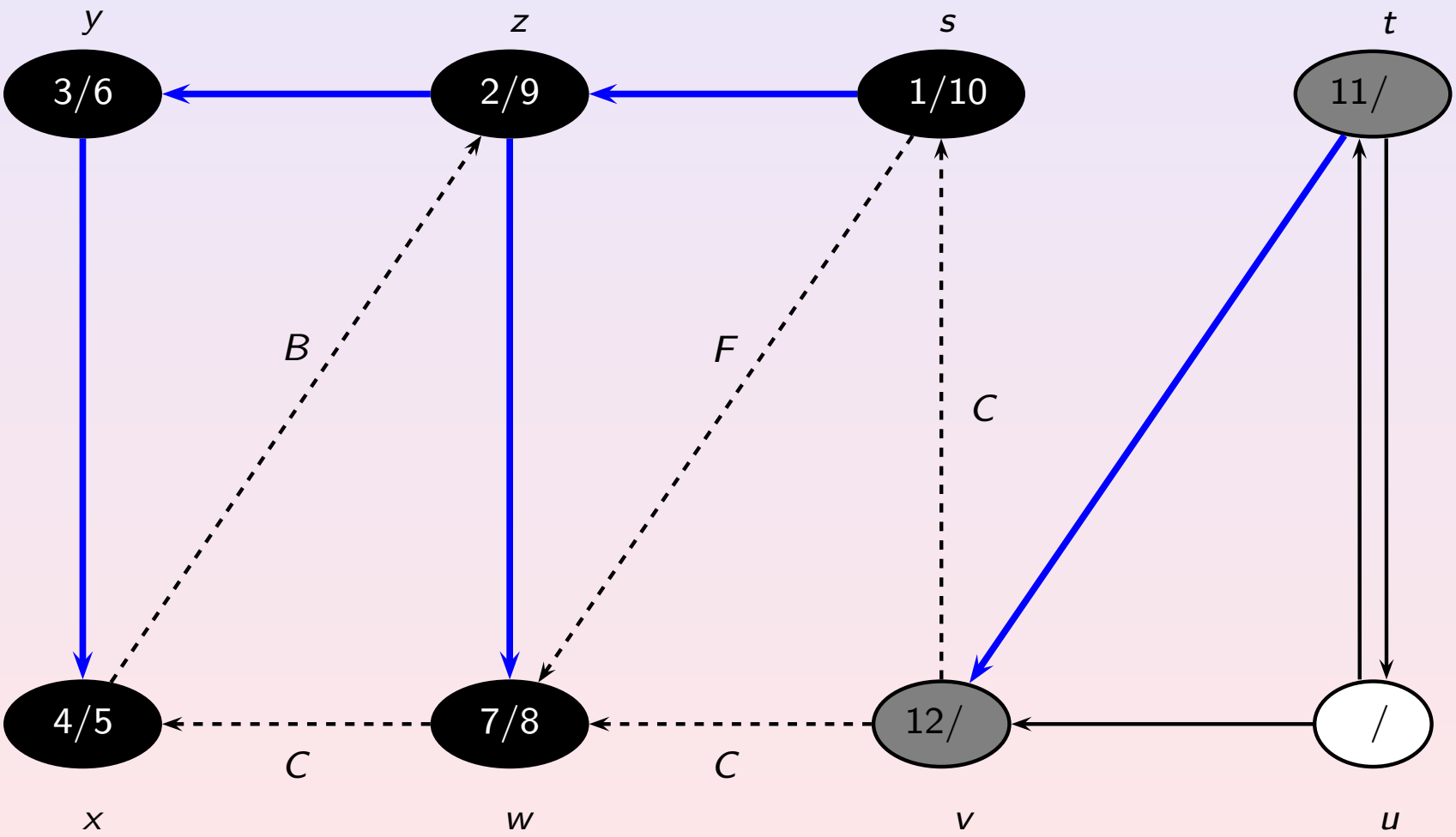
Exemplo de Aplicação do Algoritmo



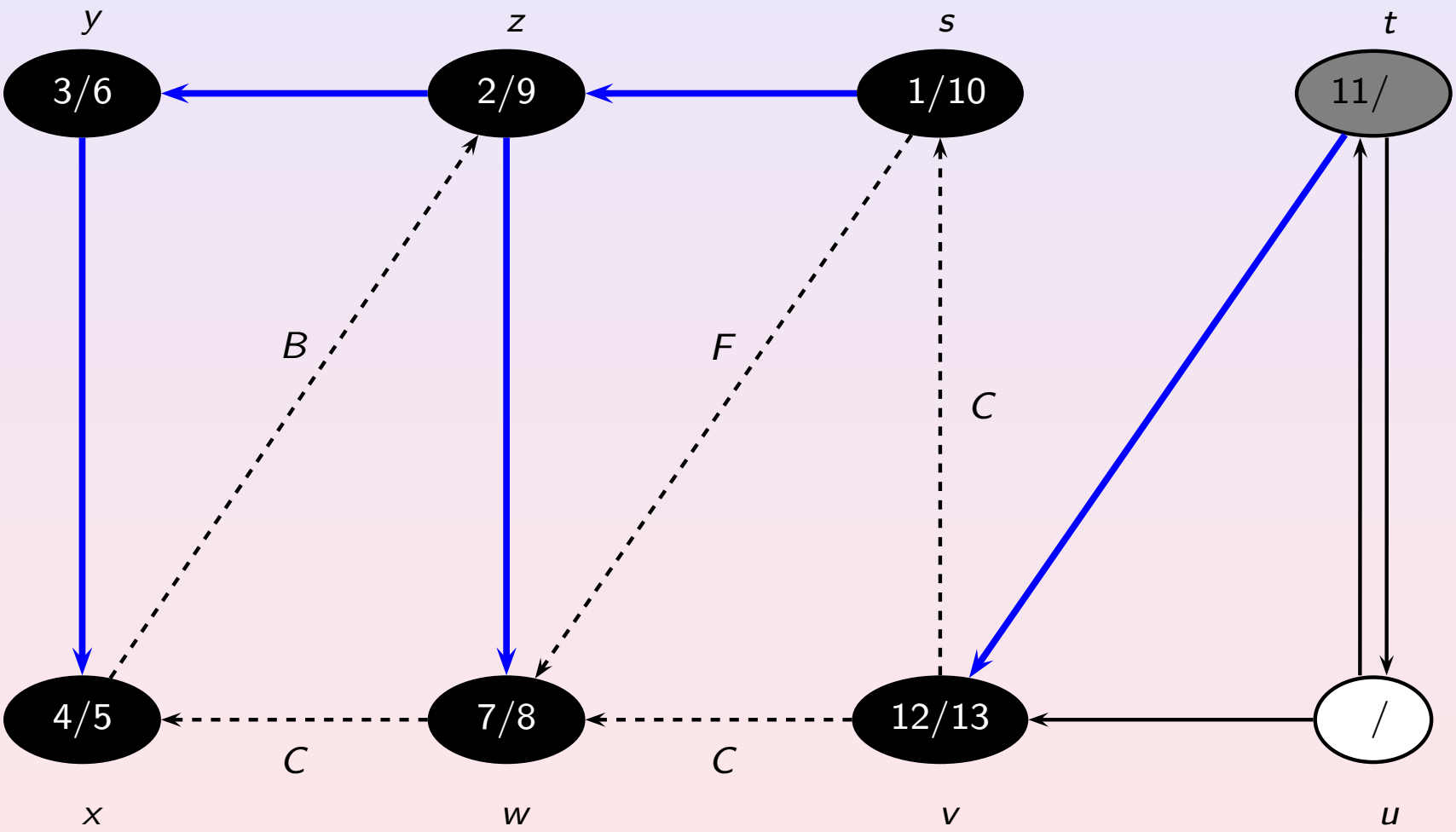
Exemplo de Aplicação do Algoritmo



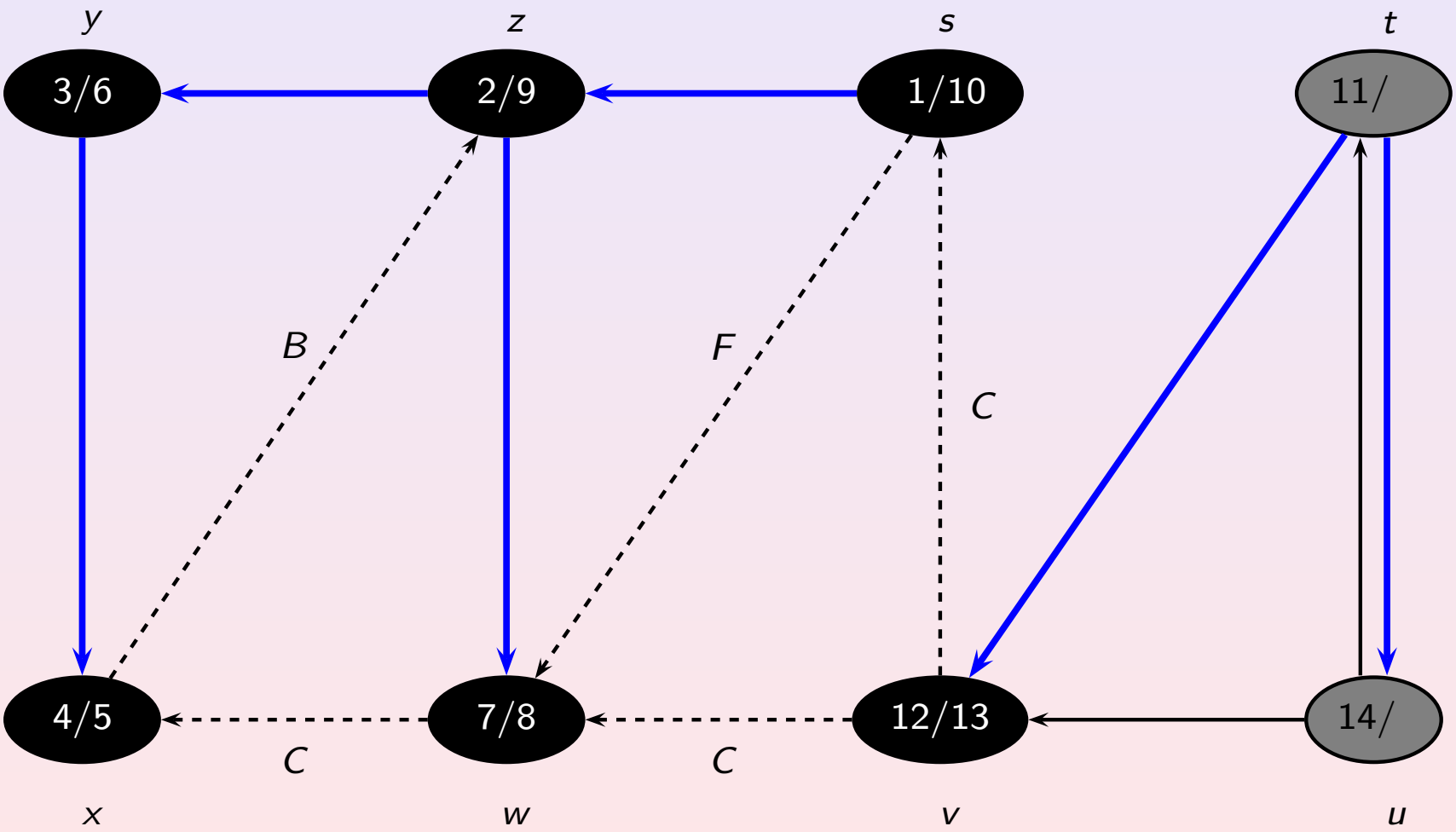
Exemplo de Aplicação do Algoritmo



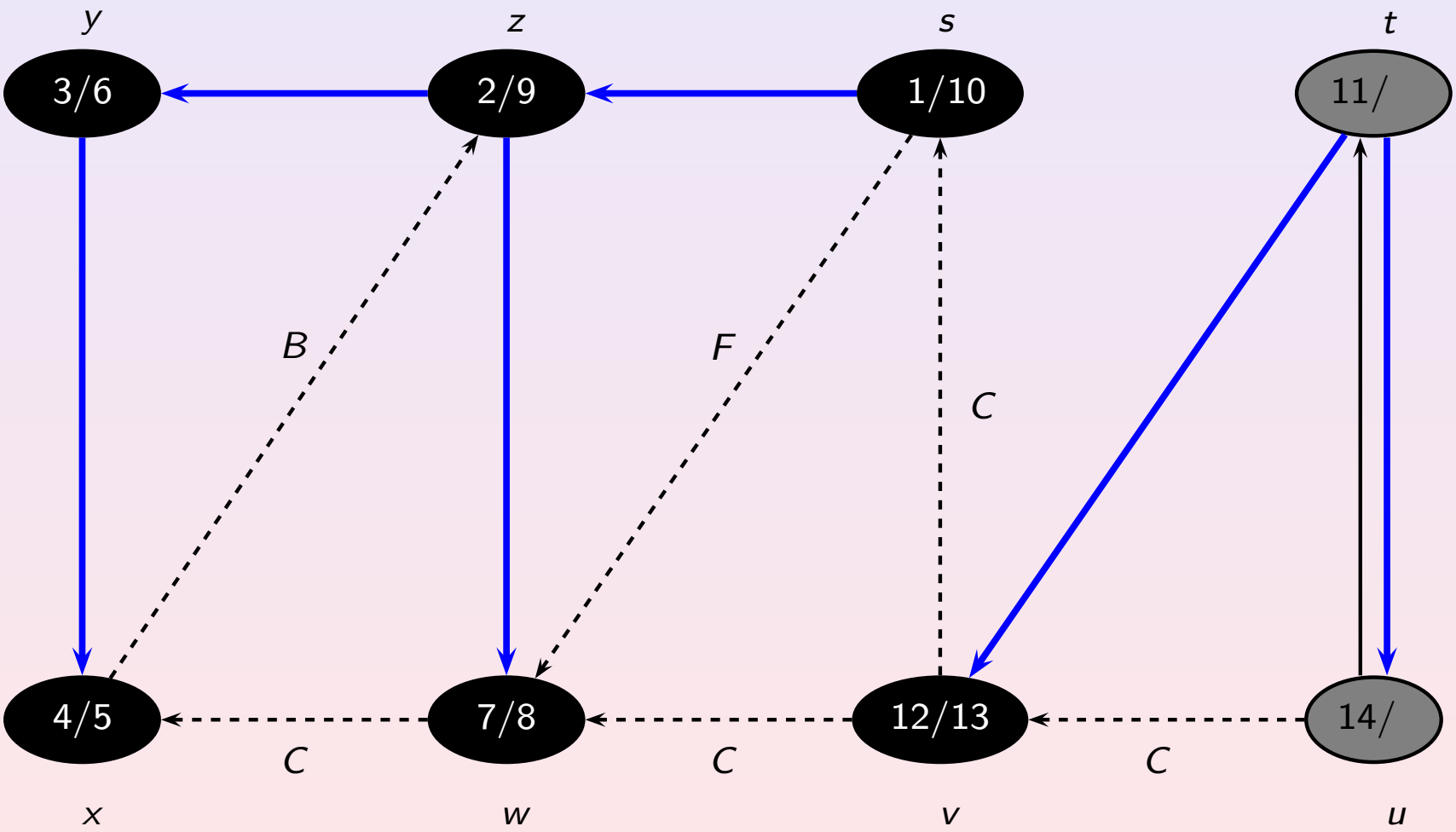
Exemplo de Aplicação do Algoritmo



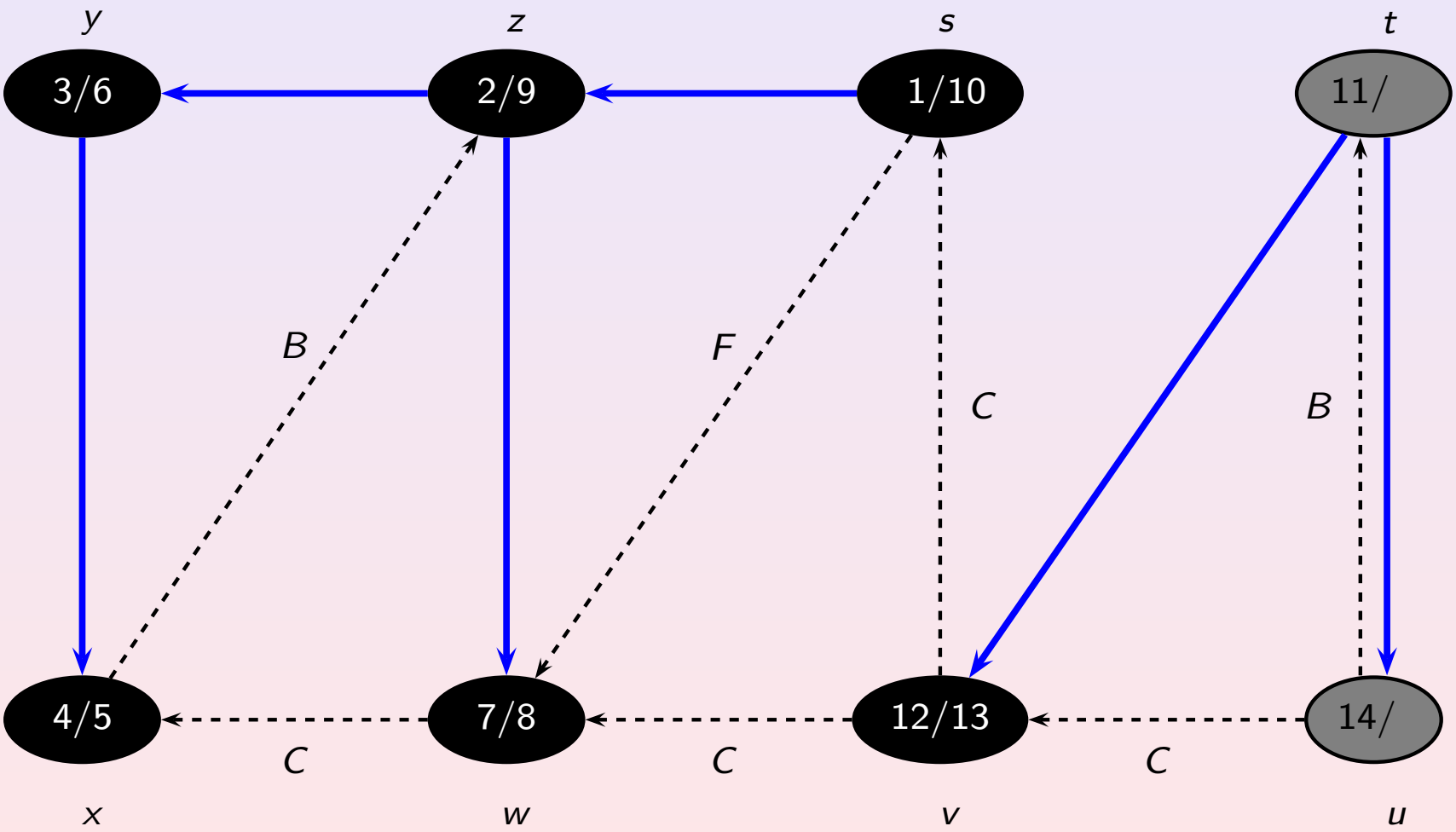
Exemplo de Aplicação do Algoritmo



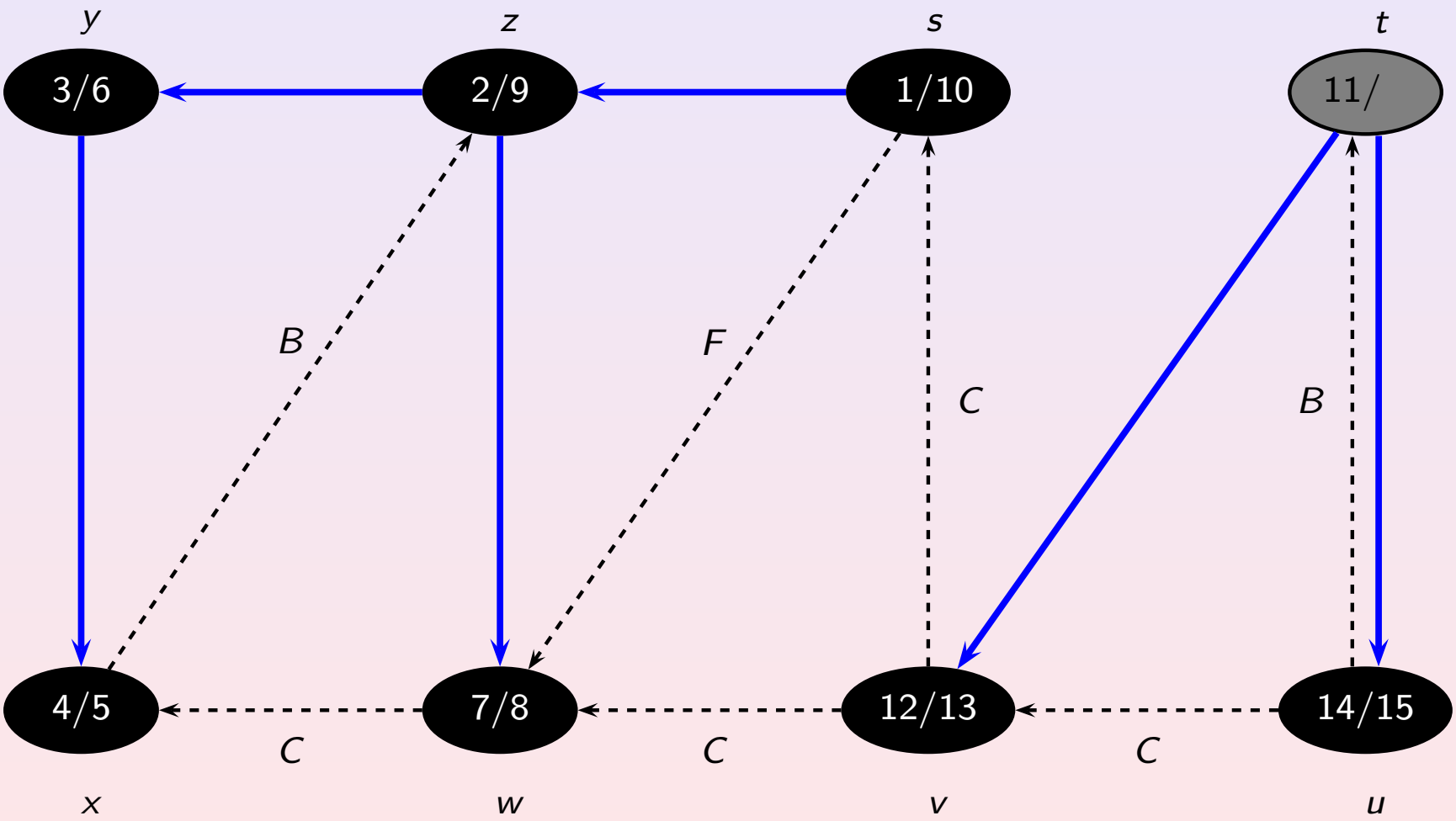
Exemplo de Aplicação do Algoritmo



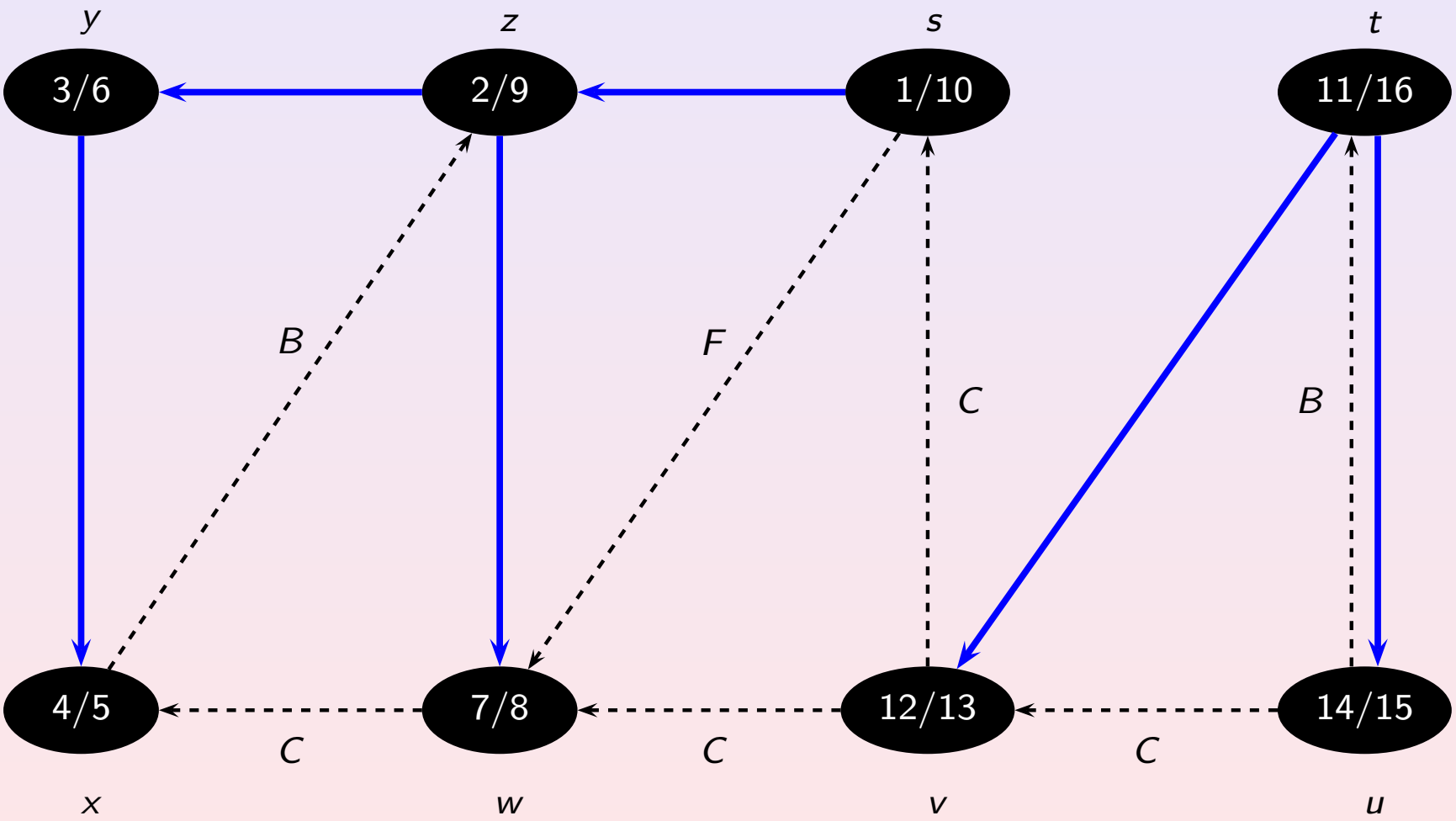
Exemplo de Aplicação do Algoritmo



Exemplo de Aplicação do Algoritmo



Exemplo de Aplicação do Algoritmo



Entendendo as cores e rótulos

- Podemos observar que: para todo $v \in V(G) \rightarrow d[v] < f[v]$
- As cores existem nos vértices no seguintes instantes:
  - Branco: Antes de $d[v]$
  - Cinza: No período $[d[v], f[v])$
  - Preto: Em $f[v]$ adiante
- Para os rótulos dos arcos
  - Sem rótulo: Aresta da árvore de busca em profundidade
  - R: Aresta de retorno (liga um vértice a seu ancestral)
