Projeto e Análise de Algoritmos

A. G. Silva

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

11 de maio de 2018

Conteúdo programático

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

Cronograma

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

Buscas em grafos

Busca em grafos

- Grafos são estruturas mais complicadas do que listas, vetores e árvores (binárias).
  Precisamos de métodos para explorar/percorrer um grafo (orientado ou não-orientado).
- Busca em largura (breadth-first search)
  Busca em profundidade (depth-first search)
- Pode-se obter várias informações sobre a estrutura do grafo que podem ser úteis para projetar algoritmos eficientes para determinados problemas.

Notação

- Para um grafo G (orientado ou não) denotamos por $V[G]$ seu conjunto de vértices e por $E[G]$ seu conjunto de arestas.
- Para denotar complexidades nas expressões com O ou Θ usaremos V e E em vez de $|V[G]|$ ou $|E[G]|$. Por exemplo, $\Theta(V + E)$ ou $O(V^2)$.

Busca em largura

- Dizemos que um vértice v é **alcançável** a partir de um vértice s em um grafo G se existe um caminho de s a v em G.
- Definição: a distância de s a v é o comprimento de um caminho mais curto de s a v.
- Se v **não é alcançável** a partir de s, então dizemos que a distância de s a v é ∞ (*infinita*).

Busca em largura

- Busca em largura recebe um grafo $G = (V, E)$ e um vértice especificado s chamado fonte (*source*).
- Percorre todos os vértices alcançáveis a partir de s em ordem de distância deste. Vértices a mesma distância podem ser percorridos em qualquer ordem.
- Constrói uma Árvore de Busca em Largura com raiz s. Cada caminho de s a um vértice v nesta árvore corresponde a um caminho mais curto de s a v.

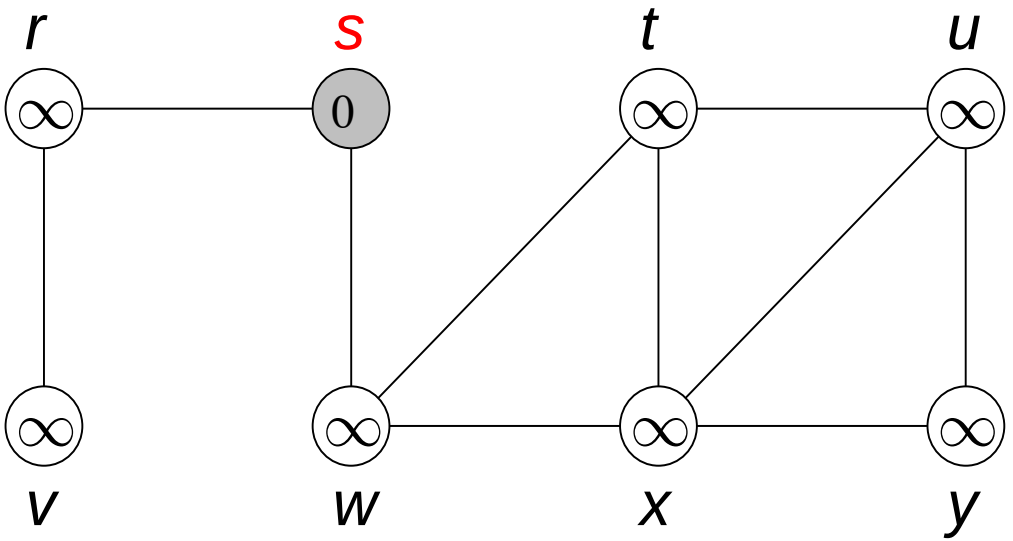
Busca em largura

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

Busca em largura

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

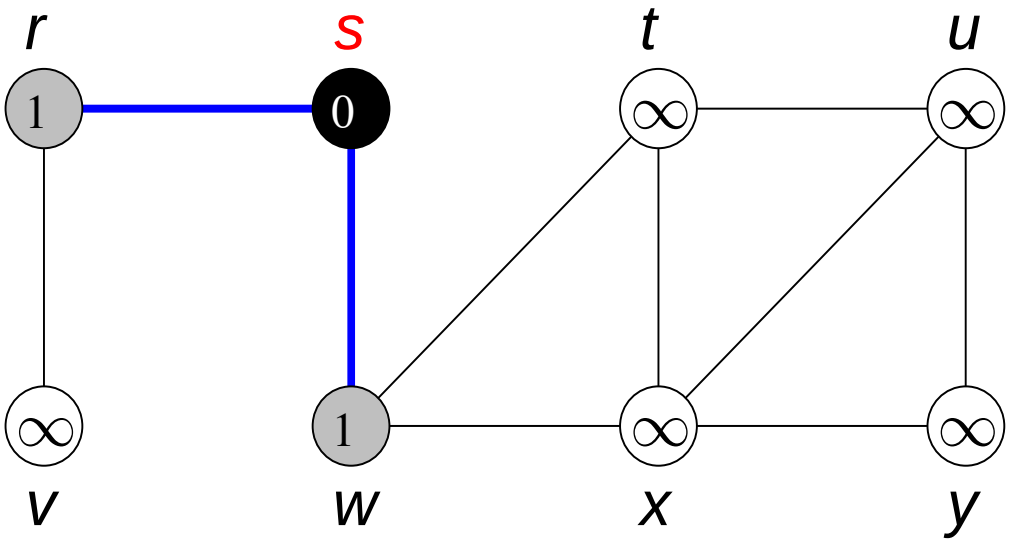
Exemplo (CLRS)



Q s

0

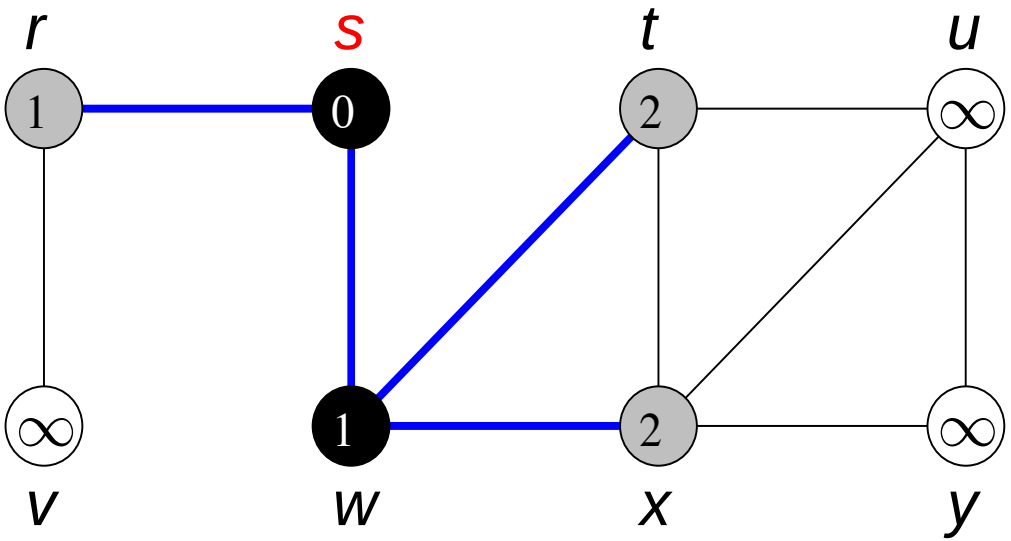
Exemplo (CLRS)

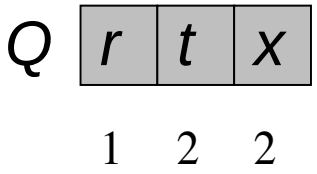


Q | w | r

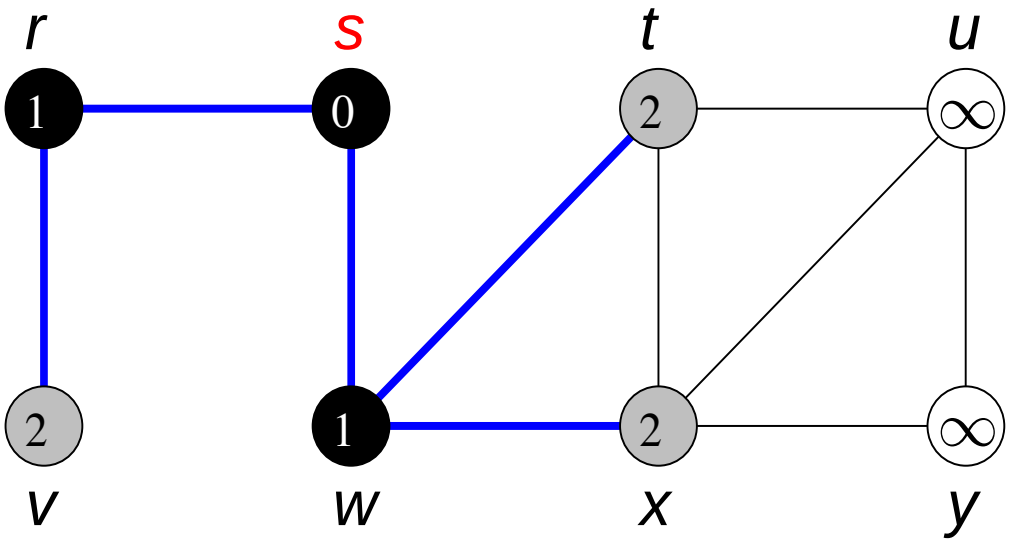
1 1

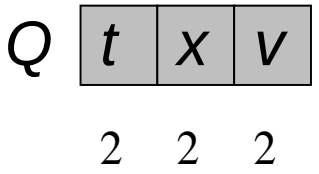
Exemplo (CLRS)



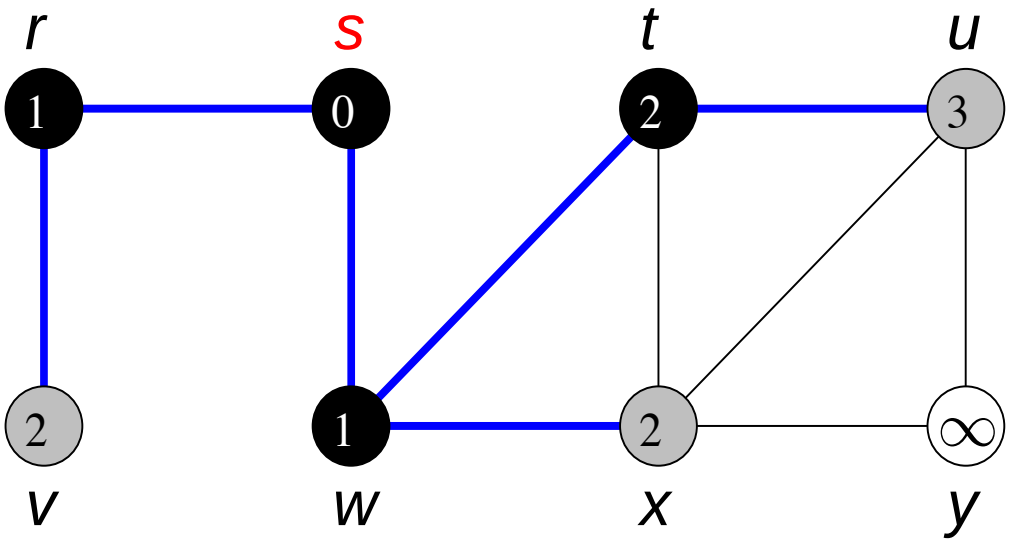


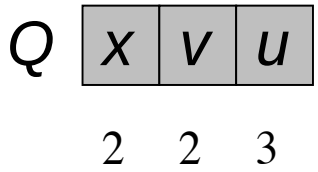
Exemplo (CLRS)