  - F: Aresta de avanço (liga um vértice a seu descendente)
  - C: Aresta de cruzamento

Algoritmo de Busca em Profundidade: Iniciação

Algoritmo $\mathrm{DFS(G)}$
  para $v \in V[G]$ **faça**
    $cor[v] \leftarrow BRANCO$
    $\pi[v] \leftarrow nulo$
  $tempo \leftarrow 0$
  para $v \in V[G]$ **faça**
    se $cor[v] = BRANCO$ **então**
      $VisitaDFS(v)$

Algoritmo de Busca em Profundidade: Visita

Algoritmo VisitaDFS(v)

$cor[v] \leftarrow CINZA$
$tempo \leftarrow tempo + 1$
$d[v] \leftarrow tempo$
para $u \in Adj[v]$ **faça**
  se $cor[u] = BRANCO$ **então**
    $\pi[u] = v$
    $VisitaDFS(u)$
$cor[v] \leftarrow PRETO$
$tempo \leftarrow tempo + 1$
$f[v] \leftarrow tempo$

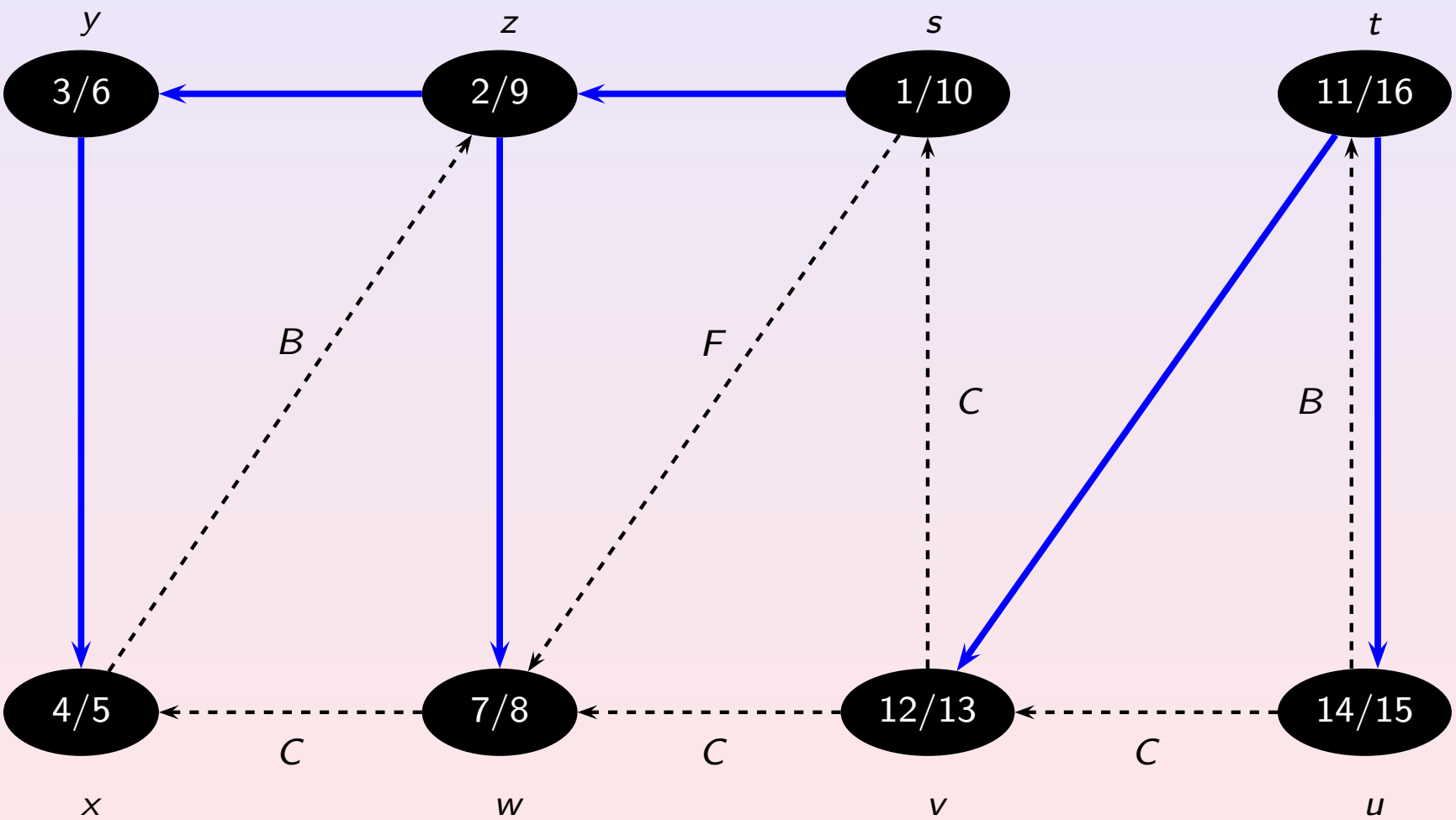
Complexidade do Algoritmo

- O tempo da iniciação: $O(V)+$ chamadas a *VisitaDFS*
- No *VisitaDFS* para o vértice v, o laço é executado $Adj[v]$
- Cada vértice é chamado uma vez no *VisitaDFS* (todos que são brancos são chamados)
- $\sum_{v \in V} Adj[v] = \Theta(E)$.
- O algoritmo é $O(V + E)$.

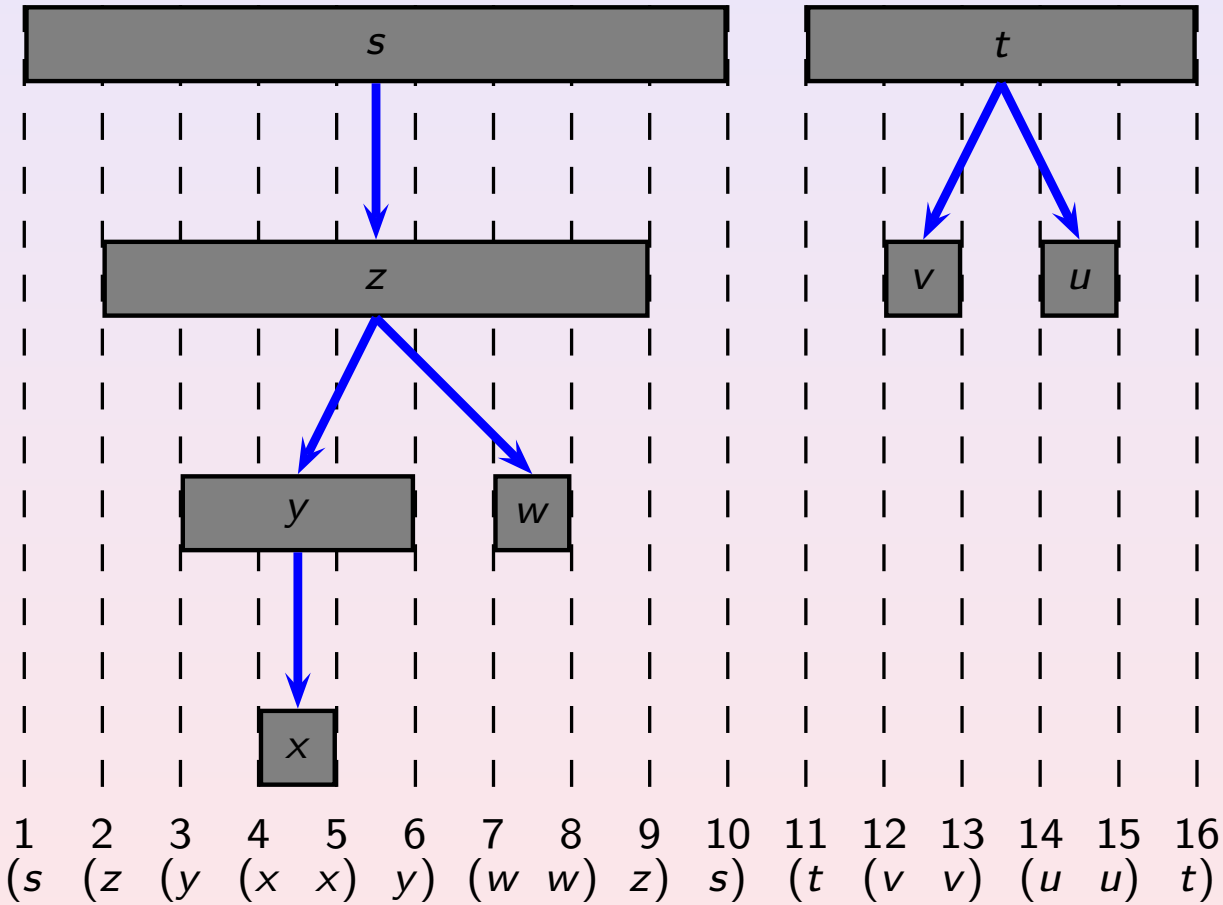
Estrutura de Parênteses

- Os rótulos de tempo: $d[v]$ e $f[v]$ definem o início e final de visita a um vértice.
- Na árvore de busca em profundiade, todos os descendentes do vértice são visitados dentro do período de tempo do vértice.
- Poderíamos interpretar o $d[v]$ como (v, e o $f[v]$ como v), ou seja, o que acontece entre o início e o final ocorre dentro dos parênteses de v.
- Esta estrutura de tempo define a *cara* da árvore de busca.

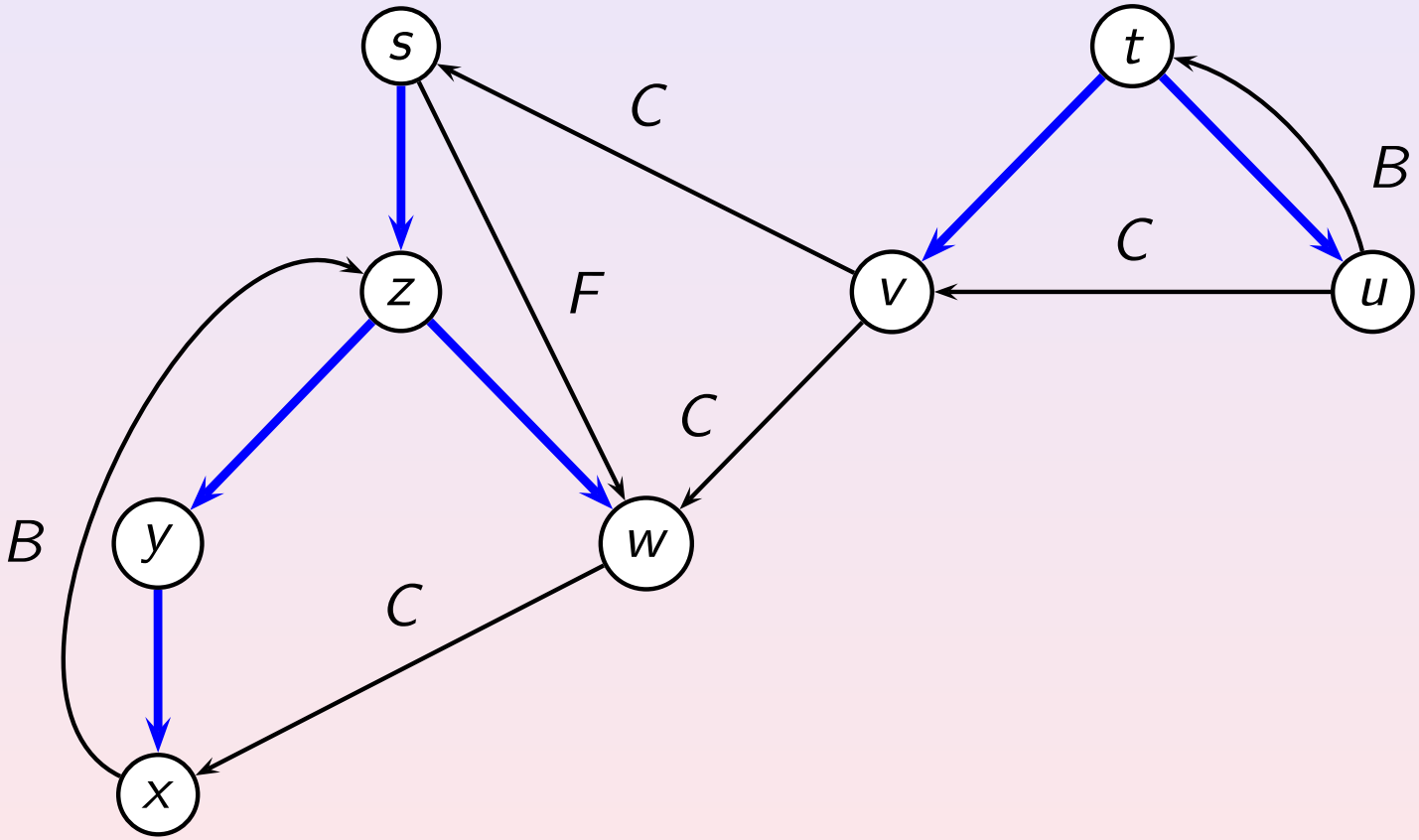
Árvore de Busca em Profundidade



Estrutura de Parênteses: Árvore de Busca em Profundidade



Floresta de Busca em Profundidade e classificação de arcos



Teorema dos Parênteses

Theorem

Em uma busca em profundidade sore 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.*
- $[d[u], f[u]]$ *está contido em* $[d[v], f[v]]$ *e u é descendente de v na árvore de Busca em Profundidade.*
- $[d[v], v[v]]$ *está contido em* $[d[u], f[u]]$ *e v é descendente de u na árvore de Busca em Profundidade.*

Corolário: Intervalos encaixantes para descendentes

Corollary

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

De forma equivalente, v é um descendente próprio de u se e somente se $[d[v], f[v]]$ está contido em $[d[u], f[u]]$.

Teorema: Teorema do Caminho Branco

Theorem

Em uma Floresta de Busca em Profundidade, 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.

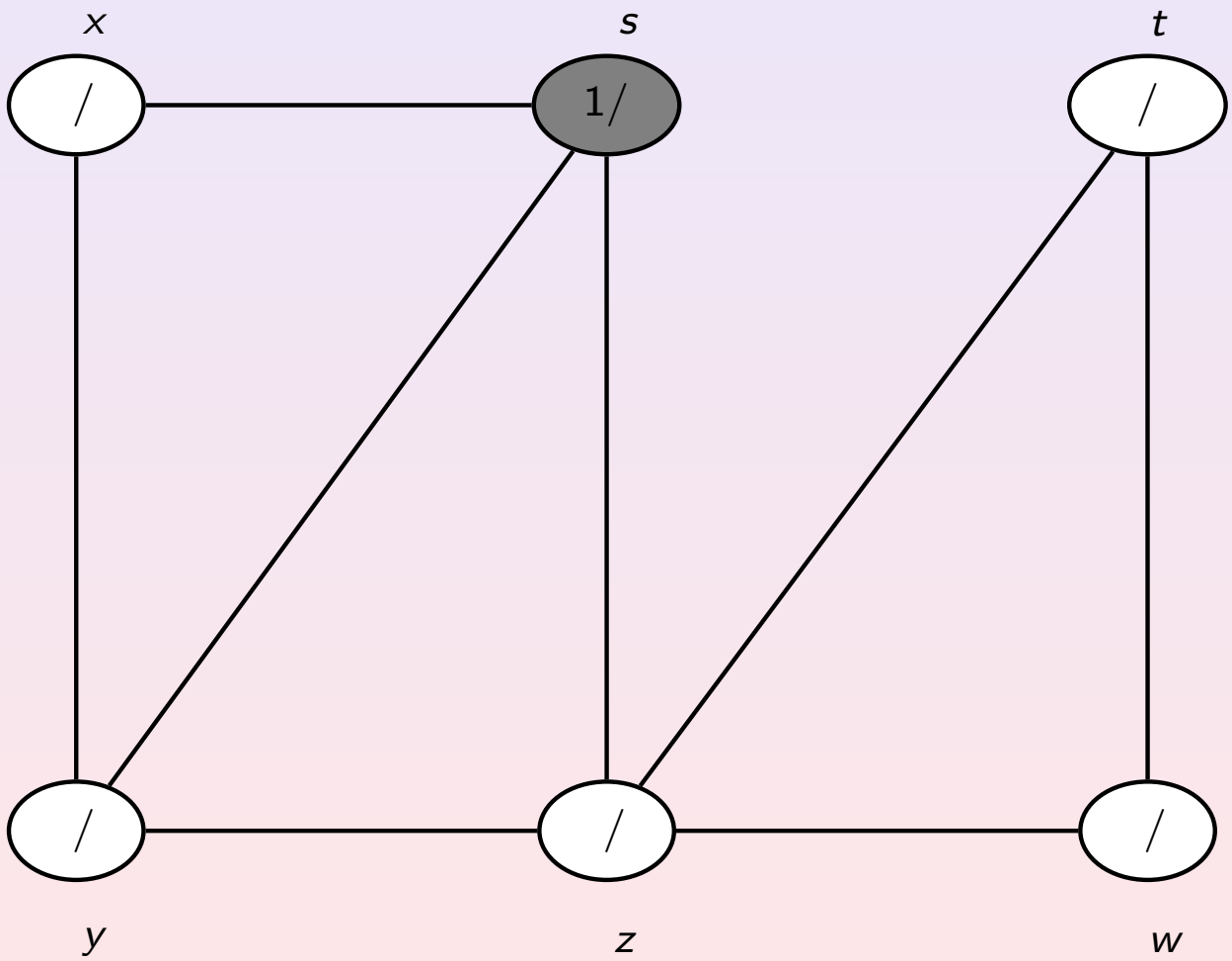
Grafos não-orientado

- Em grafos não orientados $\{u, v\}$ e $\{v, u\}$ indicam a mesma aresta.
- Existem apenas dois tipos de arestas:

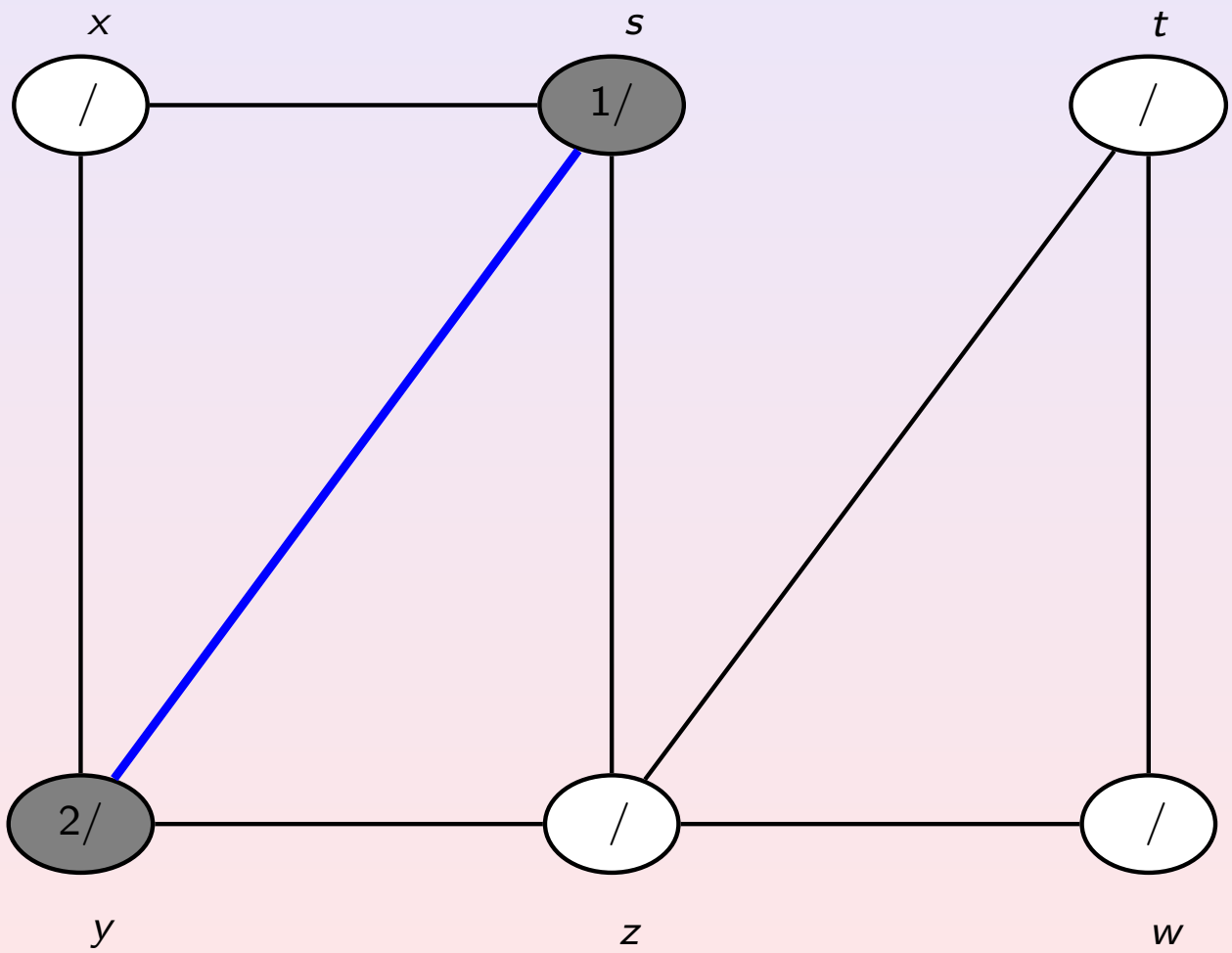
Theorem

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

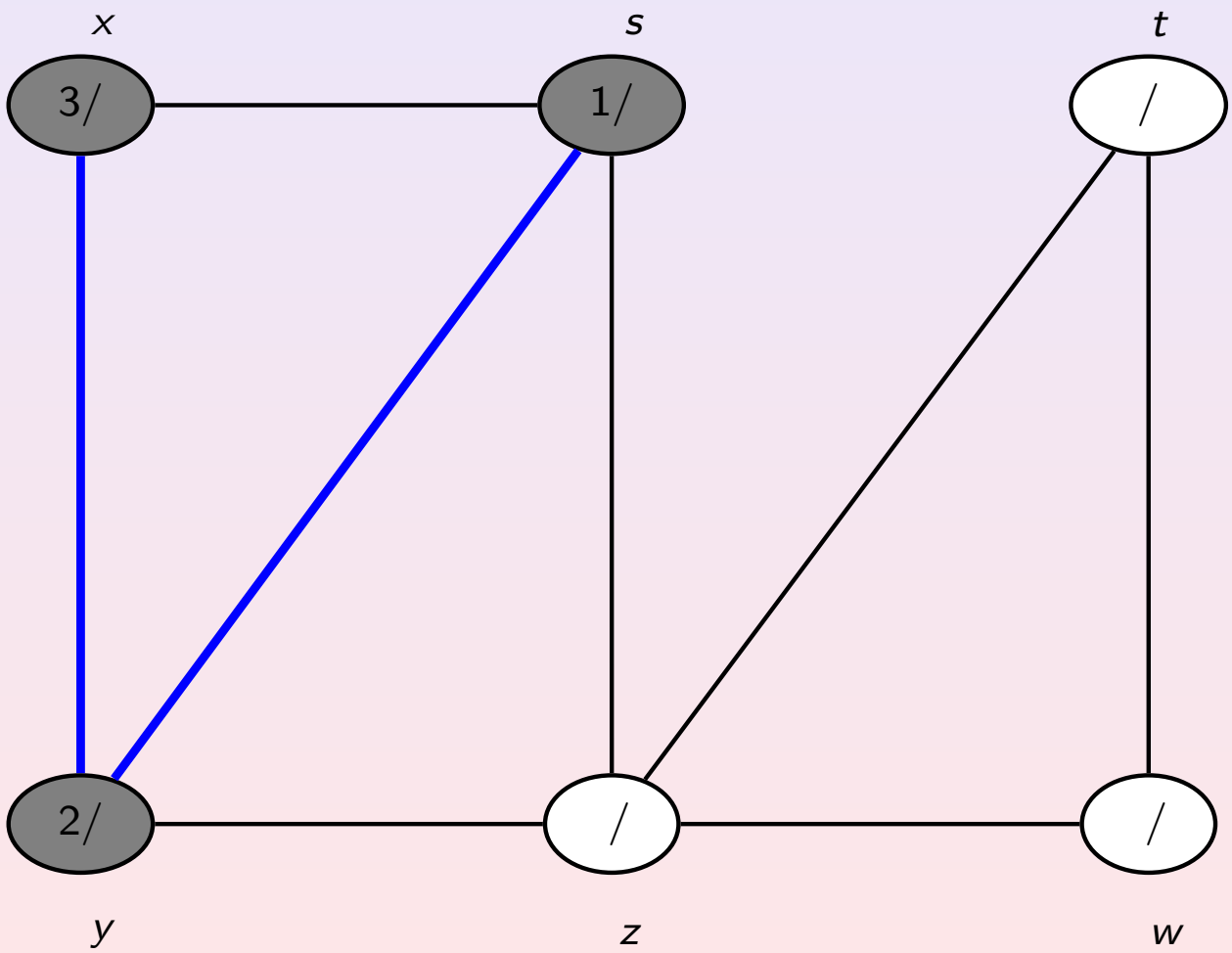
Exemplo de Busca em Profundidade: Grafo não orientado