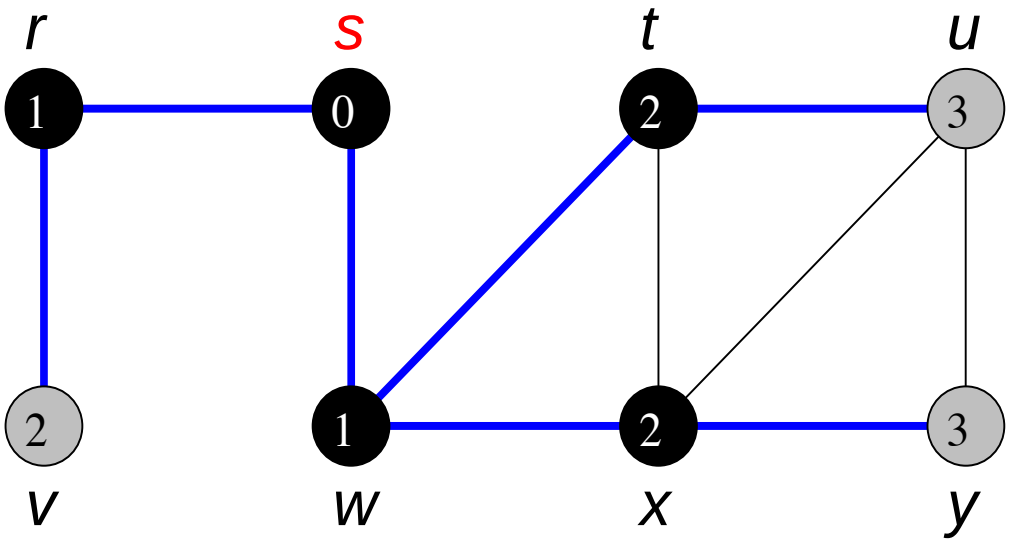


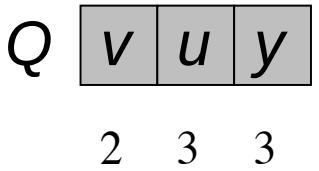
Exemplo (CLRS)



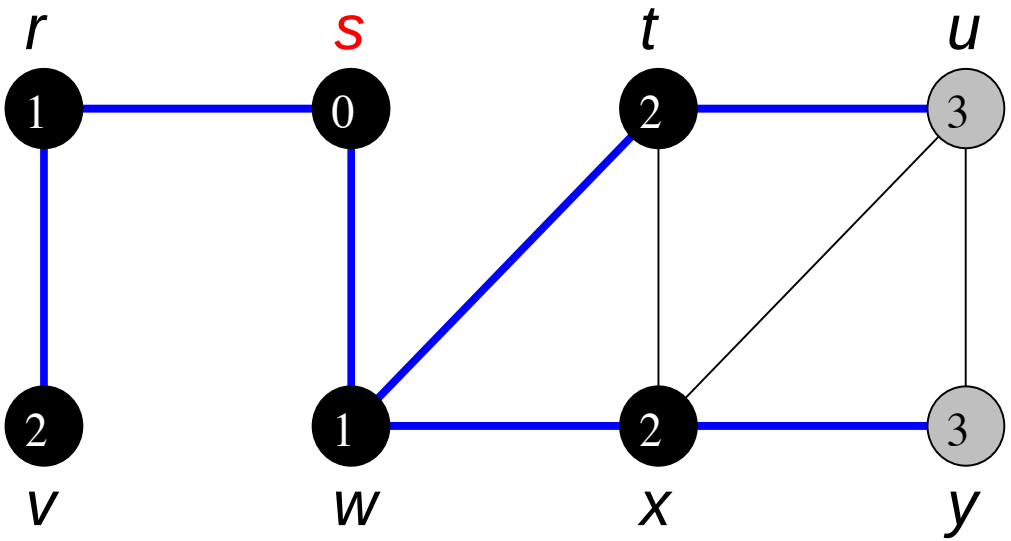


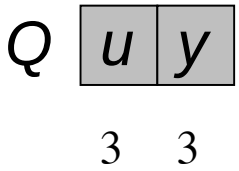
Exemplo (CLRS)



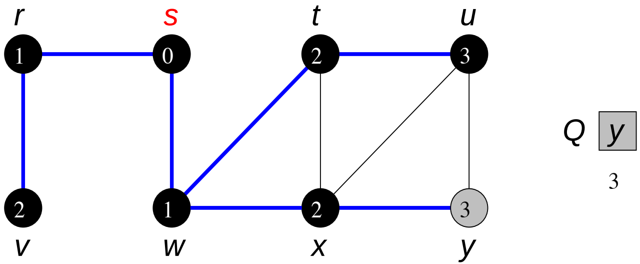


Exemplo (CLRS)

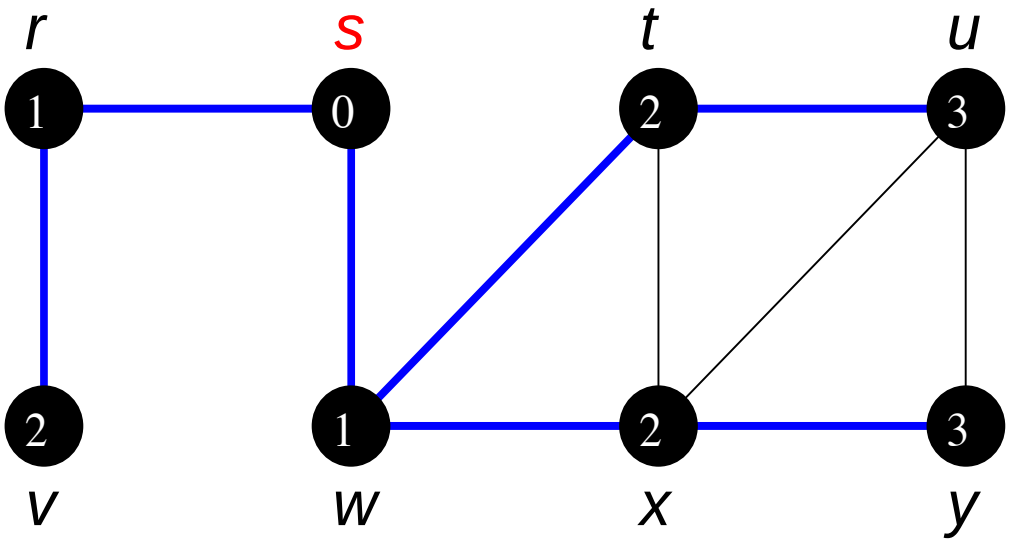




Exemplo (CLRS)



Exemplo (CLRS)



Q $\emptyset$

Cores

- Para cada vértice v guarda-se sua cor atual cor[v] que pode ser branco, cinza ou preto.
- Para efeito de implementação, isto não é realmente necessário, mas facilita o entendimento do algoritmo.

Representação da árvore e das distâncias

- A raiz da Árvore de Busca em Largura é s.
- Cada vértice v (diferente de s) possui um pai $\pi[v]$.
- O caminho de s a v na Árvore é dado por:

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

- Uma variável $d[v]$ é usada para armazenar a distância de s a v (que será determinada durante a busca).

Busca em largura

Recebe um grafo G (na forma de listas de adjacências) e um vértice $s \in V[G]$ e devolve
(i) para cada vértice v, a distância de s a v em G e
(ii) uma Árvore de Busca em Largura.

BUSCA-EM-LARGURA(G, s)

```
 0  ▷ Inicialização
 1  para cada u ∈ V[G] − {s} faça
 2        cor[u] ← branco
 3        d[u] ← ∞
 4        π[u] ← NIL
 5  cor[s] ← cinza
 6  d[s] ← 0
 7  π[s] ← NIL
```

Busca em largura

```
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  enquanto Q ≠ ∅ faça
11      u ← DEQUEUE(Q)
12      para cada v ∈ Adj[u] faça
13          se cor[v] = branco então
14              cor[v] ← cinza
15              d[v] ← d[u] + 1
16              π[v] ← u
17              ENQUEUE(Q, v)
18      cor[u] ← preto
```

Consumo de tempo

Método de análise agregado.

- A inicialização consome tempo $\Theta(V)$.
- Depois que um vértice deixa de ser branco, ele não volta a ser branco novamente. Assim, cada vértice é inserido na fila Q no máximo uma vez. Cada operação sobre a fila consome tempo $\Theta(1)$ resultando em um total de $O(V)$.
- Em uma lista de adjacência, cada vértice é percorrido apenas uma vez. A soma dos comprimentos das listas é $\Theta(E)$. Assim, o tempo gasto para percorrer as listas é $O(E)$.

Complexidade de tempo

Conclusão:

A complexidade de tempo de BUSCA-EM-LARGURA é $O(V + E)$

Caminho mais curto

Imprime um caminho mais curto de s a v.

```
Print-Path(G, s, v)
1   se v = s então
2       imprime s
3   senão
4       se π[v] = NIL então
4           imprime não existe caminho de s a v.
5       senão
6           Print-Path(G, s, π[v])
7           imprime v.
```

Busca em profundidade

Depth First Search = busca em profundidade

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

Busca em profundidade

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

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

Busca em profundidade

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

- Suponha que a busca atingiu um vértice u.
- Escolhe-se um vizinho não visitado v de u para prosseguir a busca.
- Empilhe v e repete-se o passo anterior com v.
- Se nenhum vértice não visitado foi encontrado, então desempilhe um vértice da pilha, digamos u, e volte ao primeiro passo.

Floresta de Busca em Profundidade

- A busca em profundidade associa a cada vértice x um predecessor $\pi[x]$.
- O subgrafo induzido pelas arestas

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

  é a Floresta de Busca em Profundidade.
- Cada componente desta floresta é uma Árvore de Busca em Profundidade.

Cores dos vértices

A medida que o grafo é percorrido, os vértices visitados vão sendo coloridos.

Cada vértice tem uma das seguintes cores:

- Cor branca $=$ “vértice ainda não visitado”’.
- Cor cinza $=$ “vértice visitado mas ainda não finalizado”.
- Cor preta $=$ “vértice visitado e finalizado”.