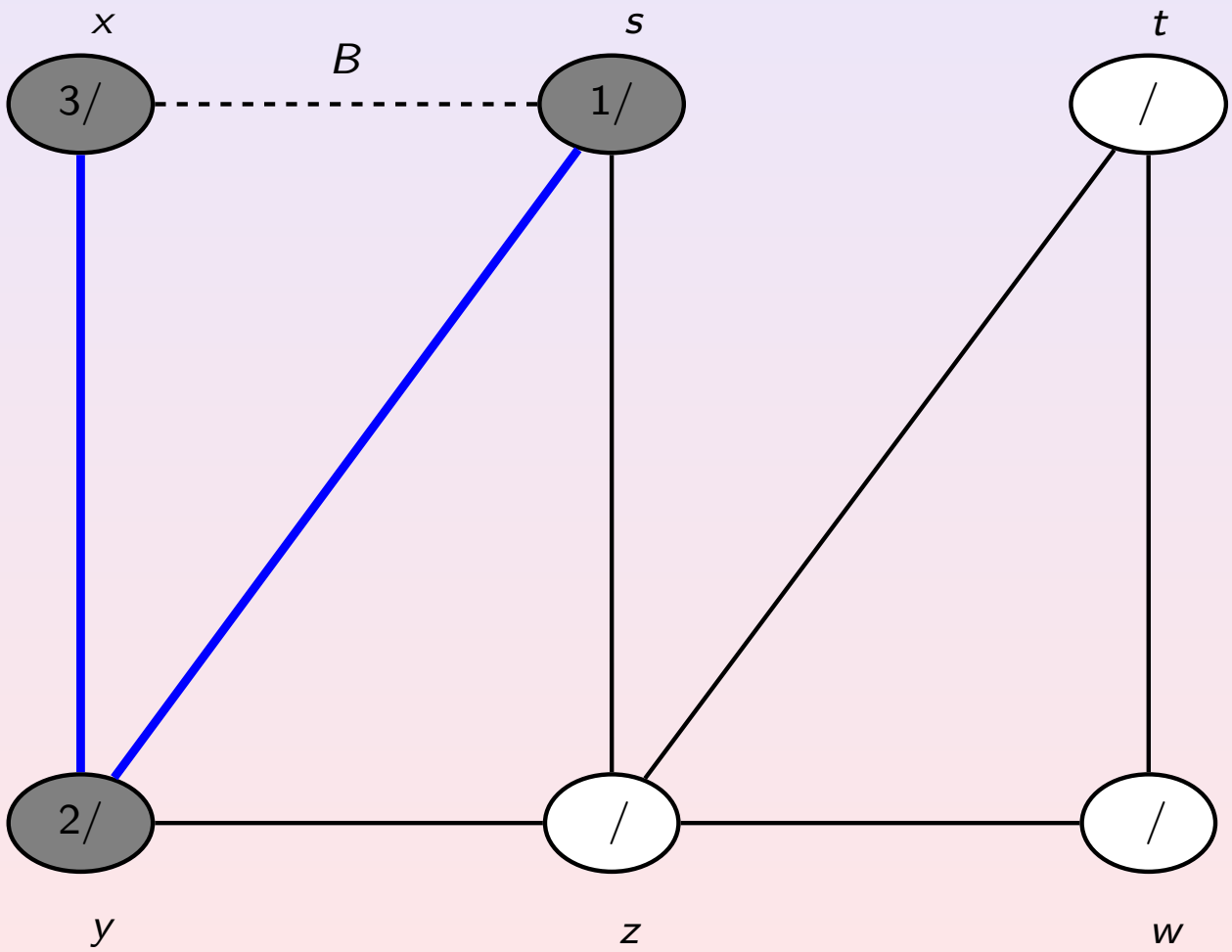
Exemplo de Busca em Profundidade: Grafo não orientado



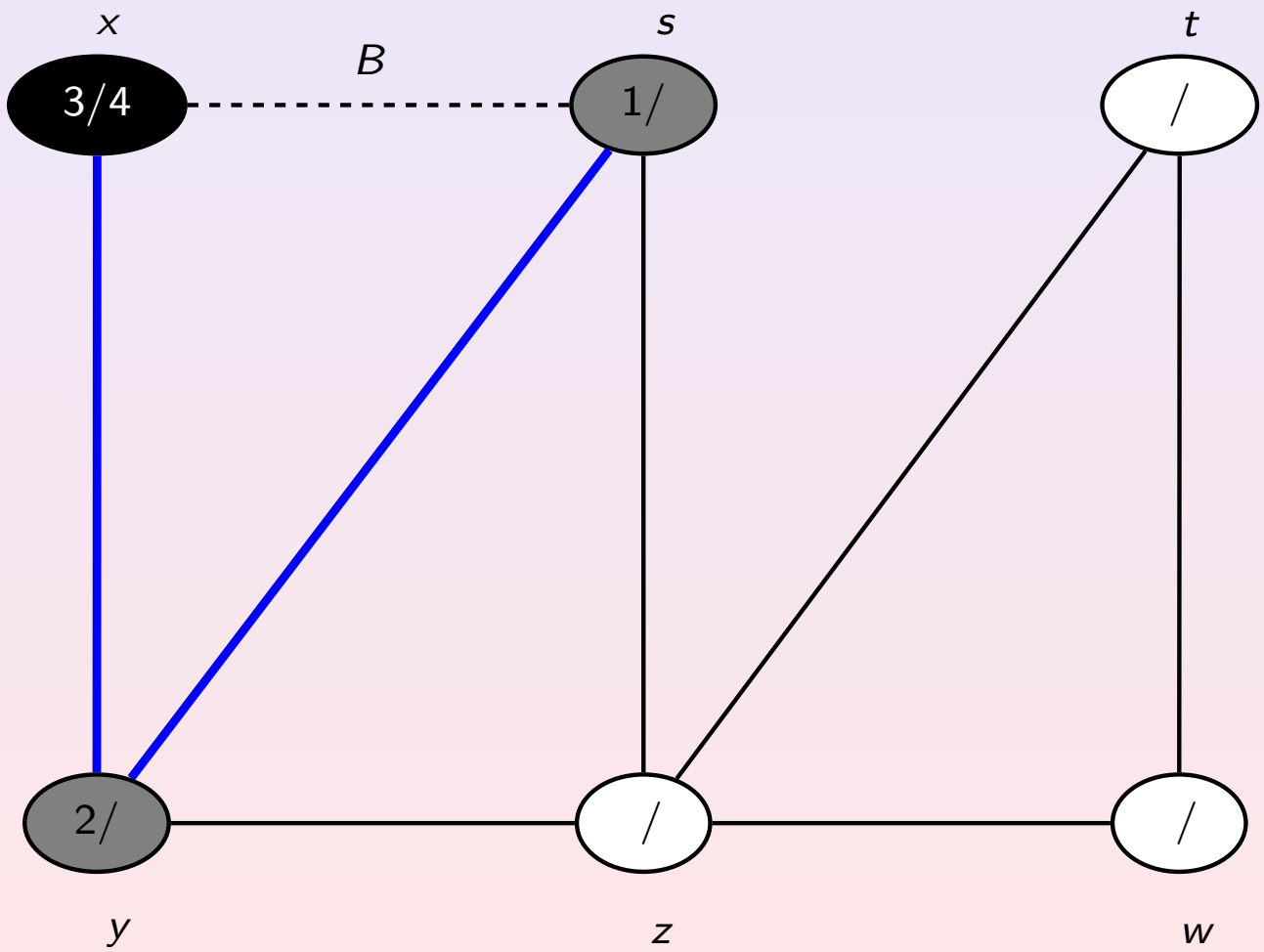
Exemplo de Busca em Profundidade: Grafo não orientado



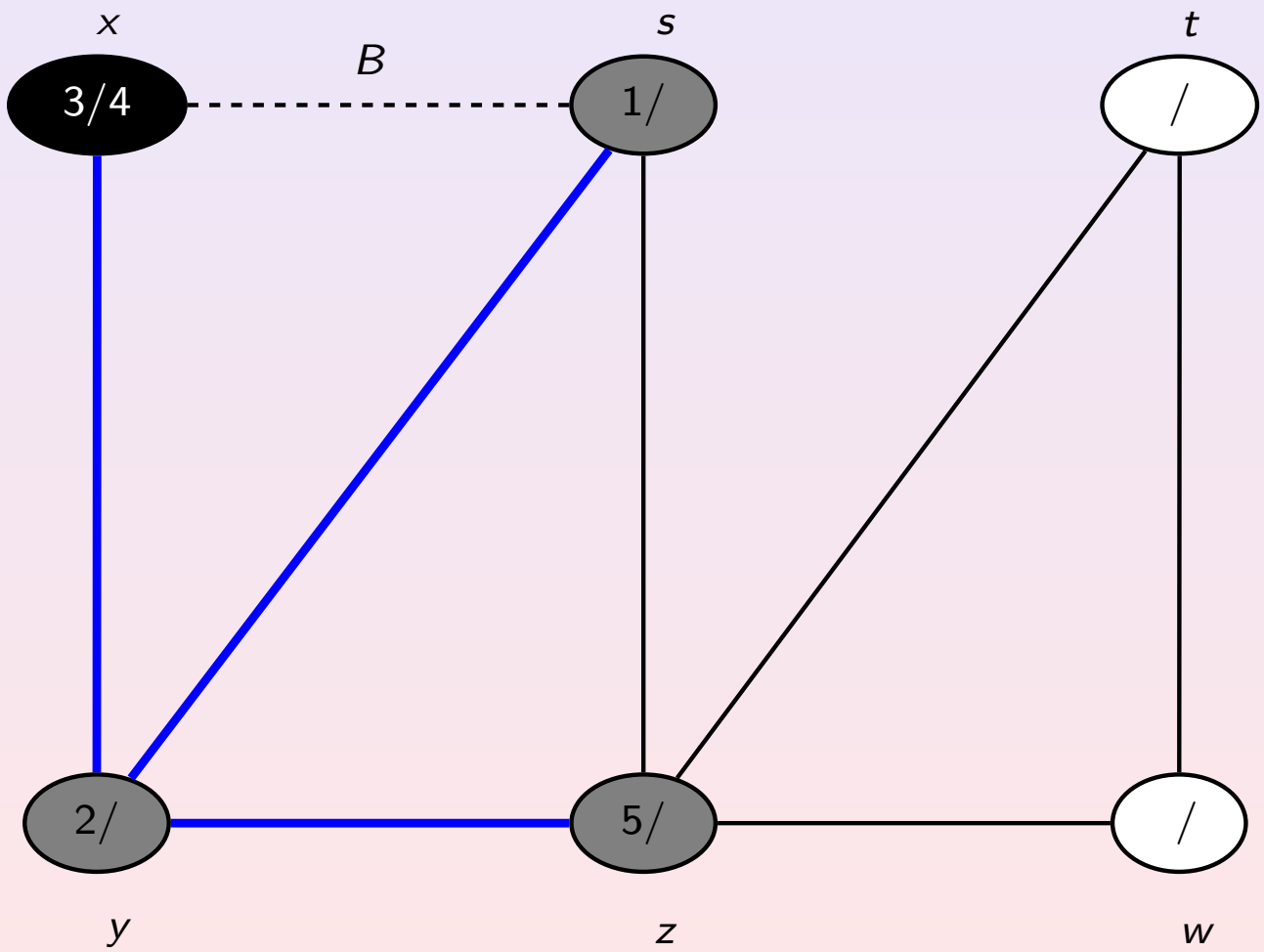
Exemplo de Busca em Profundidade: Grafo não orientado



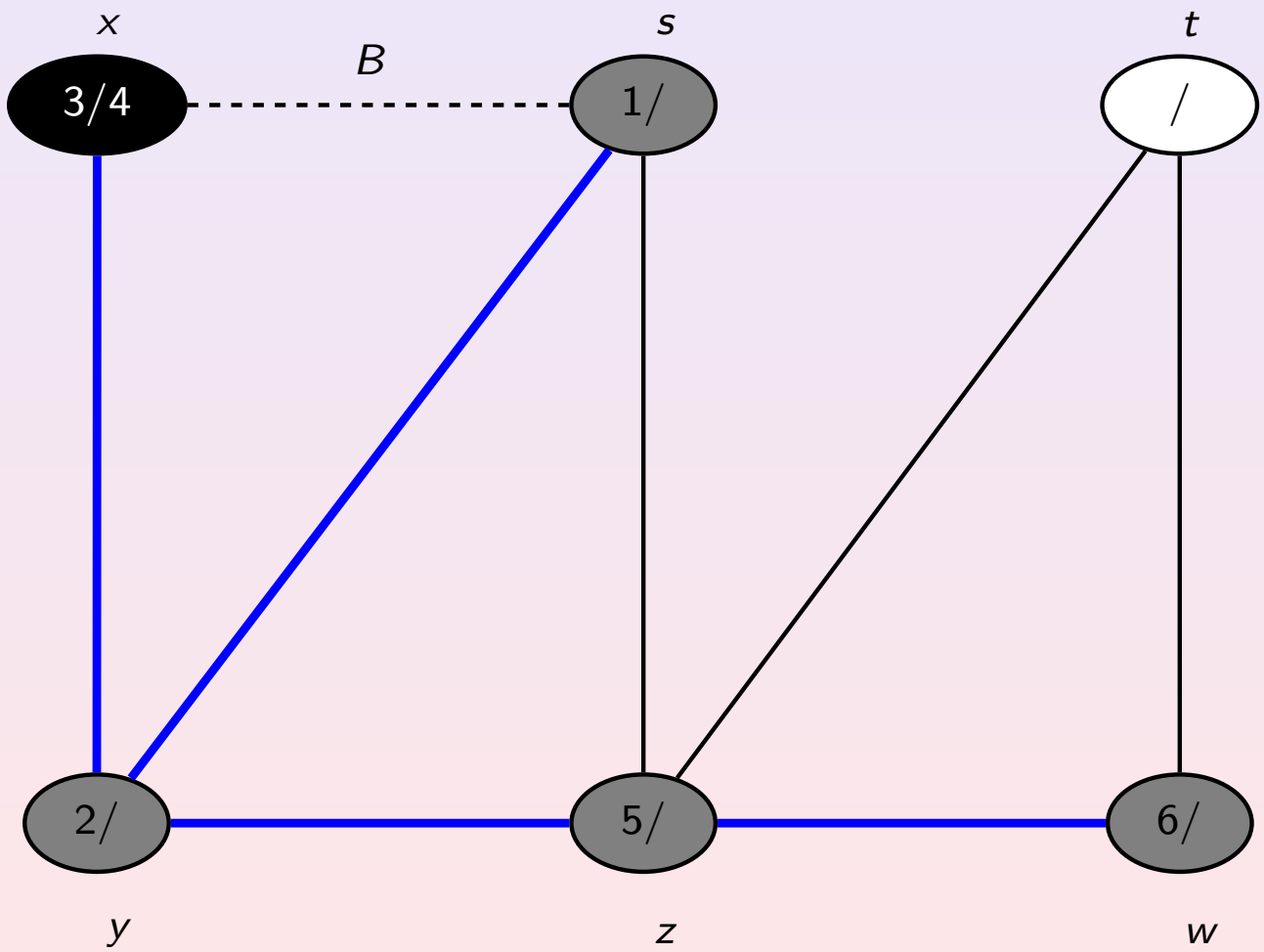
Exemplo de Busca em Profundidade: Grafo não orientado



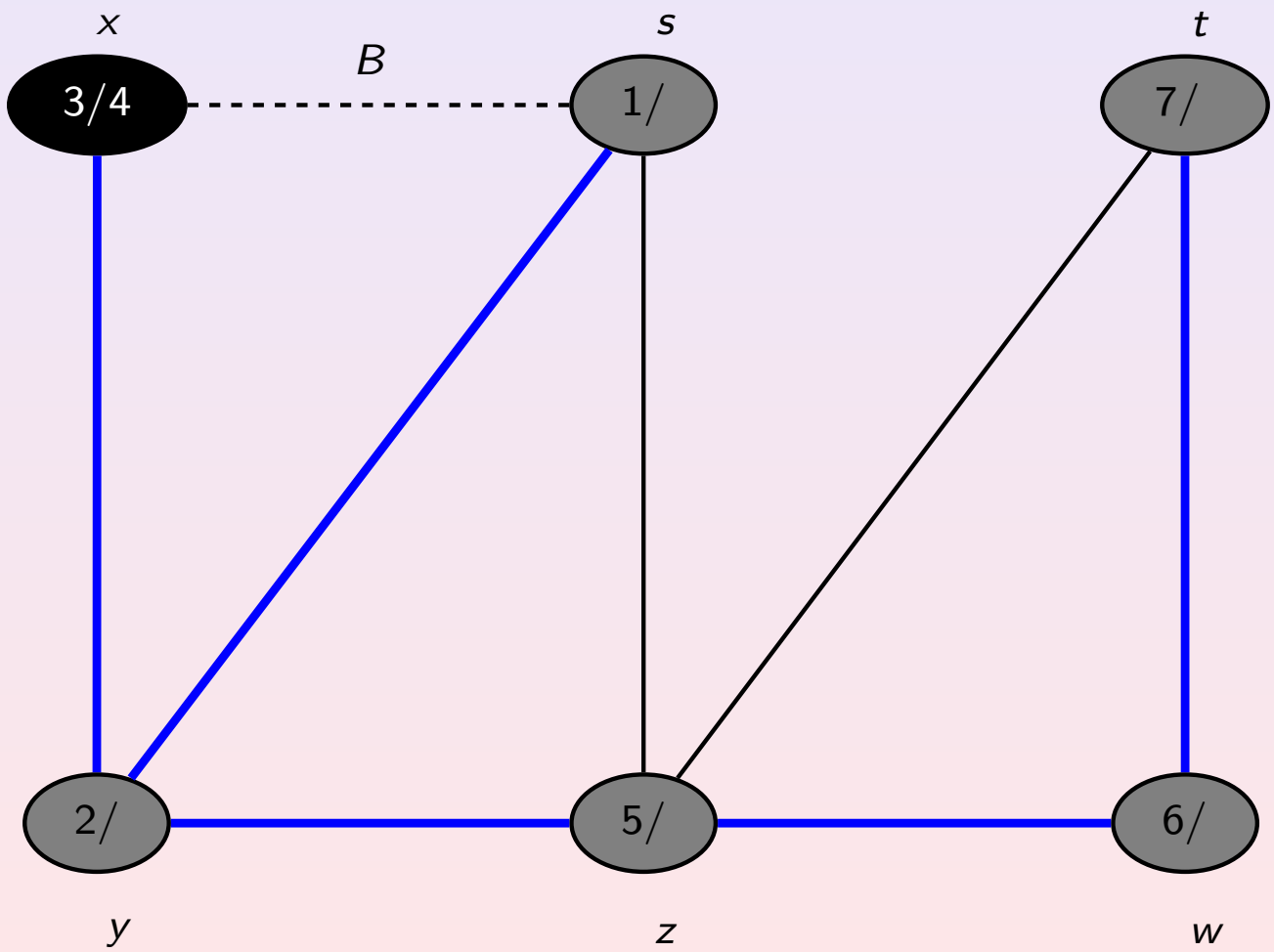
Exemplo de Busca em Profundidade: Grafo não orientado



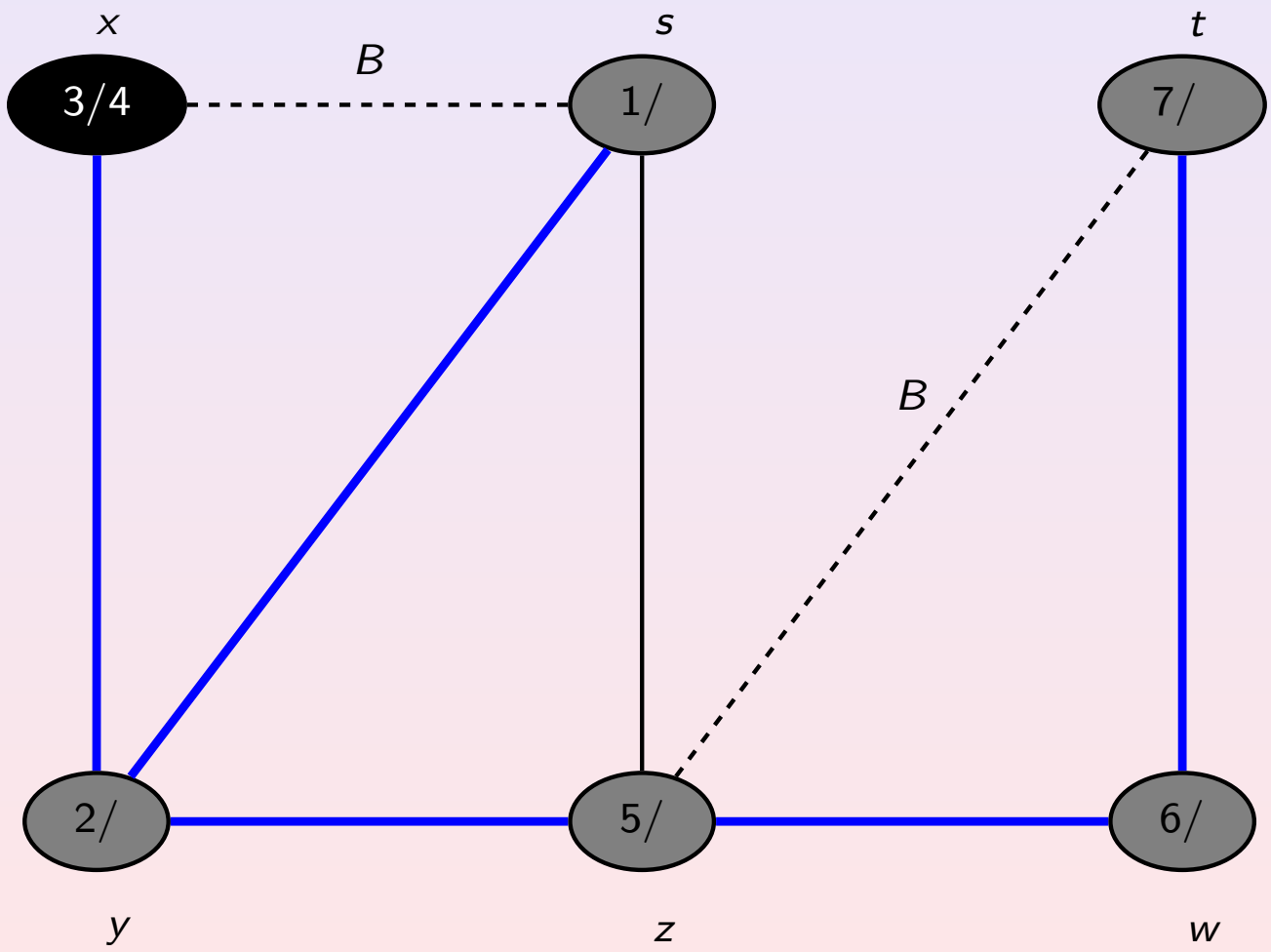
Exemplo de Busca em Profundidade: Grafo não orientado



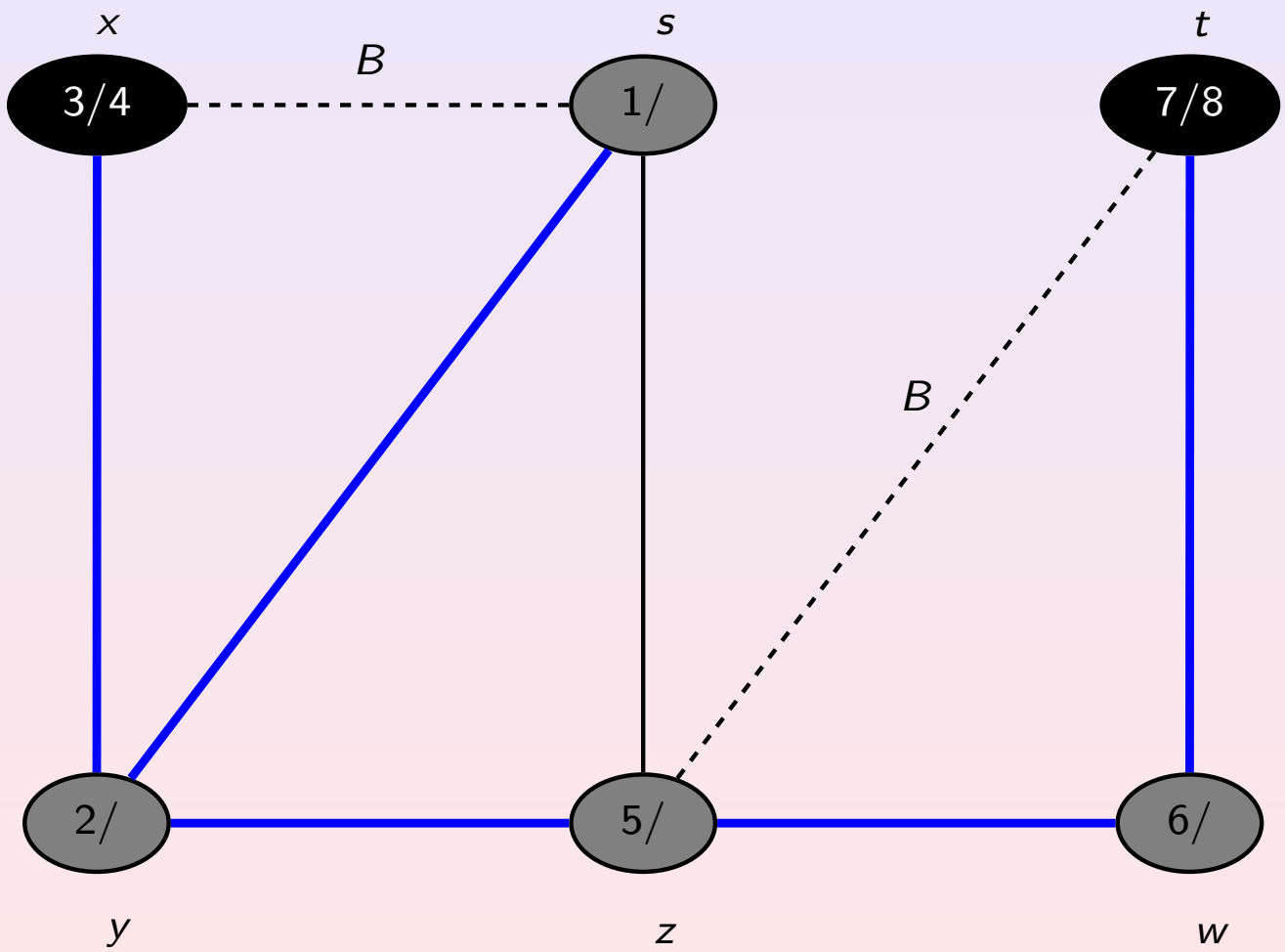
Exemplo de Busca em Profundidade: Grafo não orientado



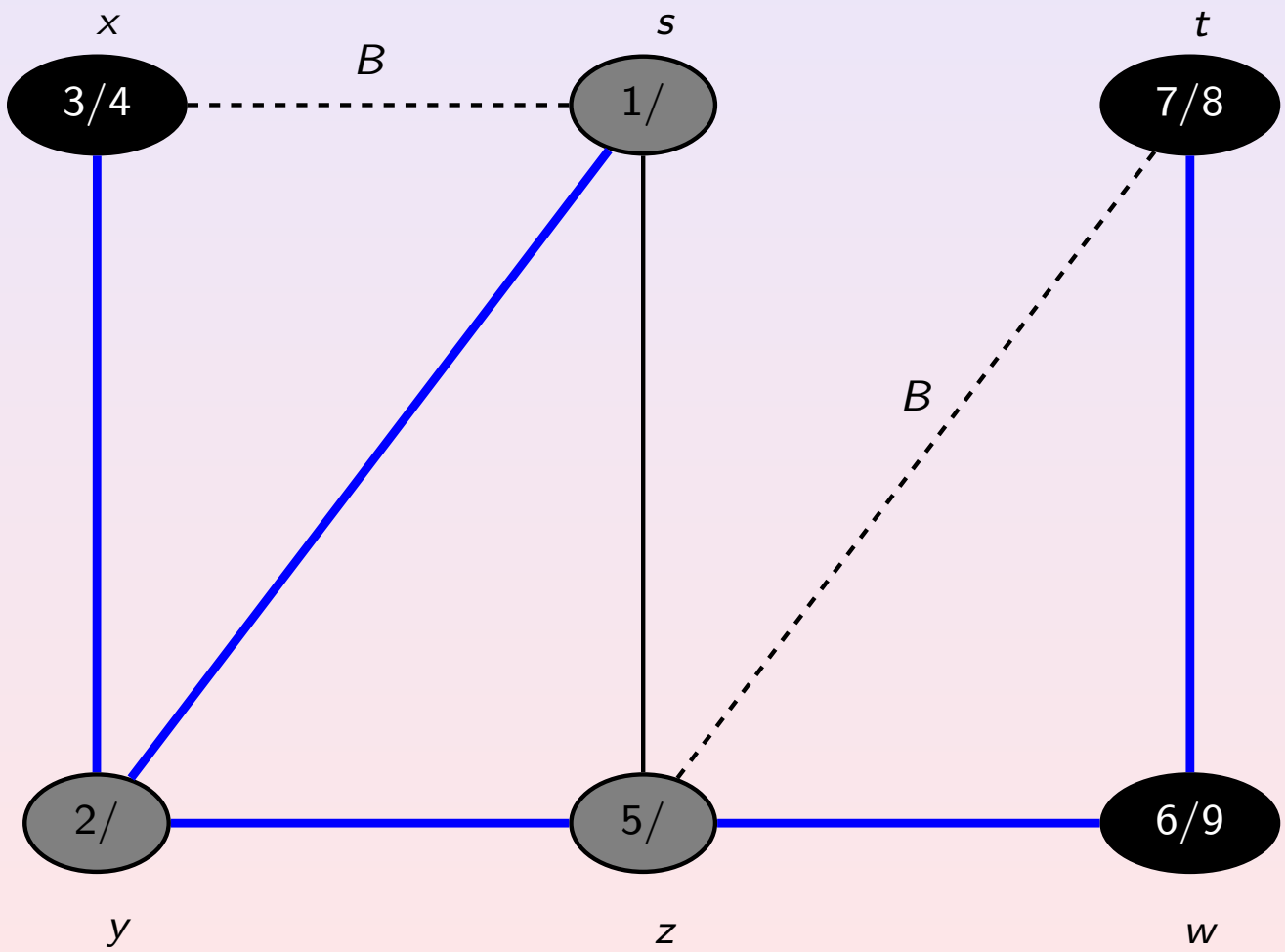
Exemplo de Busca em Profundidade: Grafo não orientado



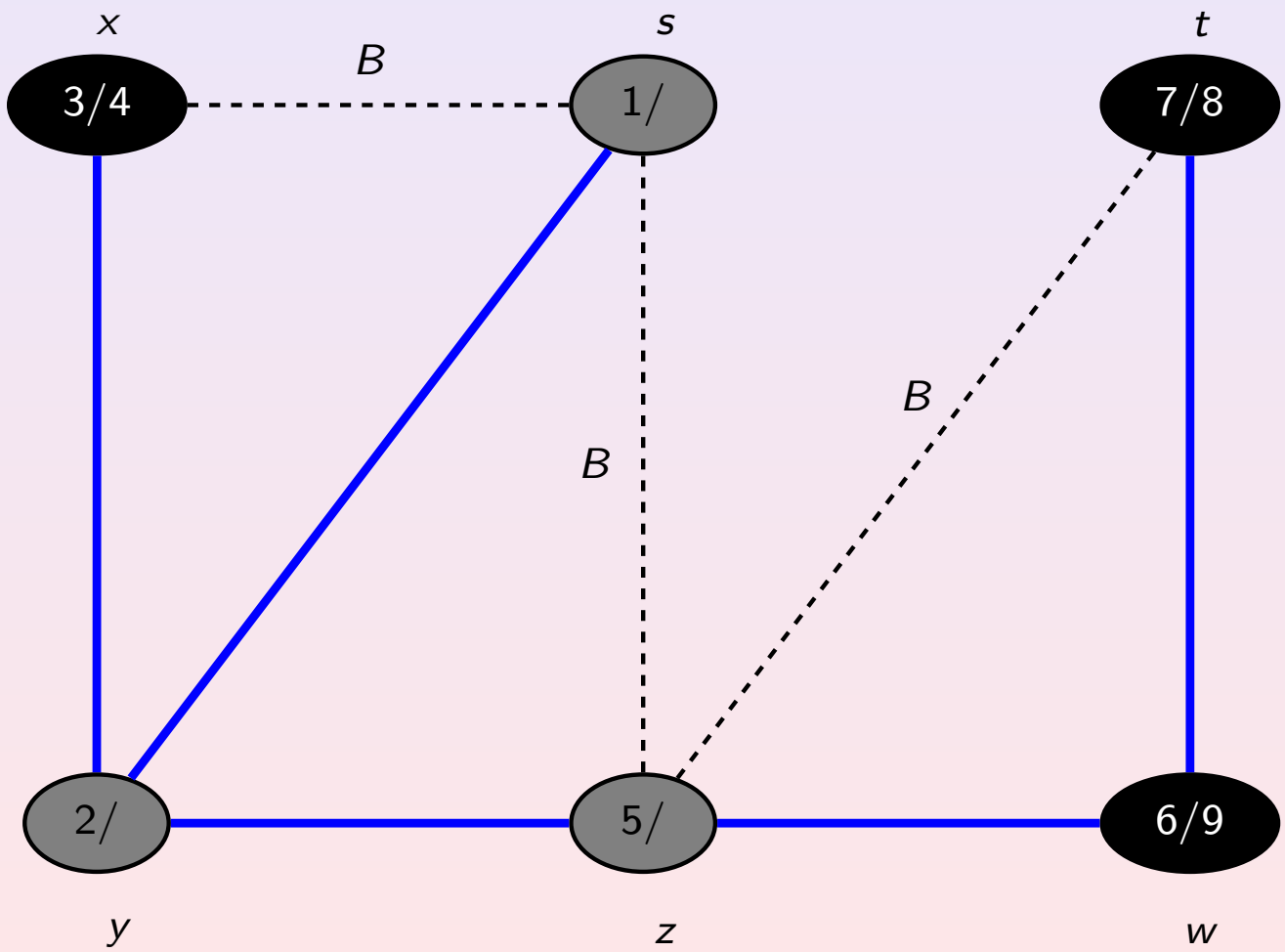
Exemplo de Busca em Profundidade: Grafo não orientado