Estampas/rótulos

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

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

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

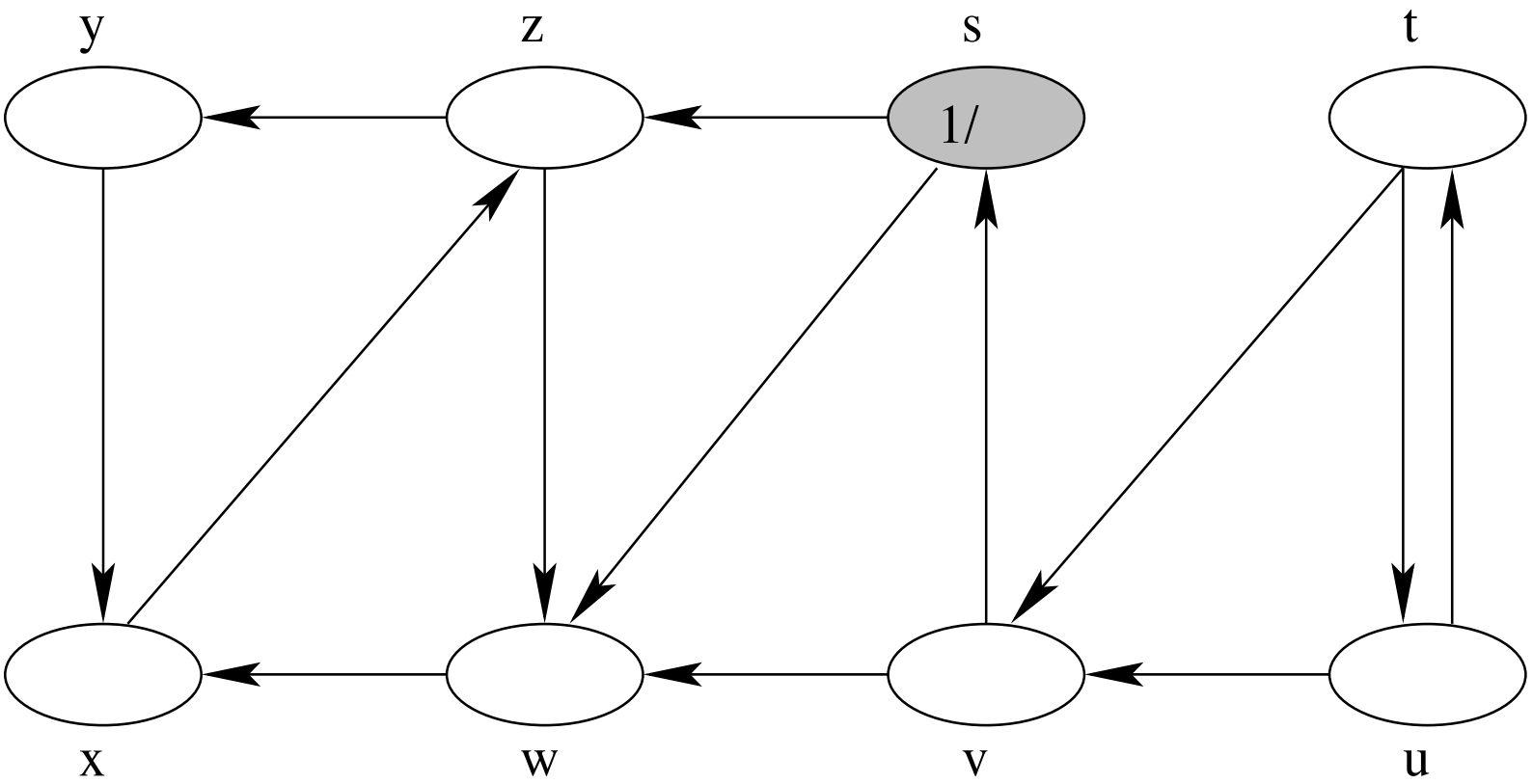
Classificação de arestas

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

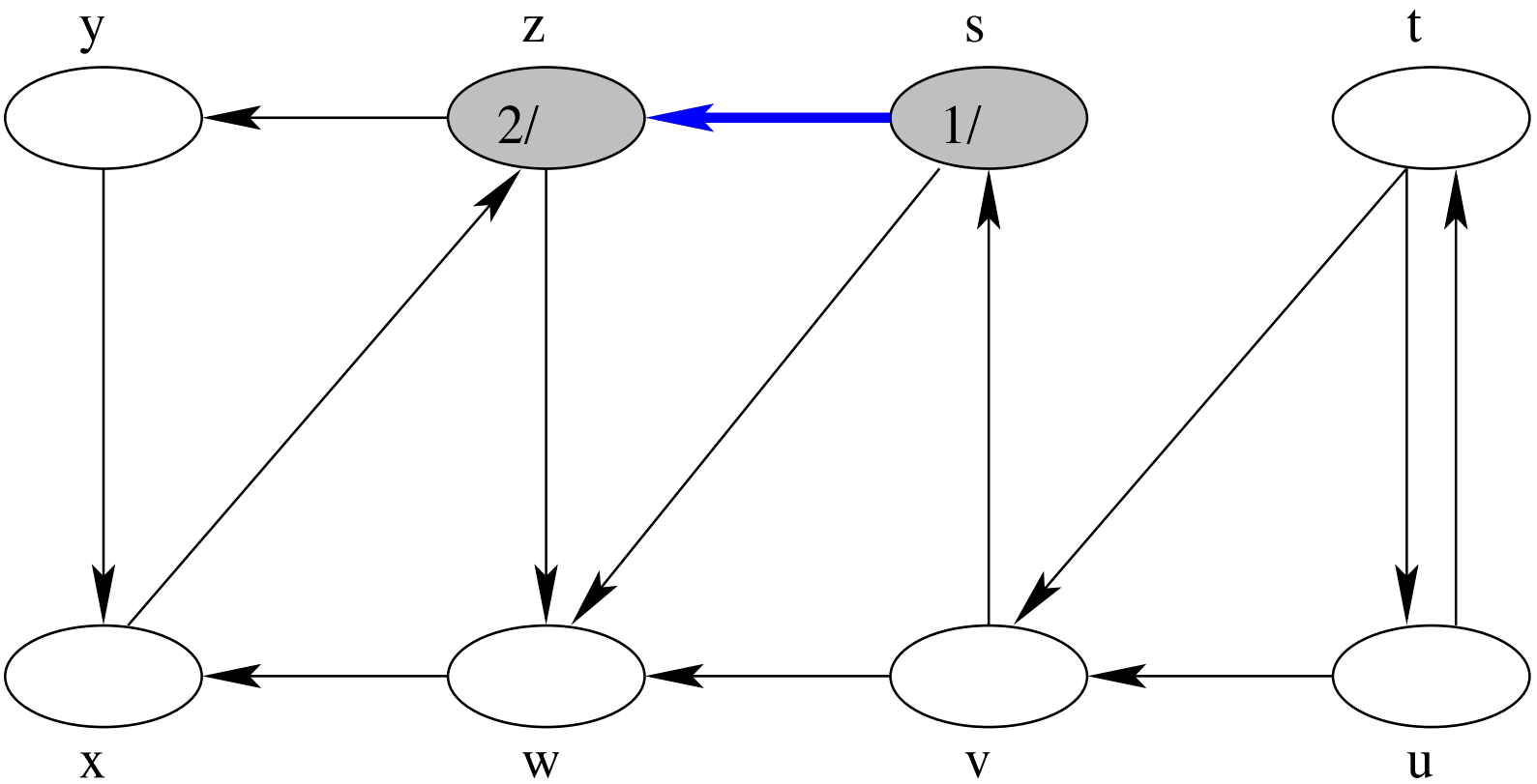
Ela classifica as arestas em quatro tipos:

- Arestas da árvore: arestas que pertencem à Floresta de BP.
- B Arestas de retorno: arestas (u, v) ligando um vértice u a um ancestral v na Árvore de BP. (**B**ackward)
- F Arestas de avanço: arestas (u, v) ligando um vértice u a um descendente próprio v na Árvore de BP. (**F**orward)
- C Arestas de cruzamento: todas as outras arestas.

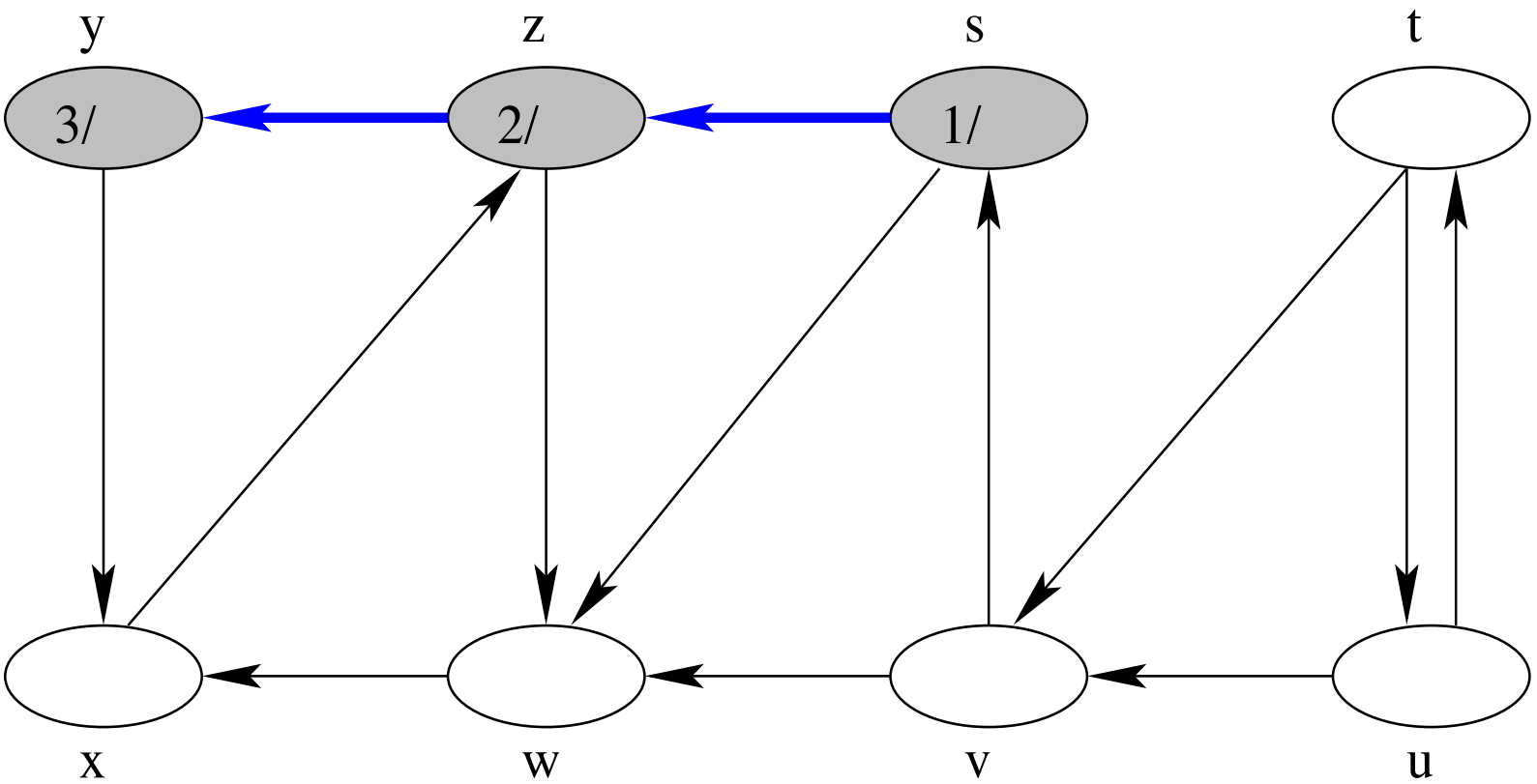
Exemplo



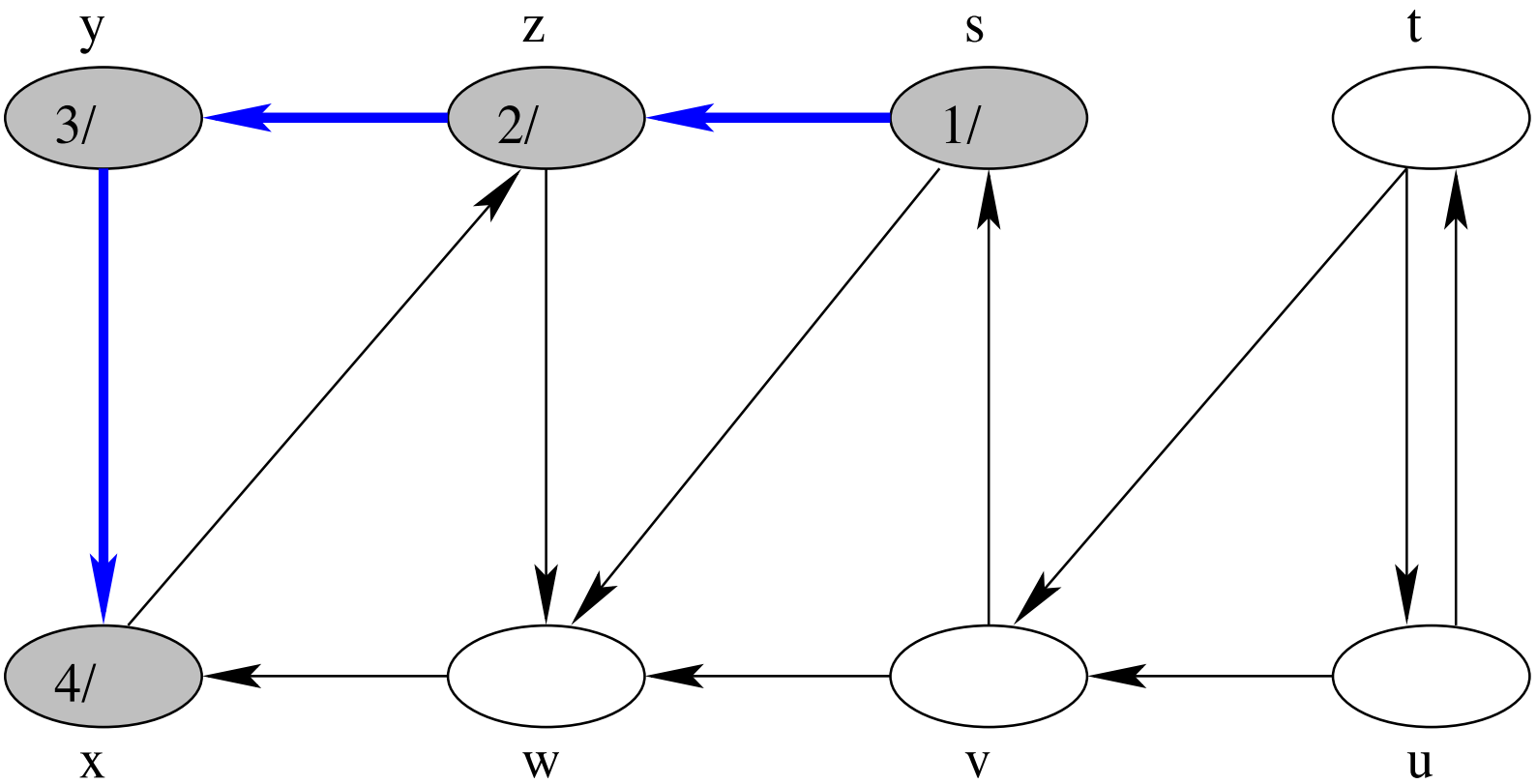
Exemplo



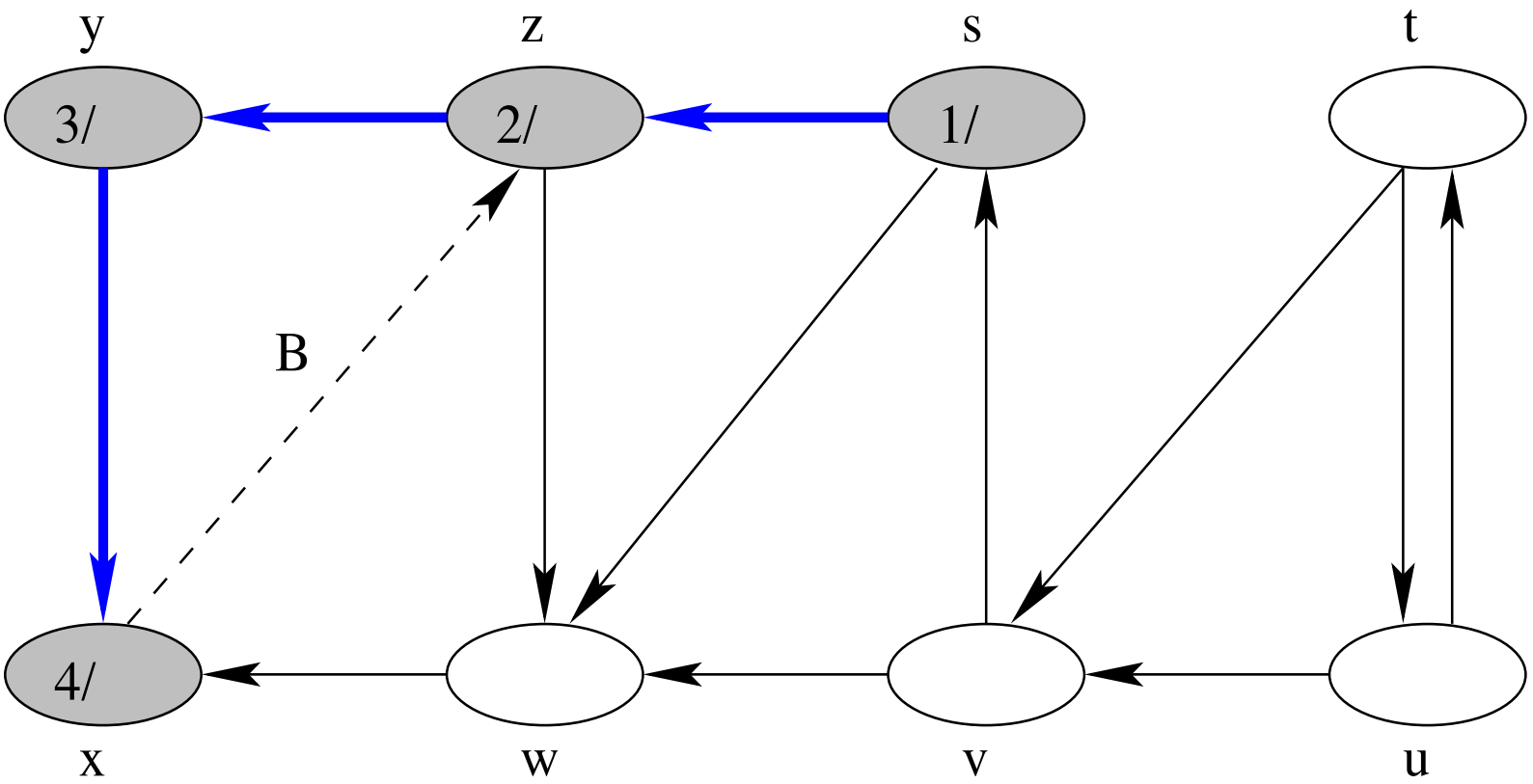
Exemplo