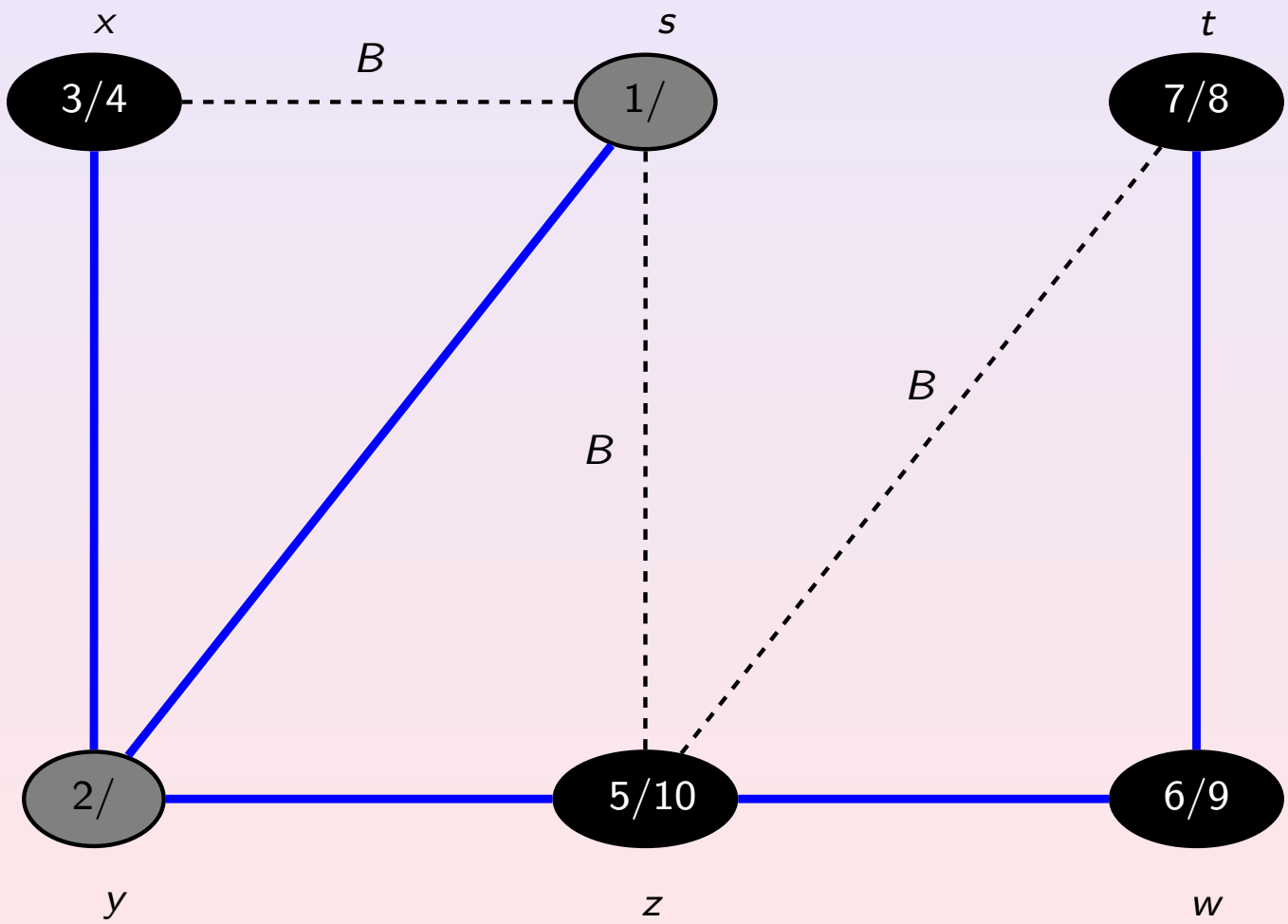
Exemplo de Busca em Profundidade: Grafo não orientado



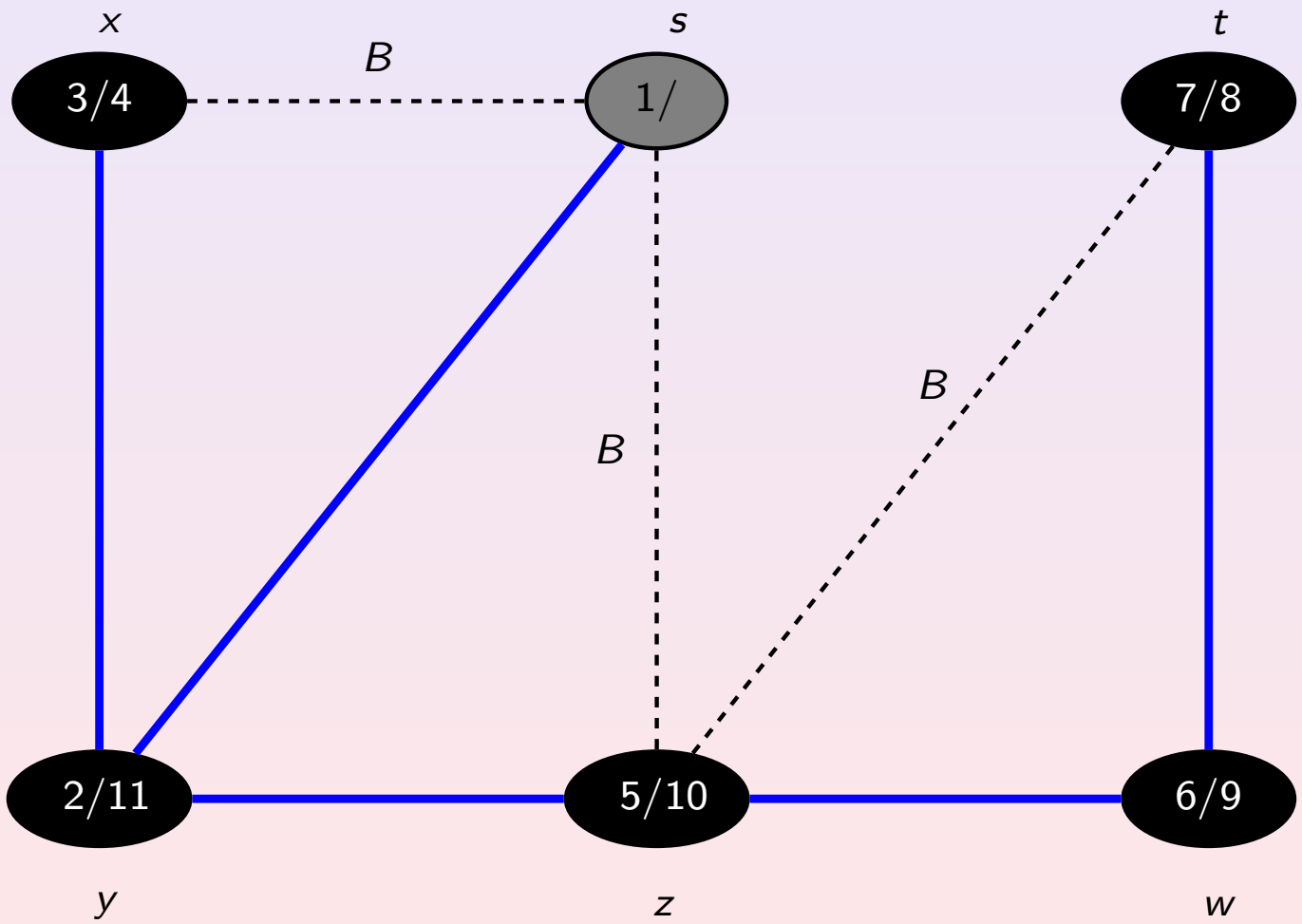
Exemplo de Busca em Profundidade: Grafo não orientado



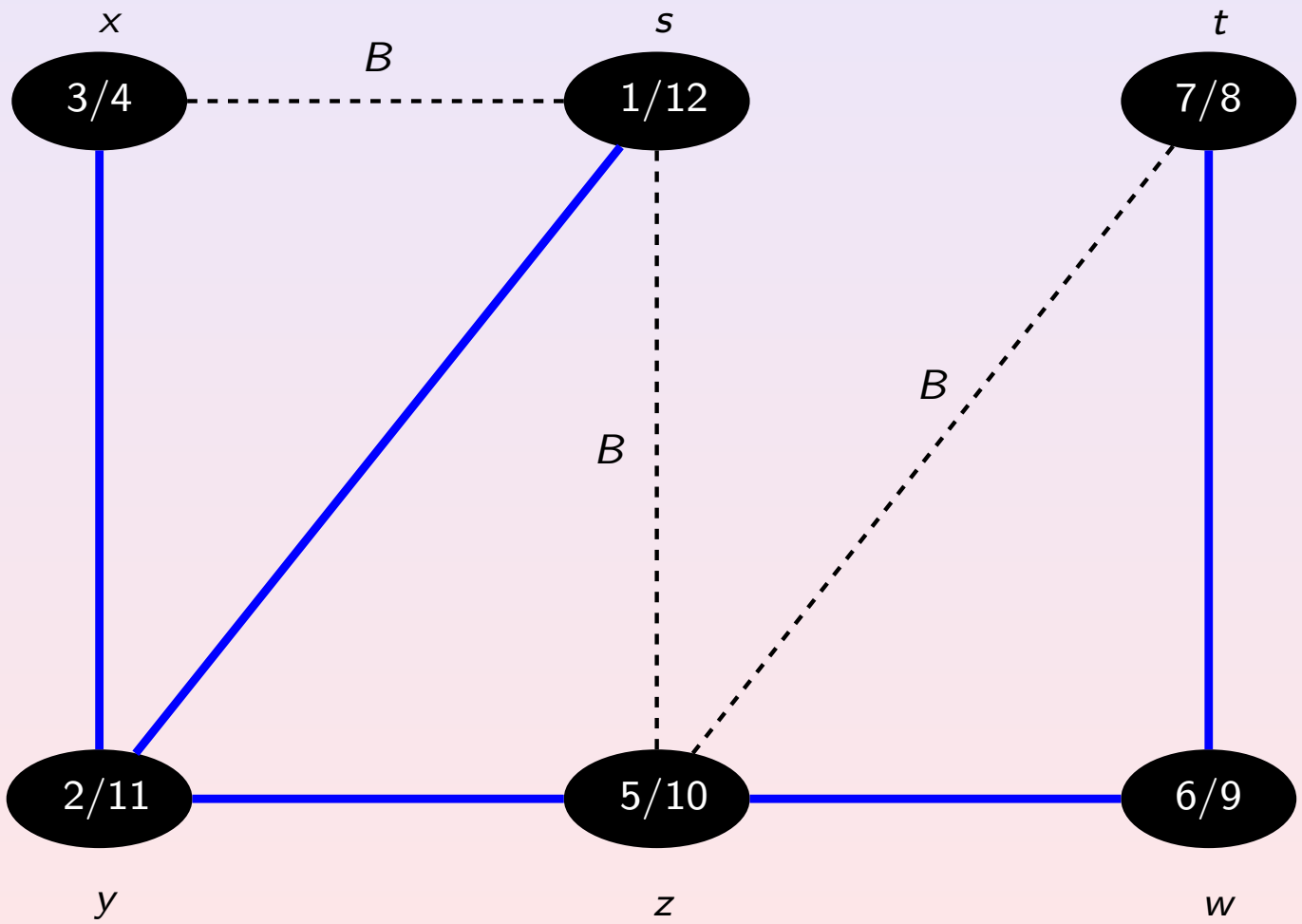
Exemplo de Busca em Profundidade: Grafo não orientado



Exemplo de Busca em Profundidade: Grafo não orientado



Exemplo de Busca em Profundidade: Grafo não orientado



Exemplo: URI - 1706 - Pontes Mágicas

Como acontece em todo réveillon, o reino Doce organizará uma incrível festa da virada. A princesa Bubblegum (PB) pediu para construir diversas torres musicais, com pontes conectando essas. Uma torre musical é uma nova ideia de PB e funciona assim: cada torre pode tocar duas notas musicais A e B. Elas começam com uma nota aleatória (A ou B) e o objetivo é fazer todas as torres tocarem a nota A. Aí que as pontes entram, se você tocar a ponte com uma varinha doce mágica, as notas das duas torres magicais conectadas por essa ponte mudarão.

Agora PB não sabe se é possível organizar o festival cumprindo tal objetivo. Ela deu aos heróis Finn e Jake o mapa com as torres musicais, as conexões das pontes e o som inicial de cada torre e perguntou a eles se seria possível organizar tal festival. Como Finn e Jake não sabem muito sobre ciência, eles pediram para você resolver essa tarefa.

Exemplu: URI - 1706 - Pontes Mágicas - Entrada/Saída

Entrada:

Terão diversos casos testes. Cada caso teste contém dois inteiros: **N** ($1 \leqslant$**N**$\leqslant 1000$) e **M** ($1 \leqslant$**M**$\leqslant 4000$). A próxima linha contém **N** letras, indicando o som inicial da i-ésima torre. As próximas **M** linhas, contém dois inteiros a ($1 \leqslant$**a**) e **b** (**b**$\leqslant$**N** e **a**$\neq$**b**), indicando que existe uma ponte conectando a torre **a** e a torre **b**. A entrada termina com o final do arquivo.

Saída:

Para cada caso você deve imprimir **Y** se for possível o festival acontecer ou **N** caso contrário.

URI - 1706 - Proposta de solução

- Vamos fazer uma busca em profundidade.
- Ao chegar a um nó que não tem mais filhos, na busca, se ele for 'B', ele troca sua nota e a nota de seu pai.
- Ao chegar no raiz da busca, se o raiz não for 'A' então não é possível organizar o festival.
- Um fato é importante: o grafo não é necessariamente conexo.

URI - 1706 - Classe de um vértice

```
class no {
public:
  int id;
  bool note;
  bool visited;
  int parent;
  vector<int> nb;
  no(int i, bool n):  id(i), note(n) {
    parent=-1; visited=false;
  }
};
```

URI - 1706 - Classe do grafo

```
class grafo {
public:
 int n;
 vector<no> gf;

 grafo(int _n, vector<bool> v):  n(_n) {
  for(int i=0; i<n; i++) {
   gf.push_back(no(i,v[i]));
  }
 }
 void aresta(int x, int y) {
  gf[x].nb.push_back(y);
  gf[y].nb.push_back(x);
  return;
 }
```

URI - 1706 - DFS na classe grafo

```
 bool dfs(int i) {
  gf[i].visited=true;
  for(int j=0; j<(int)gf[i].nb.size(); j++) {
   if(!gf[gf[i].nb[j]].visited) {
    gf[gf[i].nb[j]].parent = i;
    dfs(gf[gf[i].nb[j]].id);
   }
  }
  if(!gf[i].note && gf[i].parent!=-1) {
   gf[i].note=true;
   gf[gf[i].parent].note = !gf[gf[i].parent].note;
  }
  return gf[i].note;
 }
};
```

URI - 1706 - Main

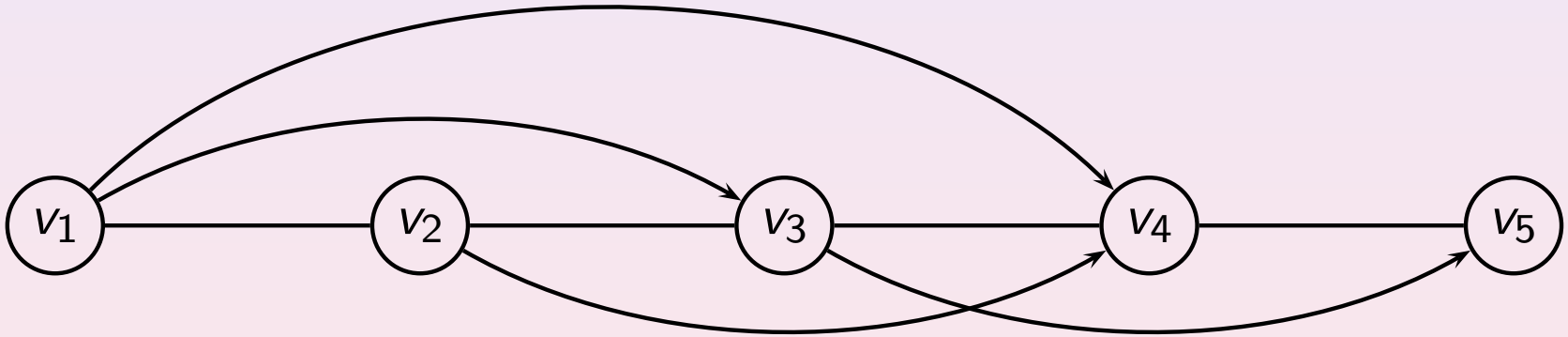
```
int main() {
 int n, m, x, y;
 char c;
 while(cin >> n >> m) {
  vector<bool> v(n);
  for(int i=0; i<n; i++) {
   cin >> c;
   if(c=='A') v[i]=true;
   else v[i]=false;
  }
  grafo g(n,v);
  for(int i=0; i<m; i++) {
   cin >> x >> y; g.aresta(x-1,y-1);
  }
  bool res = true;
  for(int i=0; i<n; i++) {
   if(!g.gf[i].note) res = res && g.dfs(i);
  }
  cout << (res ?  'Y' : 'N') << endl;
 }
 return 0;
}
```

Ordenação Topológica

- Ordenar topologicamente um grafo orientado acíclico significa arranjar seus vértices em seqüência de forma que sempre a cauda de um arco preceda sua cabeça.
- A ordenação é utilizada em aplicações onde eventos ou tarefas têm precedência sobre outros.
- Para que um grafo orientado apresente uma ordenação topológica é necessário que seja acíclico, ou seja, não existe um ciclo orientado no grafo.