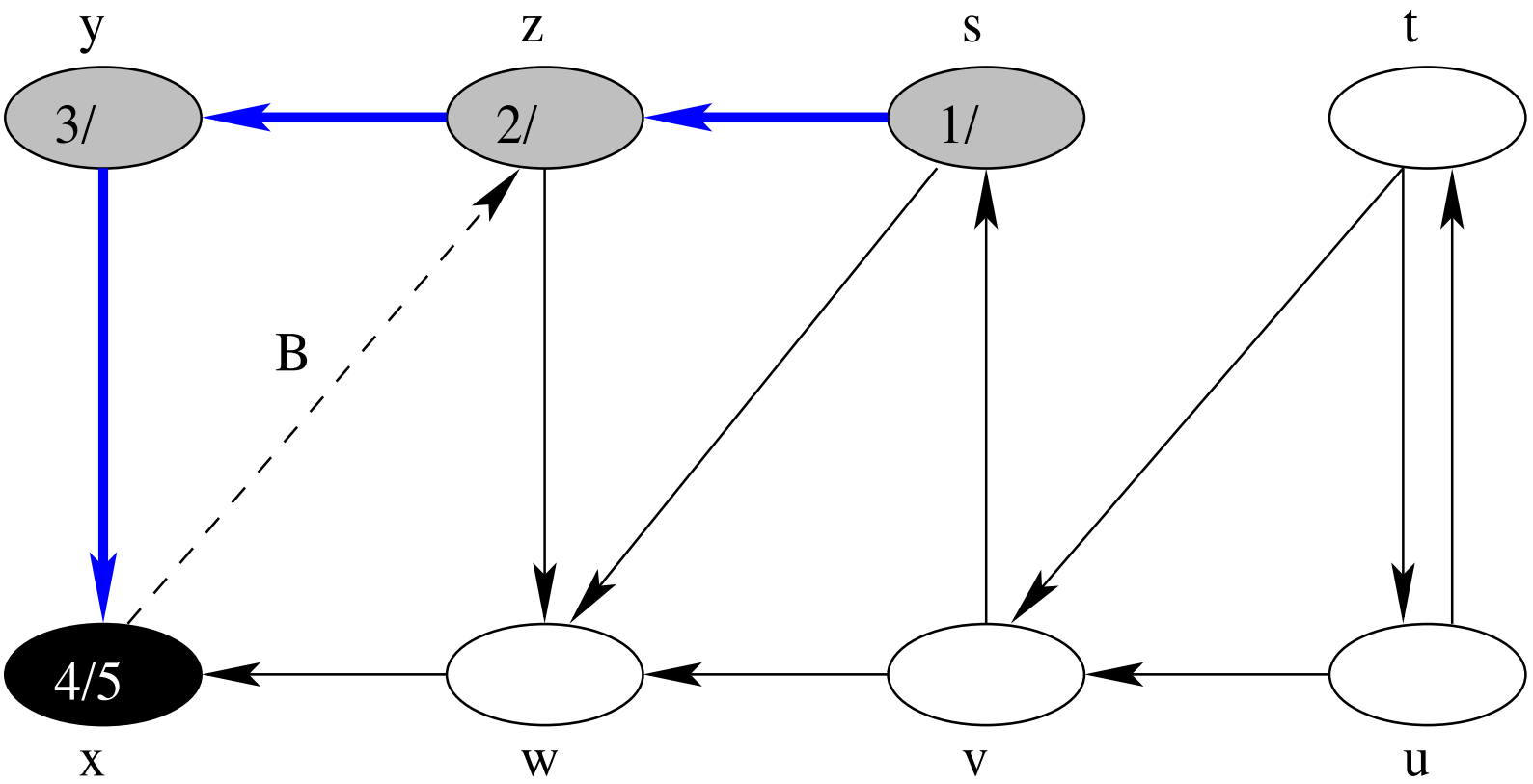
Exemplo



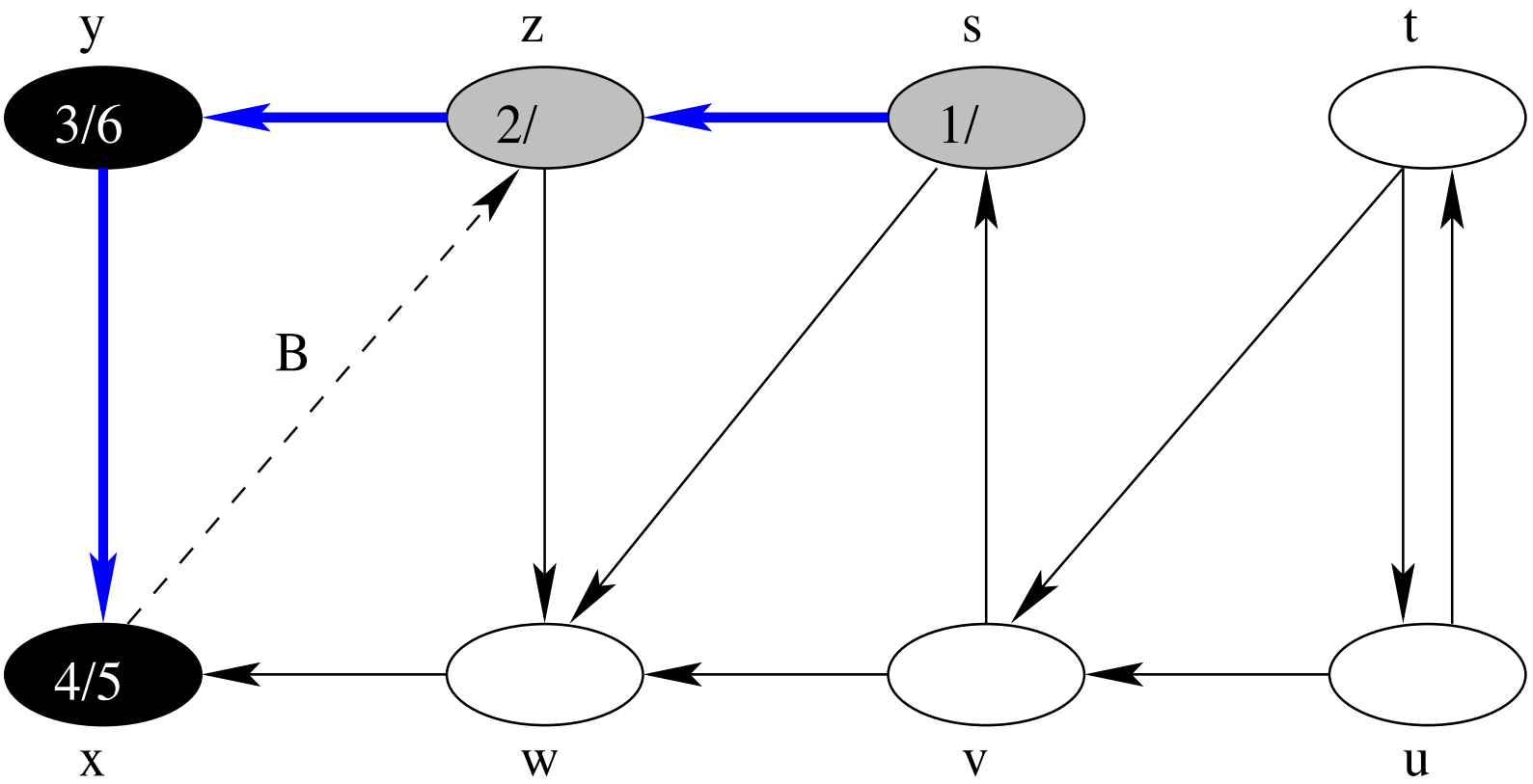
Exemplo



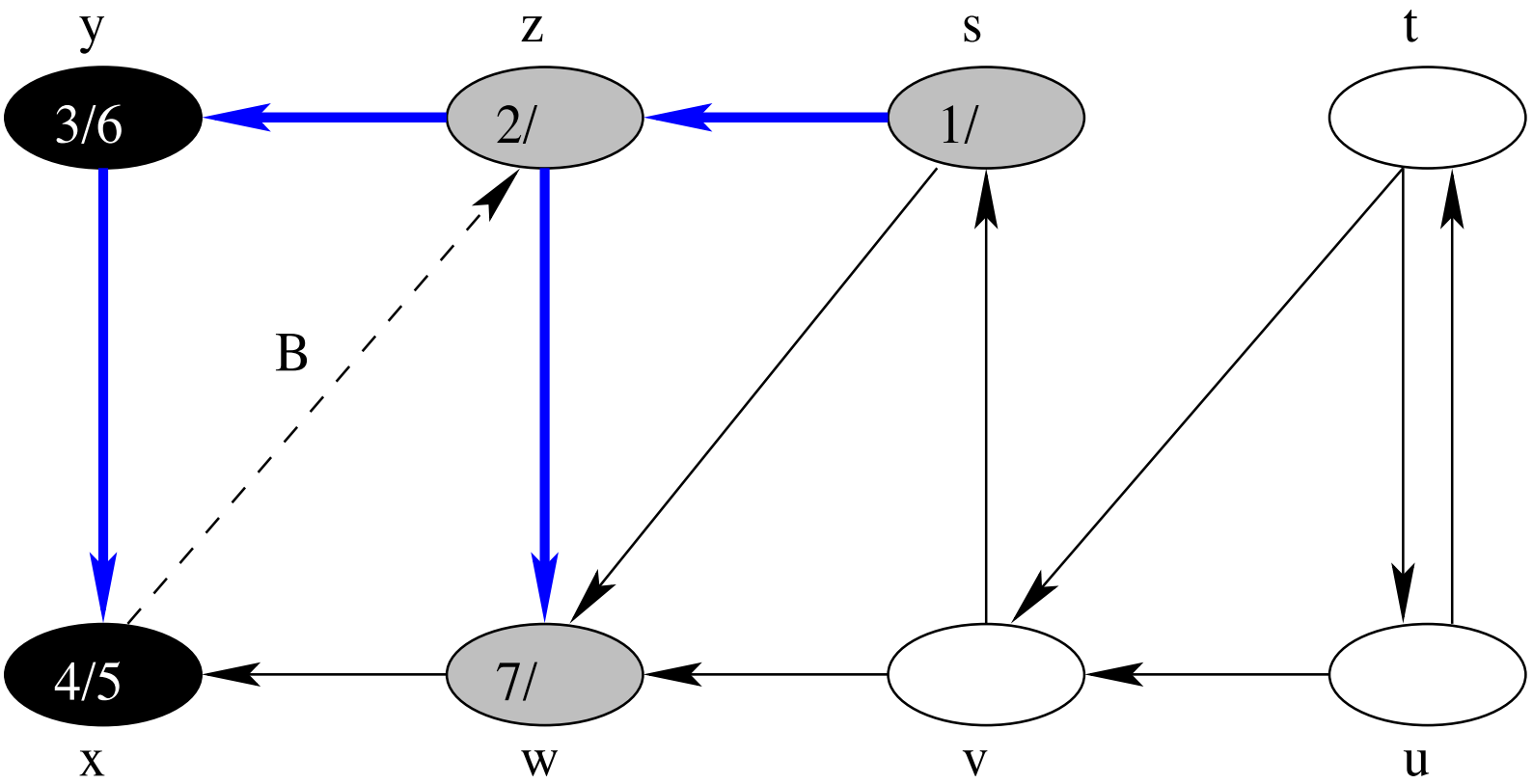
Exemplo



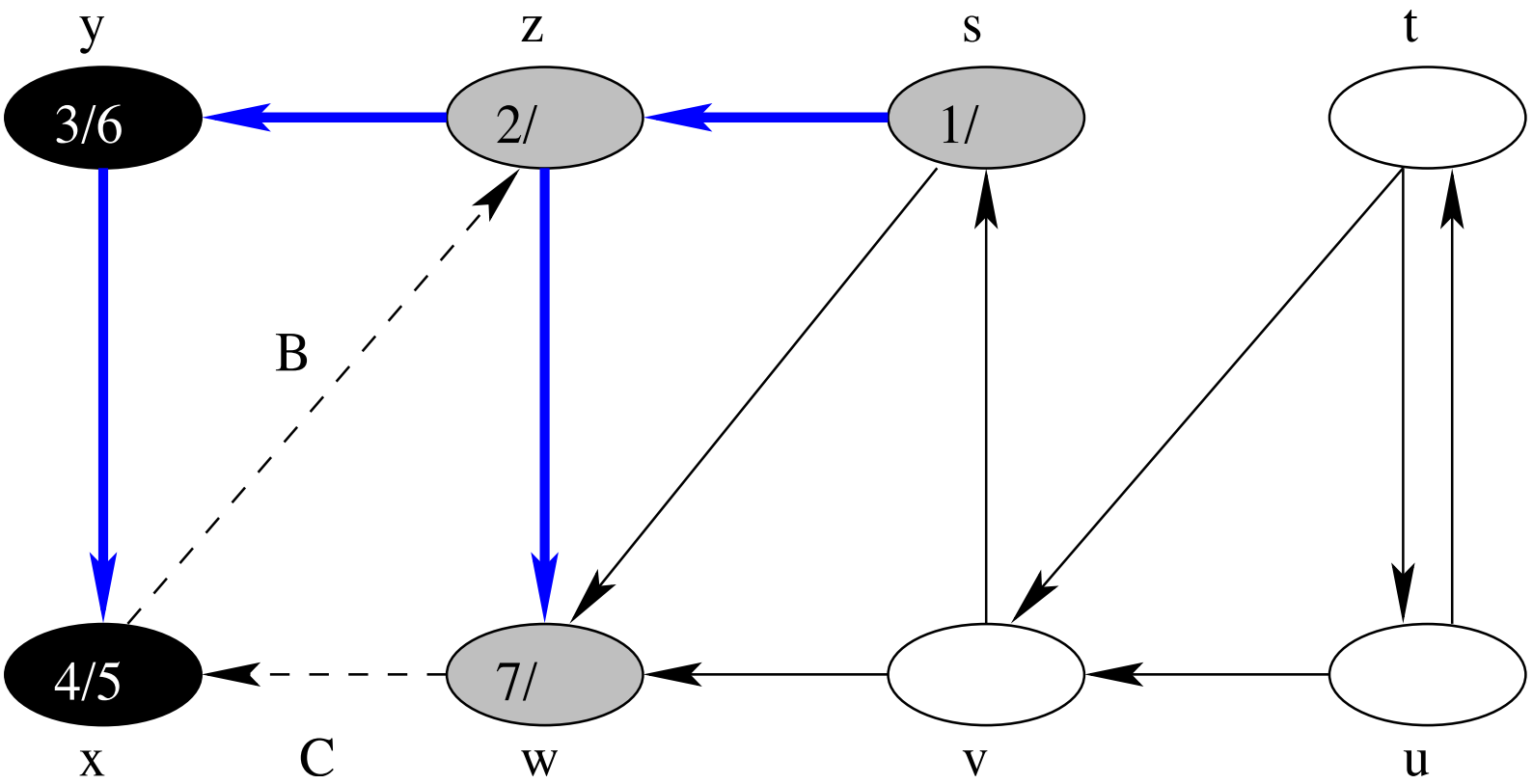
Exemplo



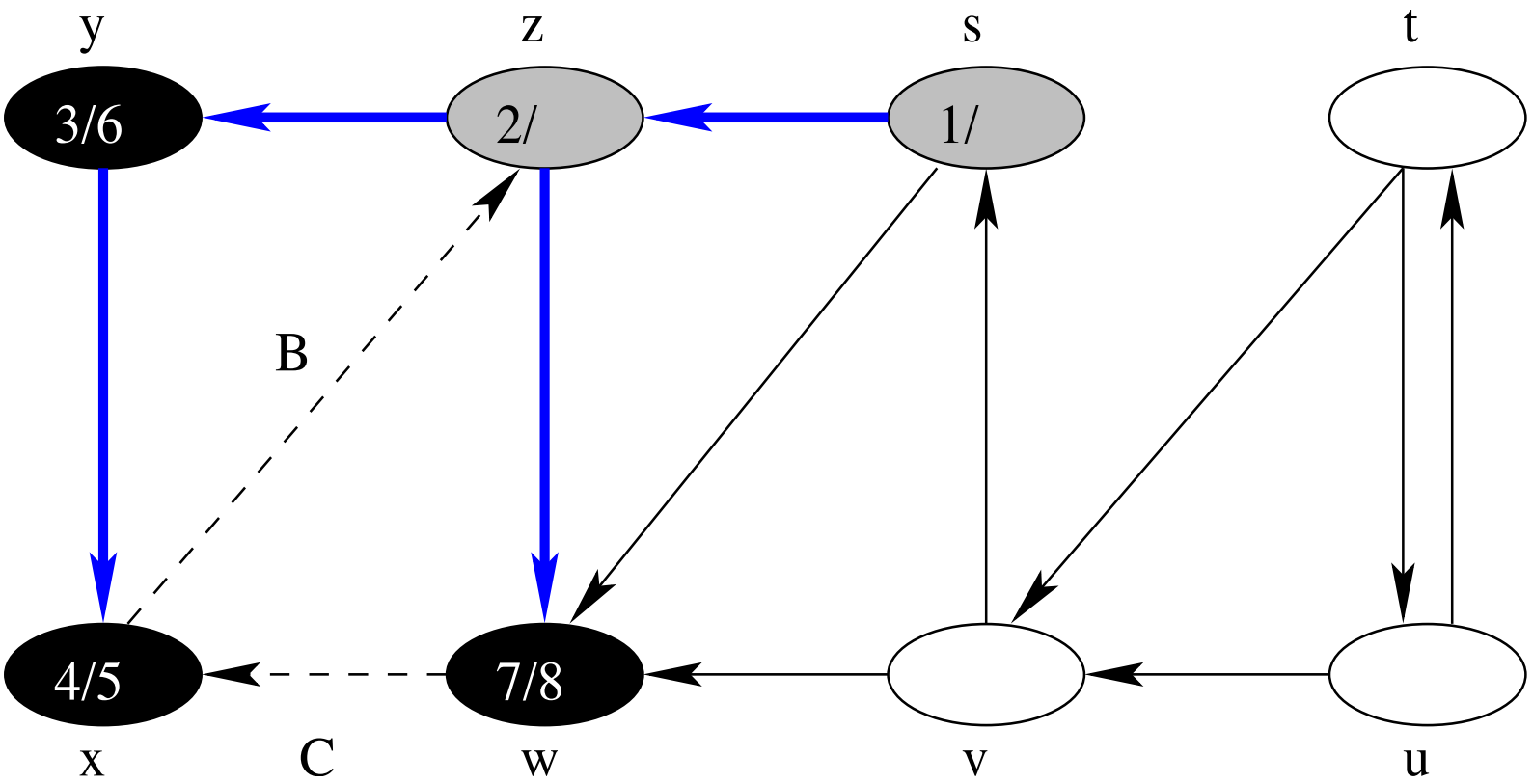
Exemplo



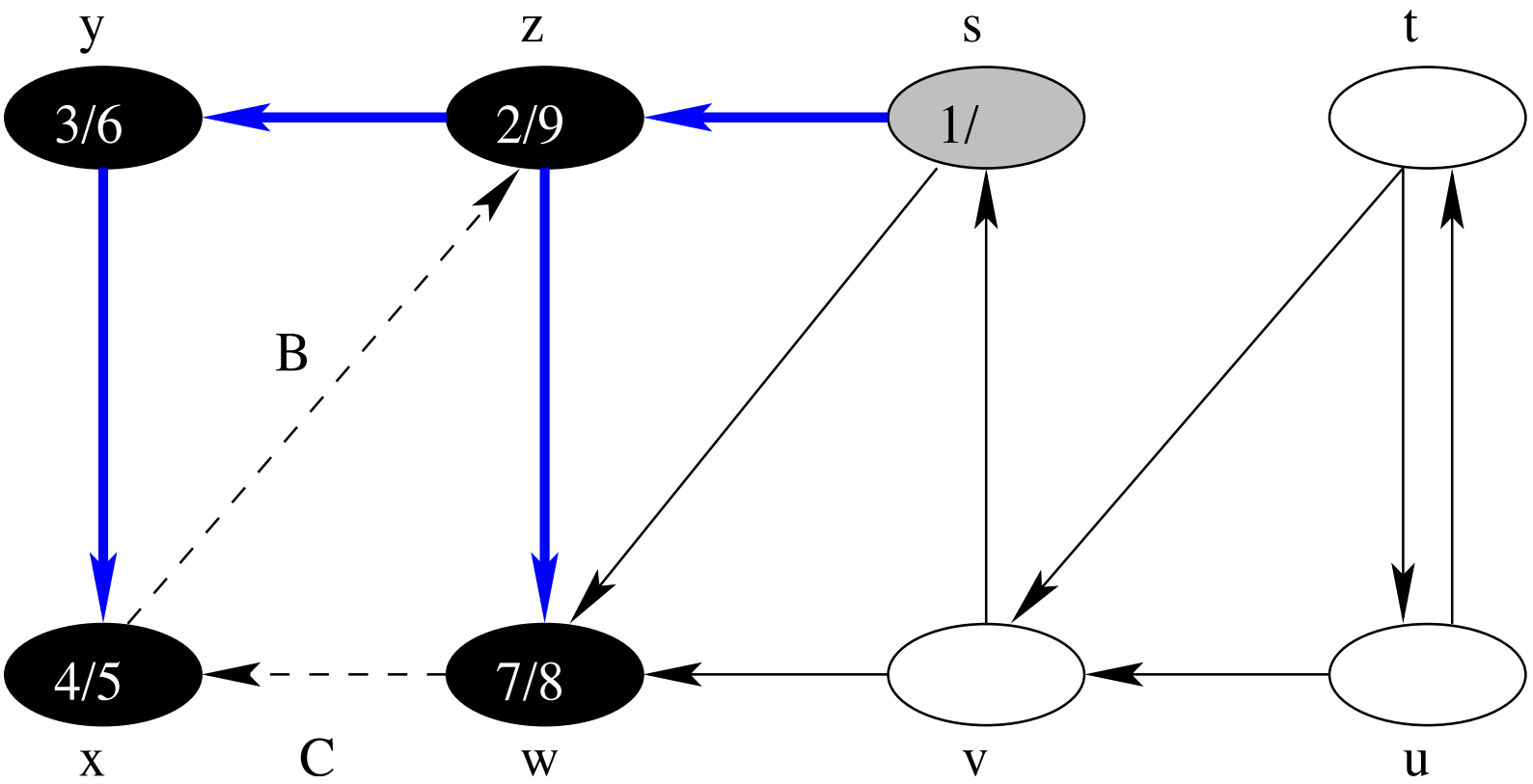
Exemplo



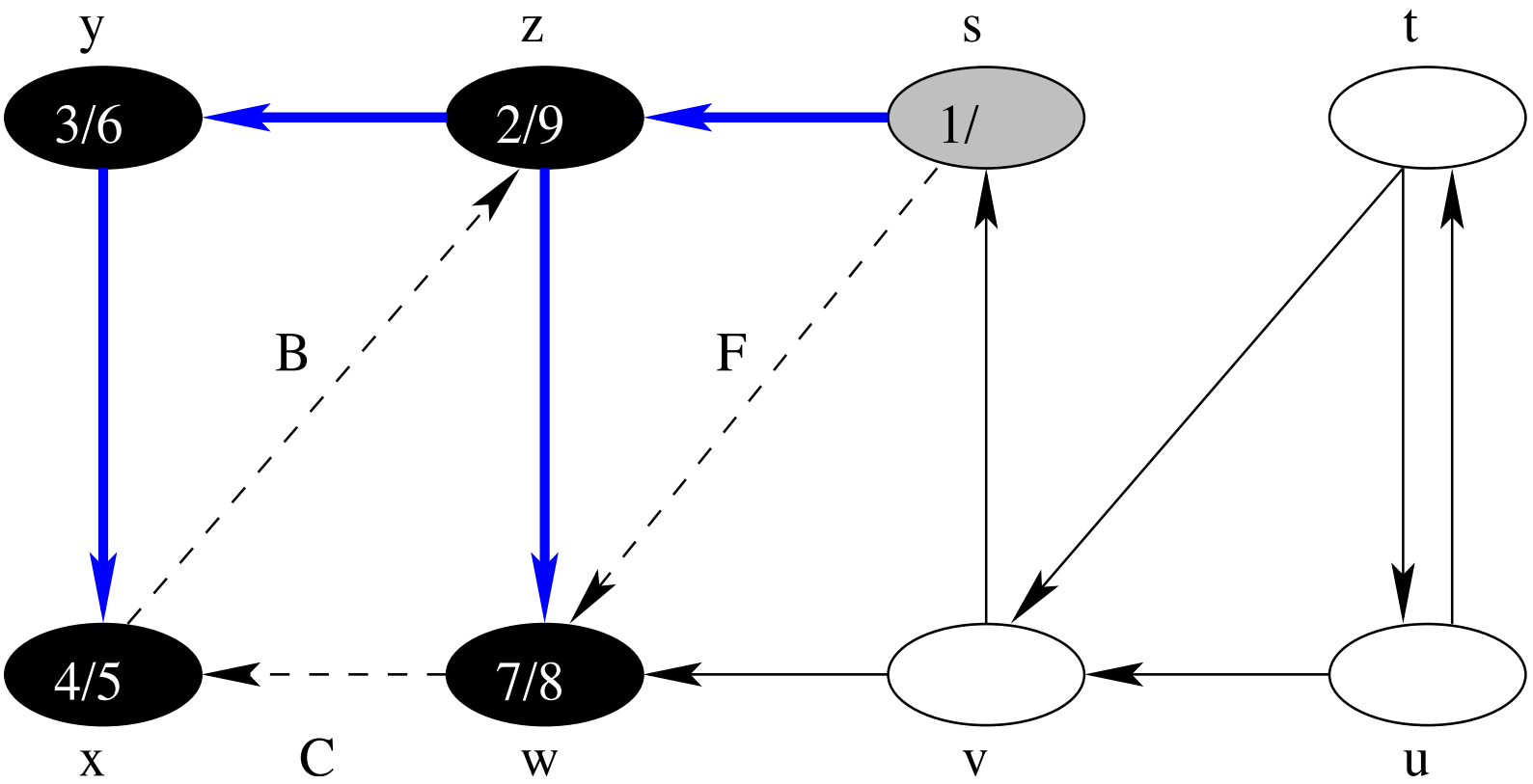