Exemplo de uma ordenação topológica

Veja um exemplo abaixo de uma ordenação topológica, dando-se índice aos vértices de acordo com sua posição na seqüência:



Teorema: Ordenação Topológica

Theorem

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

Demonstração.

$\Leftarrow$

Se G possui uma ordenação topológica, então é possível ordenar seus vértices de forma que todos os arcos possuem *cauda* mais à esquerda do que a *cabeça*. Como não há arcos de "retorno", então não é possível formar ciclos orientados, logo o grafo orientado é acíclico. □

Teorema: Ordenação Topológica, provando o $\Rightarrow$

Lemma

Todo o Grafo Orientado Acíclico possui pelo menos um sorvedouro

Demonstração.

Tomemos um caminho orientado *maximal* $P : \{v_1, \ldots, v_n\}$.
Se v_n é um sorvedouro, não há o que provar, vamos supor então que v_n não é um sorvedouro, logo existe vértice u com arco (v_n, u).
Se $u \notin P$, então $P' : \{v_1, \ldots, v_n, u\}$ é maior que P, contradição.
Logo $u \in P$, desta forma, $C : \{u, \ldots, v_n, u\}$ é um ciclo orientado no grafo, contradição.
Concluímos, então que v_n só pode ser um sorvedouro. $\square$

Também é possível provar com a mesma argumentação que um grafo orientado acíclico tem pelo menos um fonte.

Teorema: Ordenação Topológica, provando o $\Rightarrow$

Demonstração.

$\Rightarrow$

D é Acíclico, então possui sorvedouro. Indução no sorvedouro.
Base:O grafo trivial possui Ordenação Topológica.
Hipótese de Indução:Seja v um sorvedouro em D, vamos tomar o grafo $D' = D - v$. Como D' é orientado e acíclico aplicamos a Hipótese de Indução: D' possui uma ordenação topológica.
Passo:Colocamos os vértices de D' em uma seqüência ordenados topologicamente da esquerda para a direita. Recolocamos o vértice v, reconstruindo D, na posição mais à direita da seqüência. Como v é um sorvedouro qualquer arco que incide em v tem sua cabeça em v, logo qualquer arco que incide em v está orientado da esquerda para a direita. A seqüência formada é uma ordenação topológica de D. □

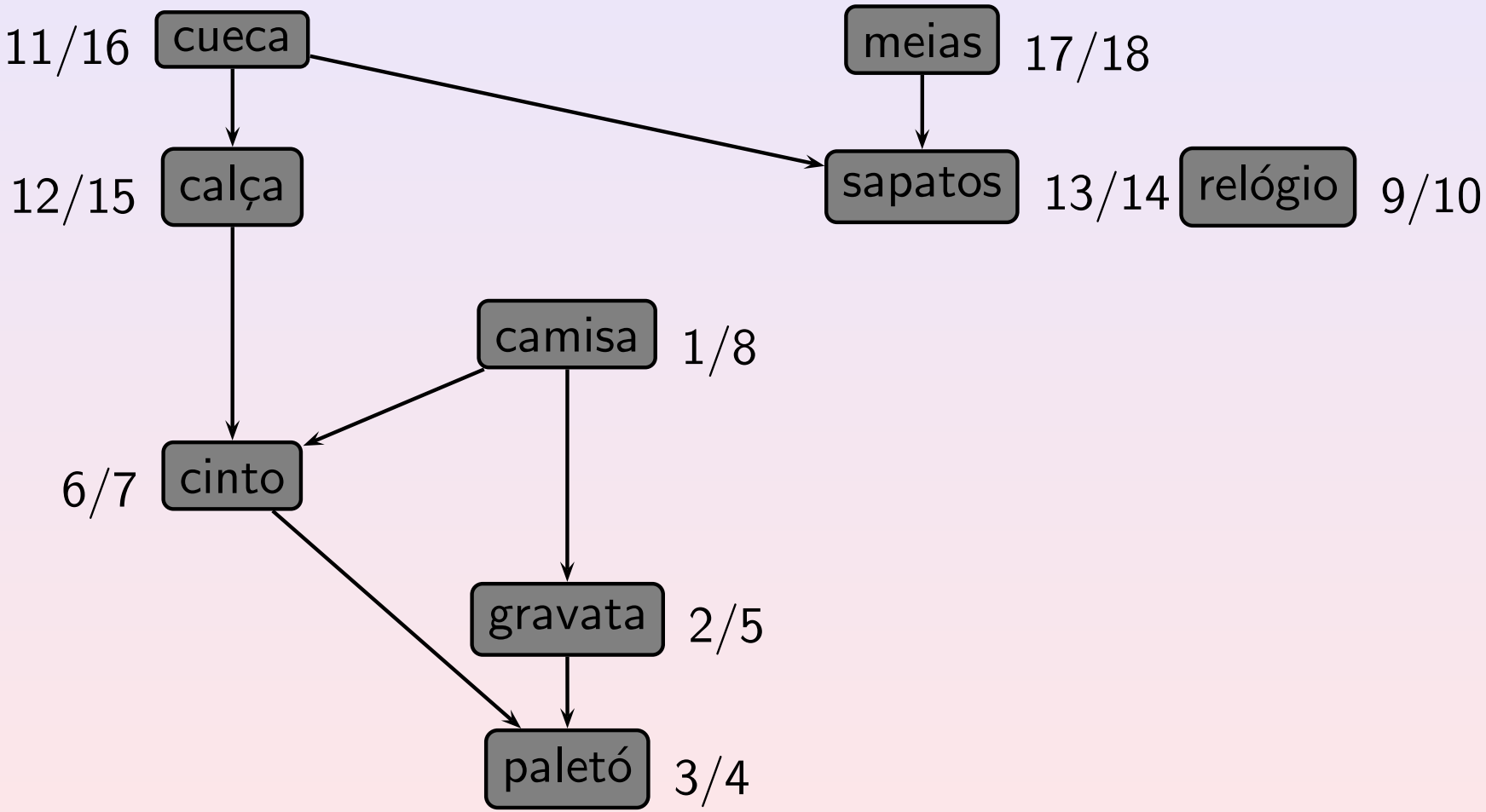
Construindo Ordenação Topológica

OrdenaçãoTopológica(G)

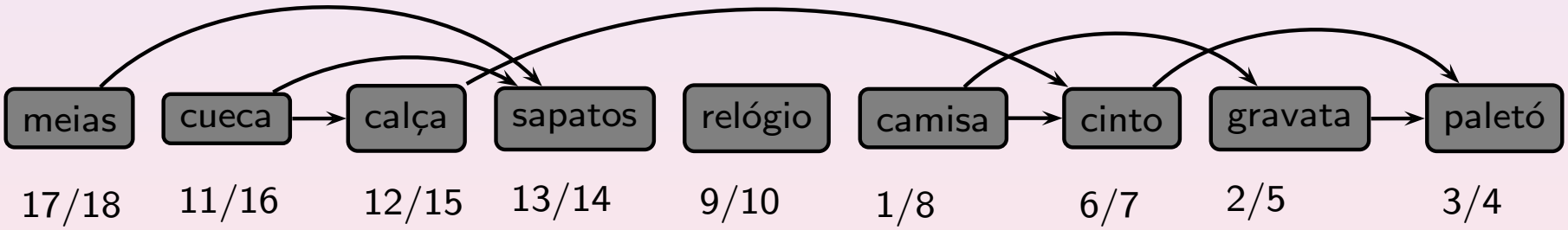
1. Chamar *DFS*(G) para calcular o tempo de término $f[v]$ para cada vértice v.
2. À medida que cada vértice é terminado, inserir o vértice à frente de uma lista ligada.
3. Retornar a lista ligada de vértices

O segundo passo pode ser interpretado como: "Imprima os vértices em *ordem decrescente* de $f[v]$.

Exemplo de construção de uma ordenação topológica



Eventos ordenados



Complexidade de Tempo

- *DFS* é $O(V + E)$
- Inserir vértice em uma lista é $O(V)$
- Logo o *OrdenaçãoTopológica(G)* é $O(V + E)$

Prova de corretude do algoritmo

Lemma

O Grafo Orientado é Acíclico se e somente se DFS não possui arcos de "retorno".

Demonstração.

$\Rightarrow$ (contrapositiva)
Suponha que (u, v) é aresta de retorno no *DFS*.
Então, v é ancestral de u, ou seja existe o caminho $P : \{v, \ldots, u\}$.
$P \cup \{(u, v)\}$ é um ciclo orientado. $\square$

Prova de corretude do algoritmo

Lemma

O Grafo Orientado é Acíclico se e somente se DFS não possui arcos de "retorno".

Demonstração.

$\Leftarrow$ (contrapositiva)
Suponha que o grafo possua um *Ciclo orientado* $C : \{v_1, \ldots, v_n, v_1\}$.
Suponha neste ciclo que v_1 é o primeiro vértice a ser descoberto.
Então no instante $d[v_1]$ existe um "caminho branco" de v_1 a v_n, e pelo teorema do caminho branco, v_k é um descendente de v_1, desta forma, no *DFS* (v_n, v_1) é uma aresta de retorno. □

Teorema: Corretude do algoritmo de Ordenação Topológica

Theorem

O OrdenaçãoTopológica(G) constrói uma ordenação topológica do grafo orientado acíclico G

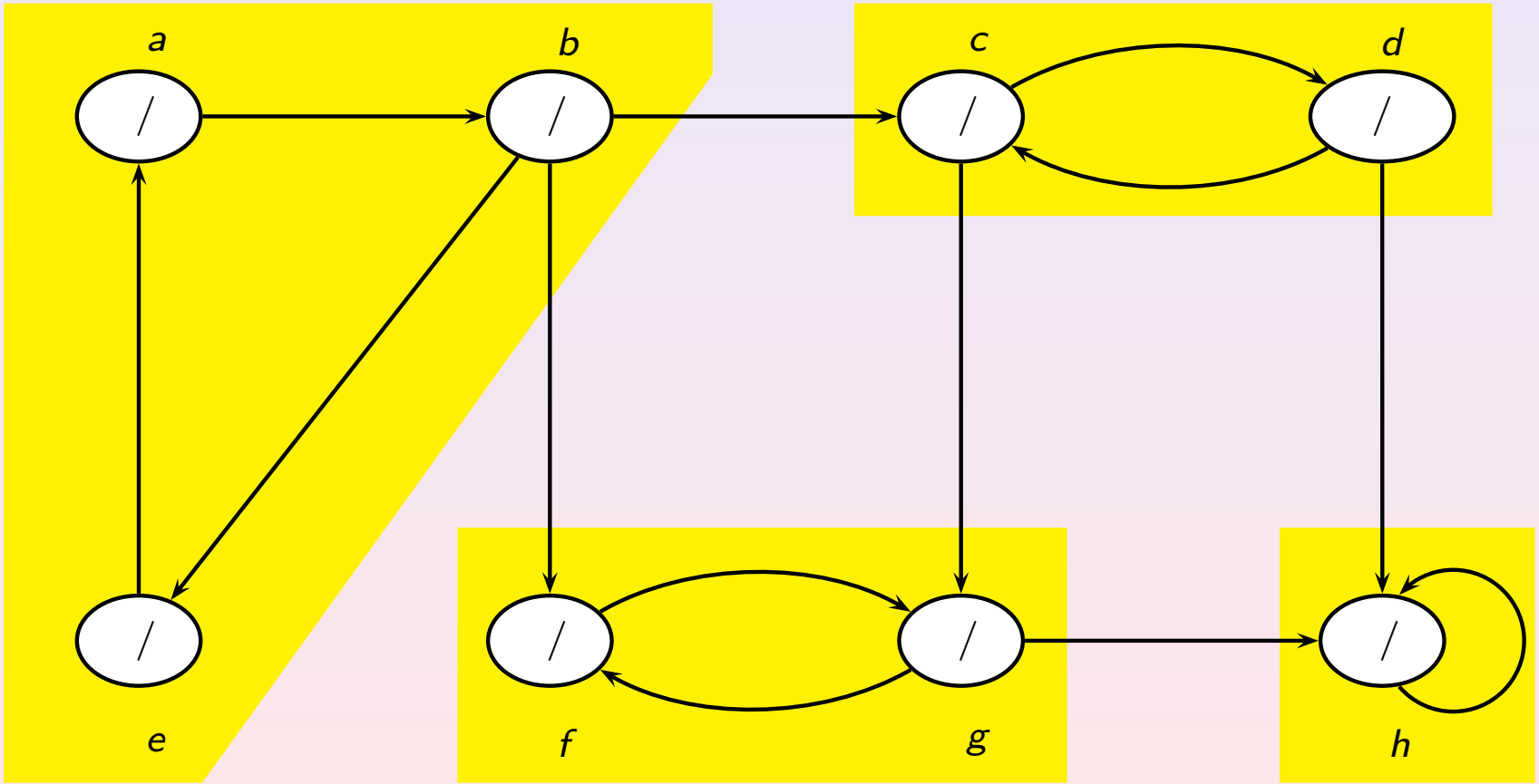
Demonstração.

Como o algoritmo constrói a ordenação topológica em ordem decrescente de tempo f, basta mostrar que se (u, v) é uma aresta do grafo, então $f[u] > f[v]$.
Vamos considerar o instante em que (u, v) é examinada no *DFS*.
u é *CINZA*, v não pode ser *CINZA* senão (u, v) seria uma aresta de retorno, e pelo lema anterior não há aresta de retorno no grafo.
Logo v é *BRANCO* ou *PRETO*.
Se v é *BRANCO*, então v é descendente de u, logo $f[v] < f[u]$.
Se v é *PRETO*, então v já foi finalizado e $f[v]$ já foi definido, logo $f[v] < f[u]$. □