Exemplo



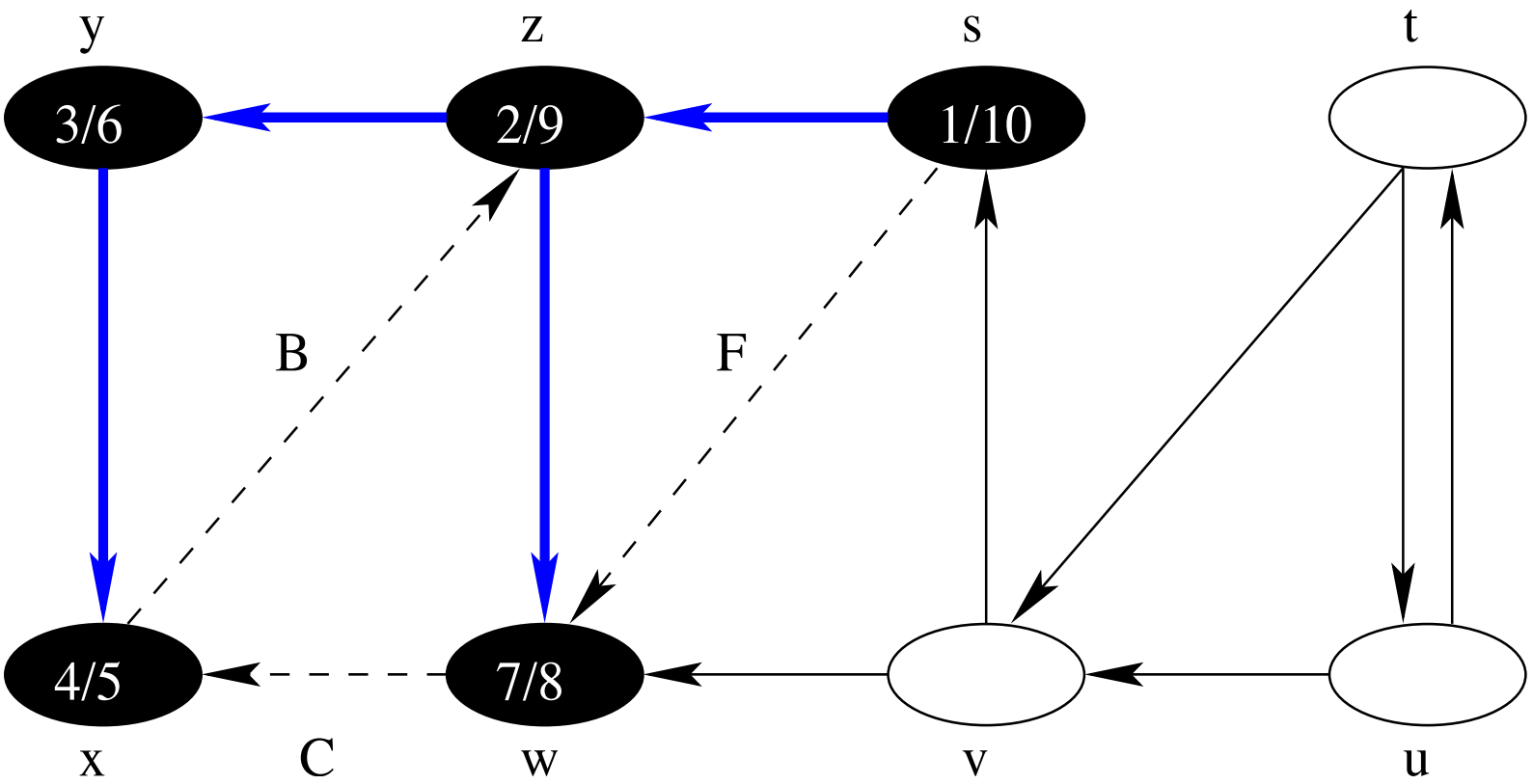
Exemplo



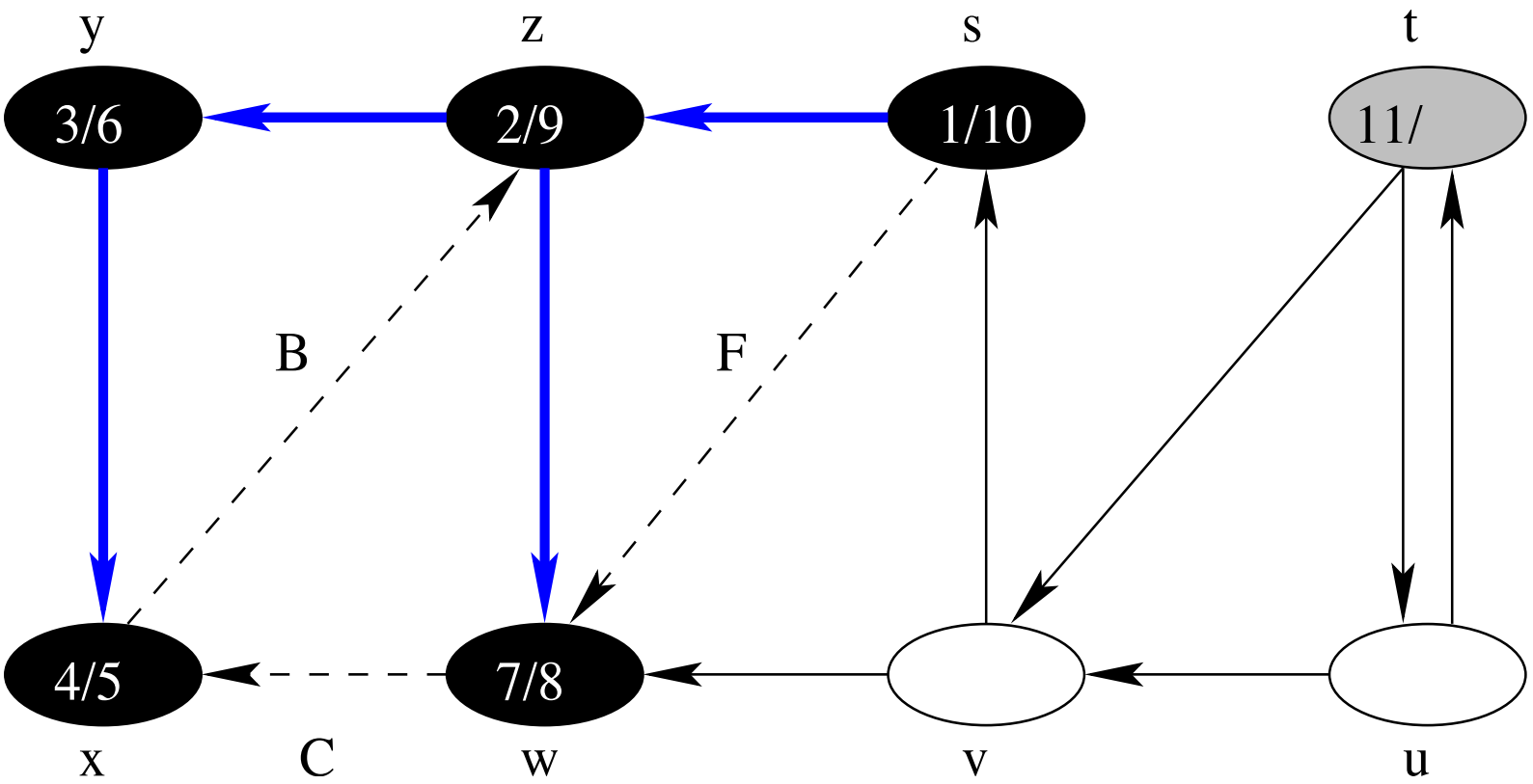
Exemplo



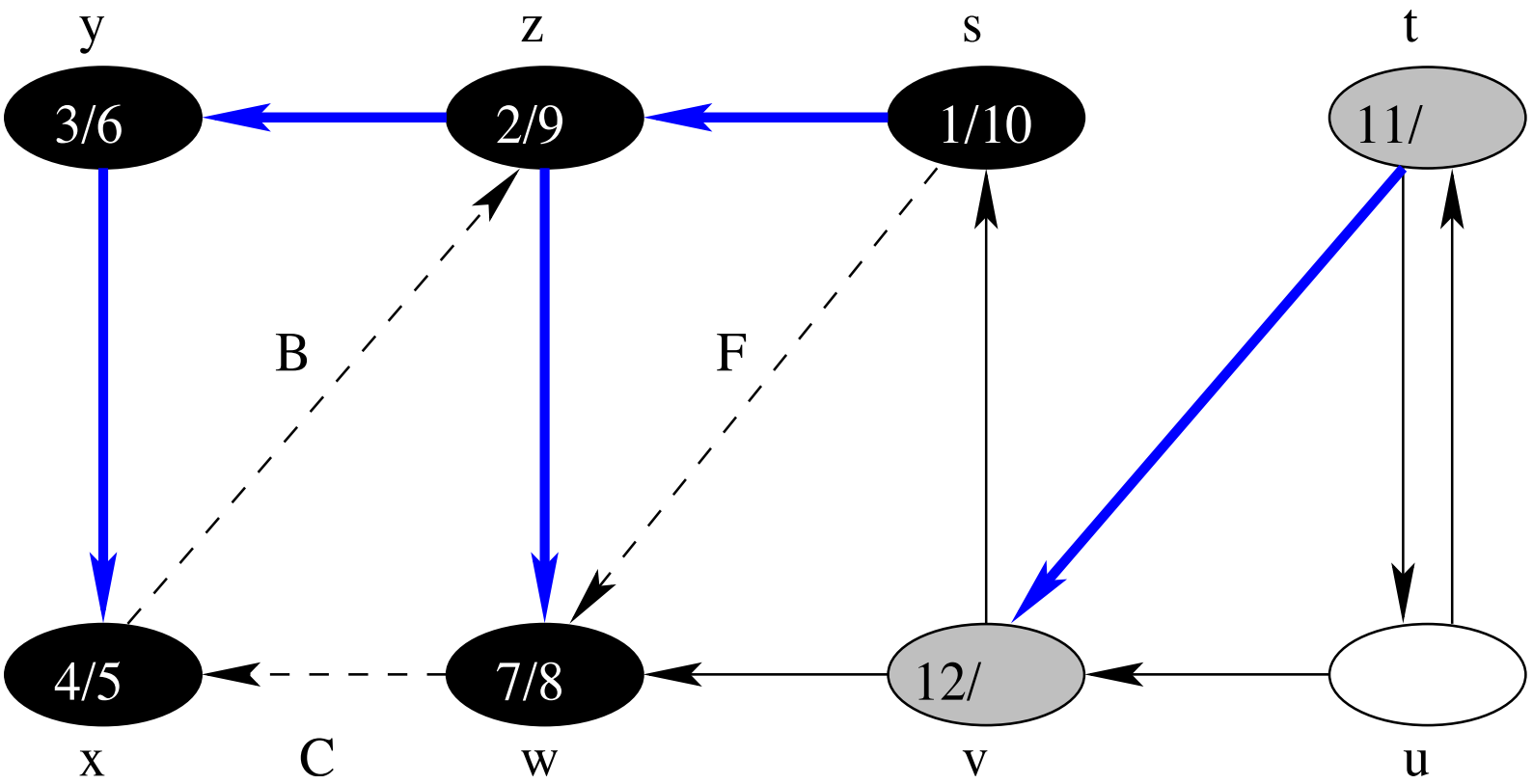