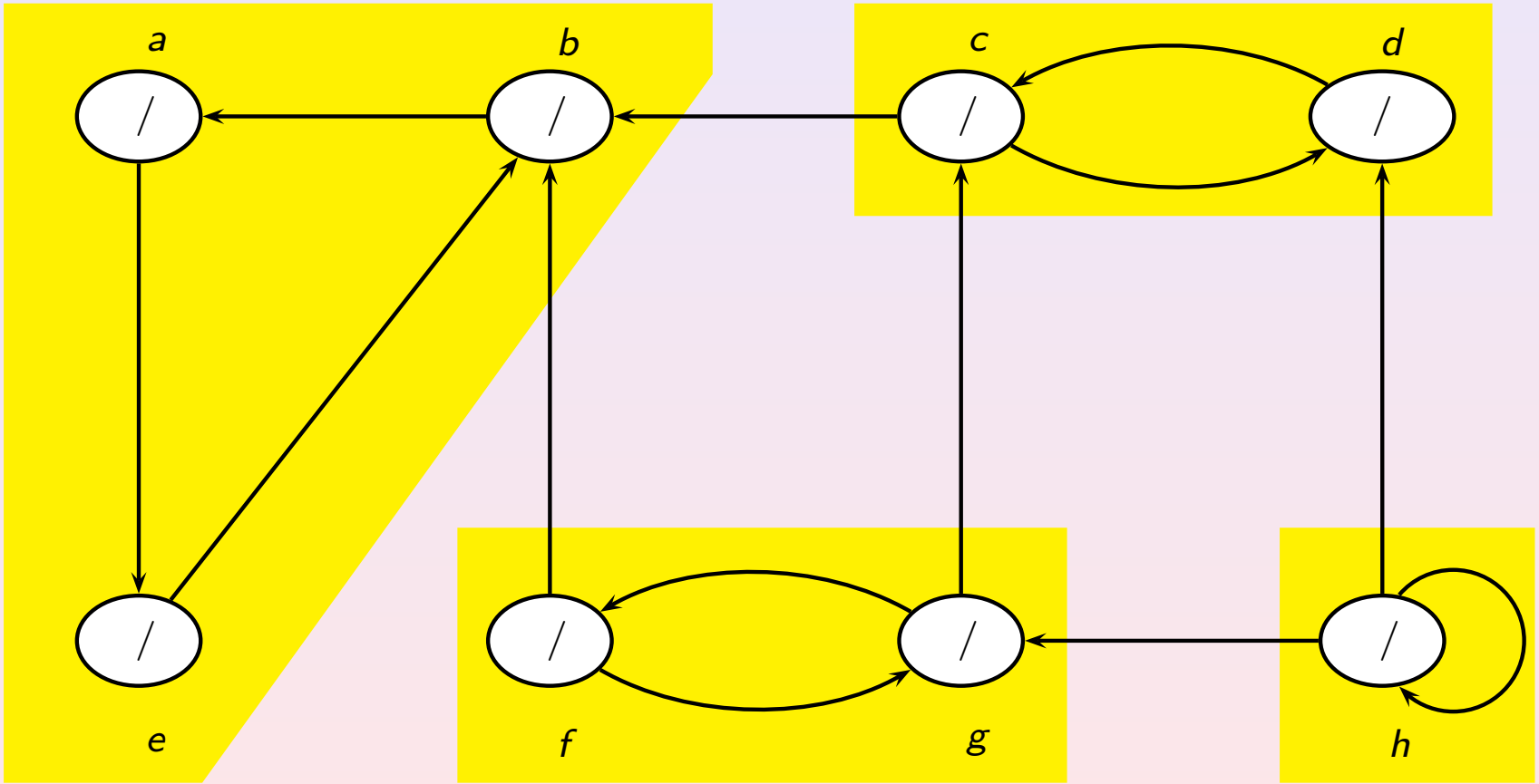
Componentes Fortemente Conexos

- Outra aplicação da busca em profundidade é a decomposição de um grafo em seus COMPONENTES FORTEMENTE CONEXOS.
- A divisão em componentes pode agilizar a aplicação de algum algoritmo.
- O algoritmo prevê a aplicação do *DFS* no grafo e em seu transposto.
- O grafo transposto D^T de D é dado por $D^T = (V^T, E^T)$ tal que:
  - $V^T = V$ e
  - $E^T = \{(u, v) : (v, u) \in E\}$

Componentes fortemente Conexos de um Grafo Orientado



Grafo Orientado Transposto



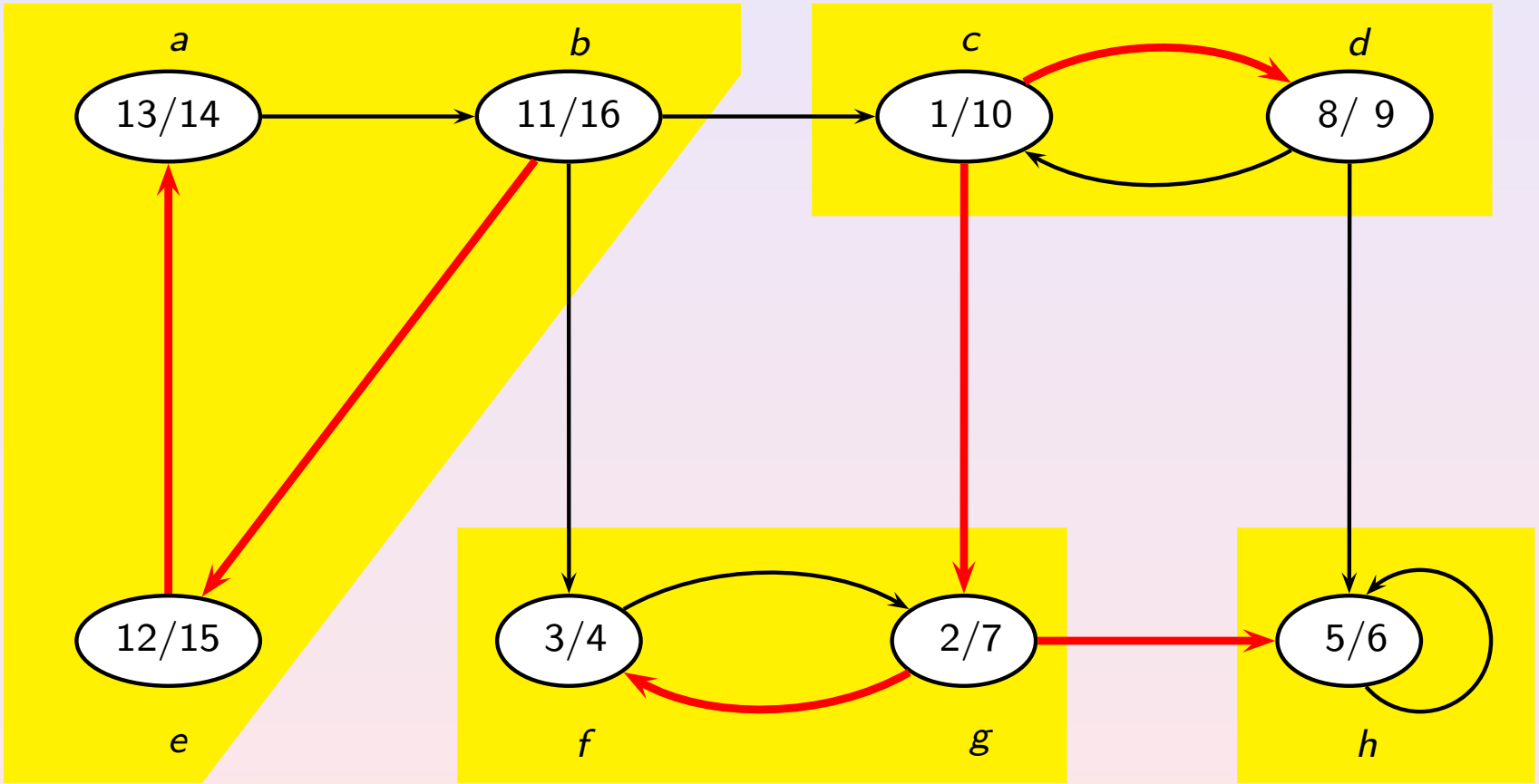
Algoritmo para separar os Componentes Fortemente Conexos

ComponentesFortementeConexos(D)

1. Execute $DFS(D)$ para obter $f[v]$ para todo $v \in V$.
2. Encontre D^T a partir de D
3. Execute $DFS(D^T)$ considerando os vértices em ordem decrescente de $f[v]$.
4. Devolva os *conjuntos de vértices* de cada *árvore* da *Floresta de Busca em Profundidade* obtida na última busca.

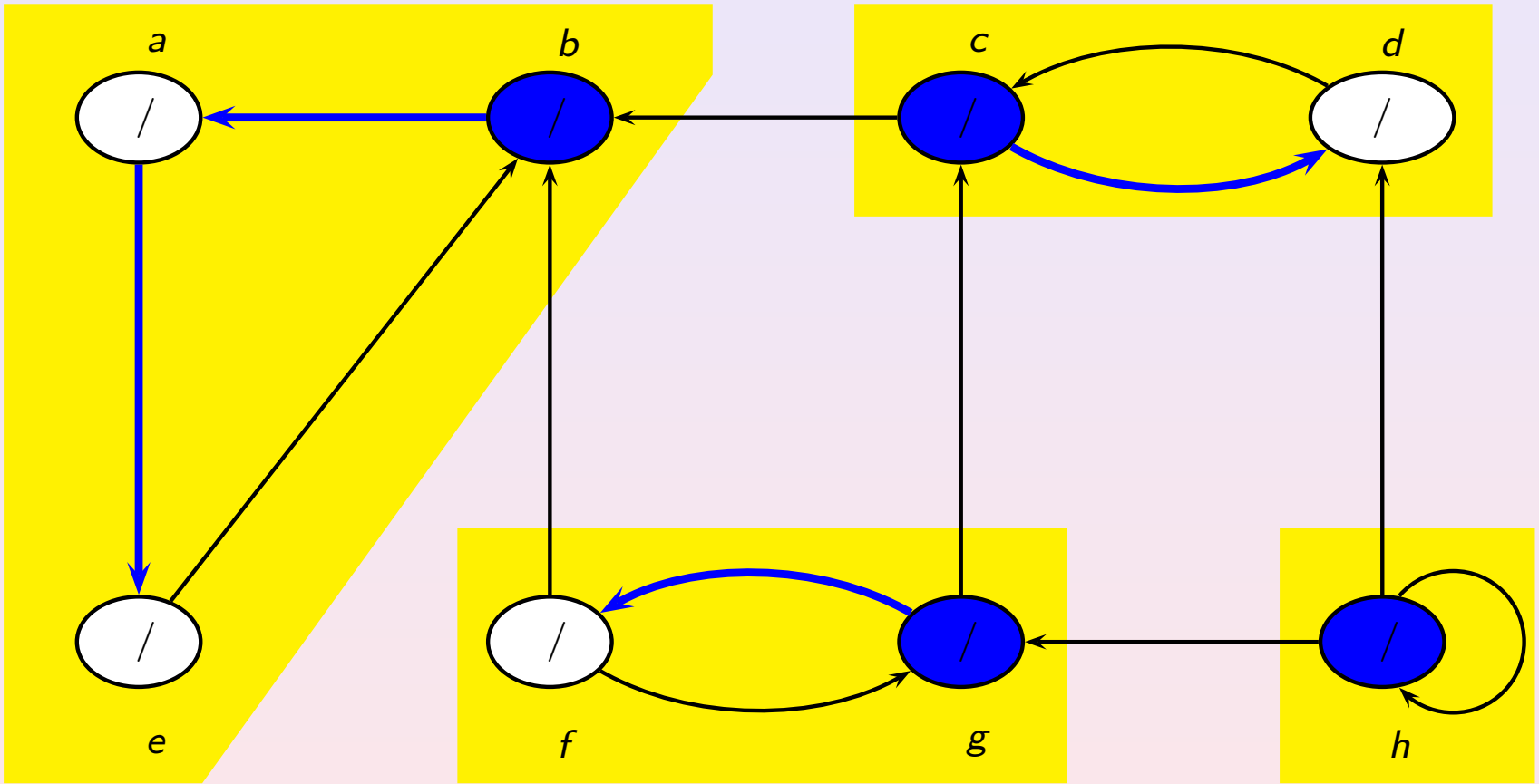
Cada árvore da floresta é possui como vértices os vértices de um conjunto fortemente conexo de D.

Aplicando o Algoritmo



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

Grafo Orientado Transposto

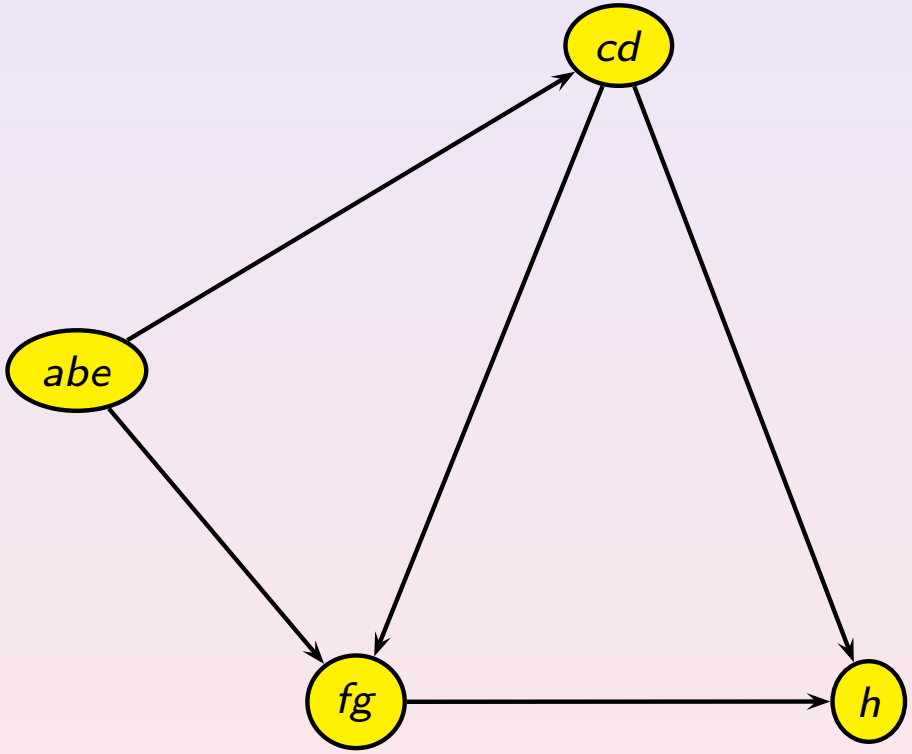


3. Execute $DFS(D^T)$ vértices em ordem decrescente de $f[v]$.
4. Devolva os *conjuntos de vértices* de cada *árvore*.

Grafo Componente

- A prova da corretude do algoritmo utiliza um recurso de demonstração que depende da contração de cada componente conexa em apenas um vértice.
- Esta contração é obtida realizando uma operação de contração de aresta para cada arco que possua como extremos vértices da mesma componente convexa.
- O grafo resultante, que chamamos D^{CFC} é acíclico.
- Cada componente conexo passa a ser visitado em ordem topológica com relação a D^{CFC}.

Grafo Componente: D^{CFC}



Lema: O grafo D^{CFC} é acíclico

Lemma

Sejam C e C' dois componentes fortemente conexos de D
Sejam $u, v \in V(C)$ e $u', v' \in V(C')$
Suponha que existe um caminho $P : \{u, \ldots, u'\}$ em D, então não existe um caminho $P' : \{v', \ldots, v\}$ em D.

Demonstração.

Por contradição. Vamos considerar que exista o caminho $P' : v', \ldots, v$ em G.
Neste caso existem os caminhos: $P_{v',v} = P'$ e $P_{v,v'} = \{v, \ldots, u\} \cup P \cup \{u', \ldots, v'\}$ o que é uma contradição, pois se existessem ambos caminhos, v e v' estariam na mesma componente conexa. □

Corretude do algoritmo: Lema ordem topológica

Sejam d e f os tempos obtidos na *DFS* sobre o grafo D original (não transposto).

Definition

Para todo subconjunto U de vértices defina-se:

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

Lemma

Sejam C e C' dois componentes fortemente conexos de D. Suponha que existe (u, v) em E onde $u \in C$ e $v \in C'$, então $f(C) > f(C')$

Corretude do algoritmo: Lema ordem topológica

Demonstração.

Precisamos identificar qual componente teve o primeiro vértice descoberto.
Se $d(C) < d(C')$, seja x o primeiro vértice descoberto em C. No tempo $d[x]$, todos os vértices de C e C' são brancos. Para todos vértices em C existe um caminho branco partindo de x como $(u, v) \in E$, então também existe caminho branco de x para todo vértice em C'. Com isto, $f[x] = f(C) > f(C')$.
Se $d(C') < d(C)$, seja y o primeiro vértice descoberto em C' todos vértices de C' serão visitados e como $(u, v) \in E$, então não há caminho ligando y a qualquer vértice de C, então $f[y] = f(C') < d(C) < f(C)$. □

Corretude do algoritmo: Corolário da ordem topológica

Corollary

Sejam C e C' componentes fortemente conexos distintos no grafo orientado $D = (V, E)$. Suponha que exista uma aresta $(u, v) \in E^T$, onde $u \in C$ e $v \in C'$. Então $f(C) < f(C')$.

Demonstração.

Como $(u, v) \in E^T$, então $(v, u) \in E$. Pelo lema anterior, $f(C') > f(C)$ □

Teorema de Prova de Corretude do algoritmo

Theorem

O algoritmo ComponentesFortementeConexos(D) calcula corretamente os componentes fortemente conexos de D.

Demonstração.

Prova no grafo de componentes D^{CFC}. Como D^{CFC} é acíclico, existe uma ordenação topológica em D^{CFC}, o vértice C mais à esquerda da ordenação tem $f(C)$ máximo.
Em $(D^{CFC})^T$ ele será o primeiro na busca em profundidade.
Nesta busca, não existe arco partindo do vértice, este componente será uma árvore isolada. Isto representa que os vértices de D que pertençam a C pertence a uma árvore do componente conexo.
Esta é uma escolha gulosa, por indução podemos perceber que o *DFS* no $(D^{CFC})^T$ calcula corretamente os componentes fortemente conexos em D. □

Atividades baseadas no CLRS

- Ler os capítulos 22 introdução, 22.1 e 22.2.
- Resolver exercícios: 22.1-1, 22.2-1, 22.2-2, 22.2-3, 22.2-4, 22.2-5, 22.2-6[1], 22.2-7, 22.2-8.
- Ler o capítulo 22.3.
- Resolver exercícios: 22.3-2, 22.3-3, 22.3-4, 22.3-6, 22.3-7, 22.3-8, 22.3-9, 22.3.10, 22.3.11,
- Ler o capítulo 22.4.
- Resolver exercícios: 22.4-1.
- Ler o capítulo 22.5.
- Resolver exercícios: 22.5-1, 22.5-2, 22.5-3, 22.5-7.
- Resolver os problemas: 22-2, 22-3.

[1] Podemos reescrever o enunciado: Dado um grafo bipartido, mostre um algoritmo que encontre a bipartição ou que o grafo não é bipartido (ciclo ímpar)