Exemplo



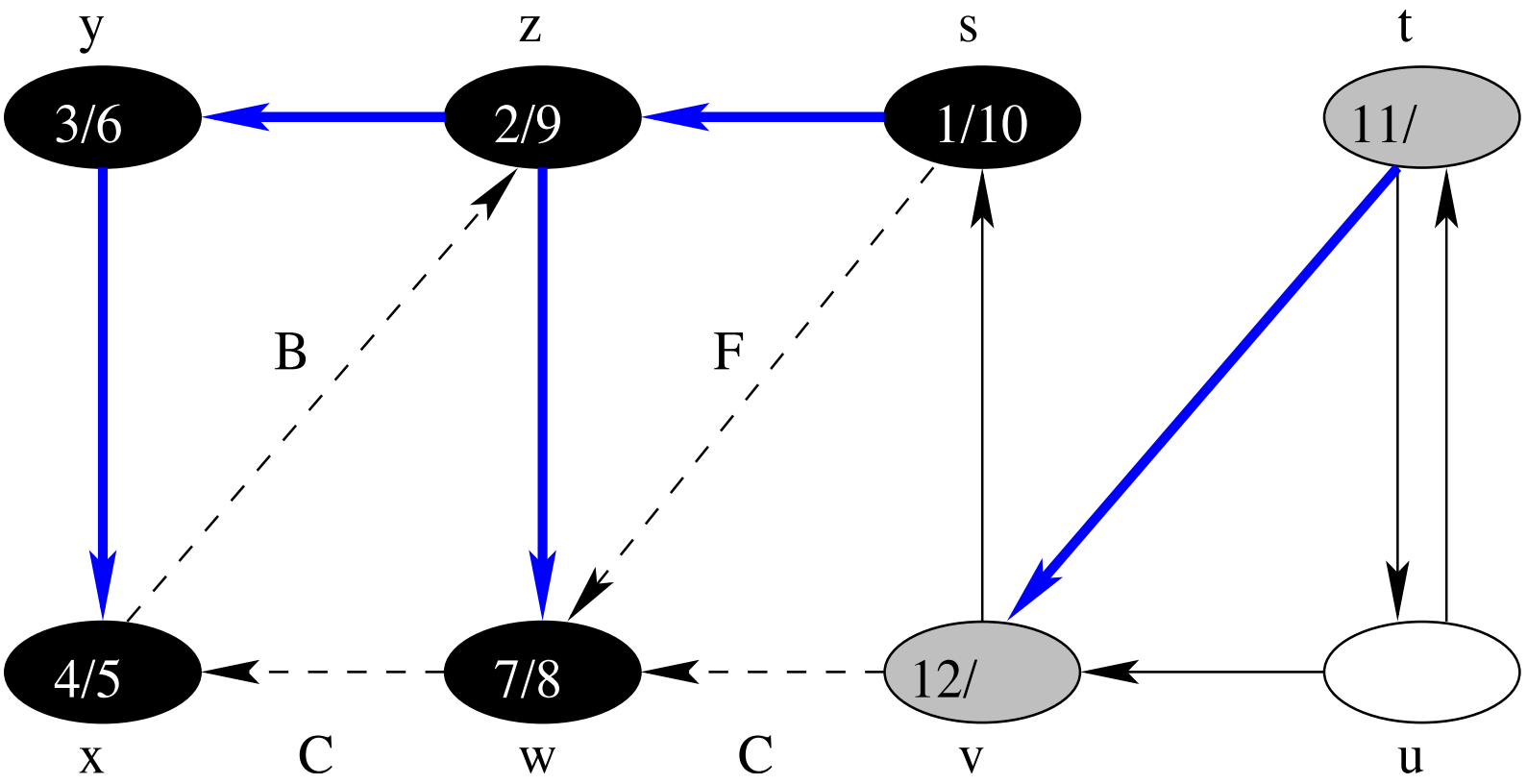
Exemplo



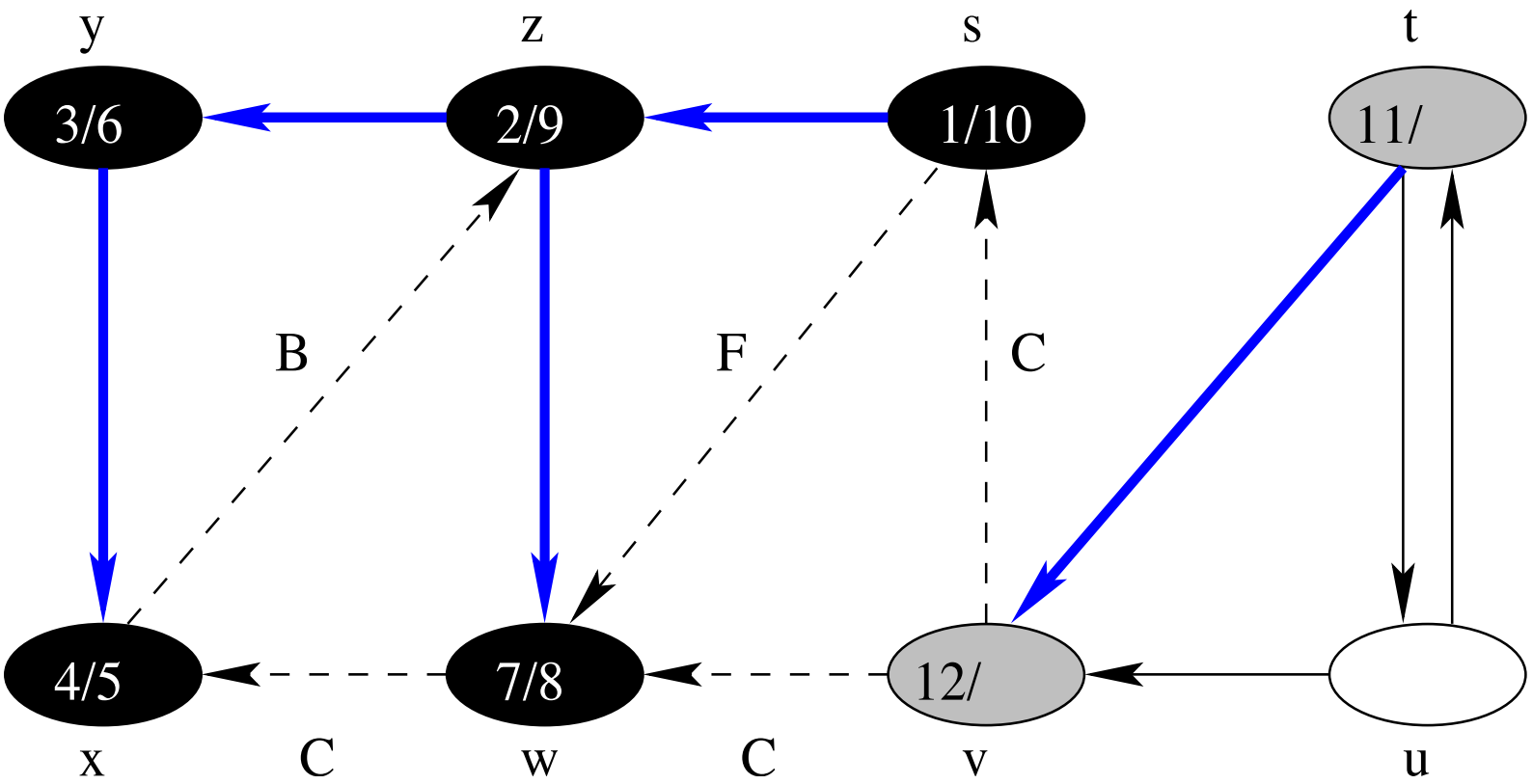
Exemplo



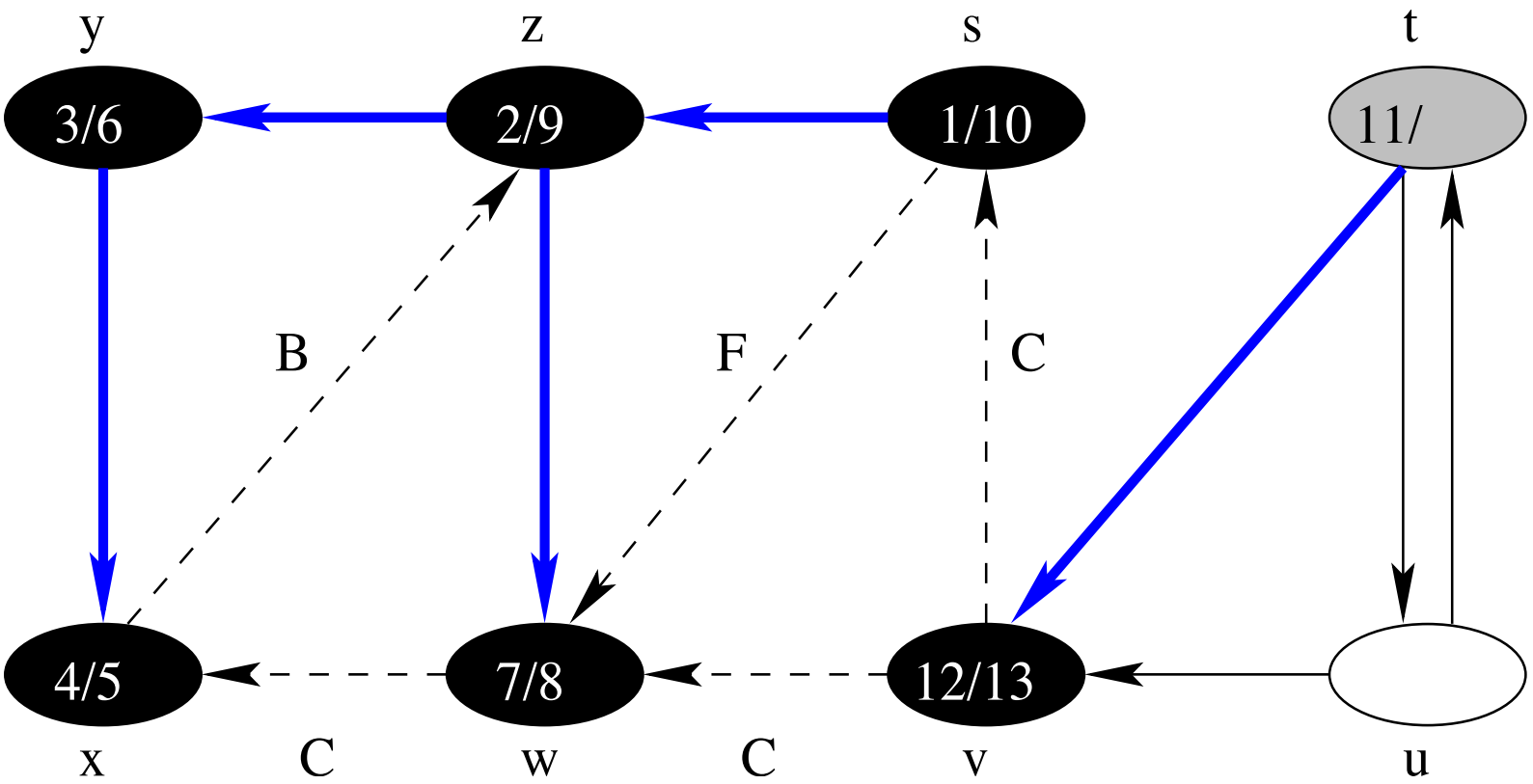
Exemplo



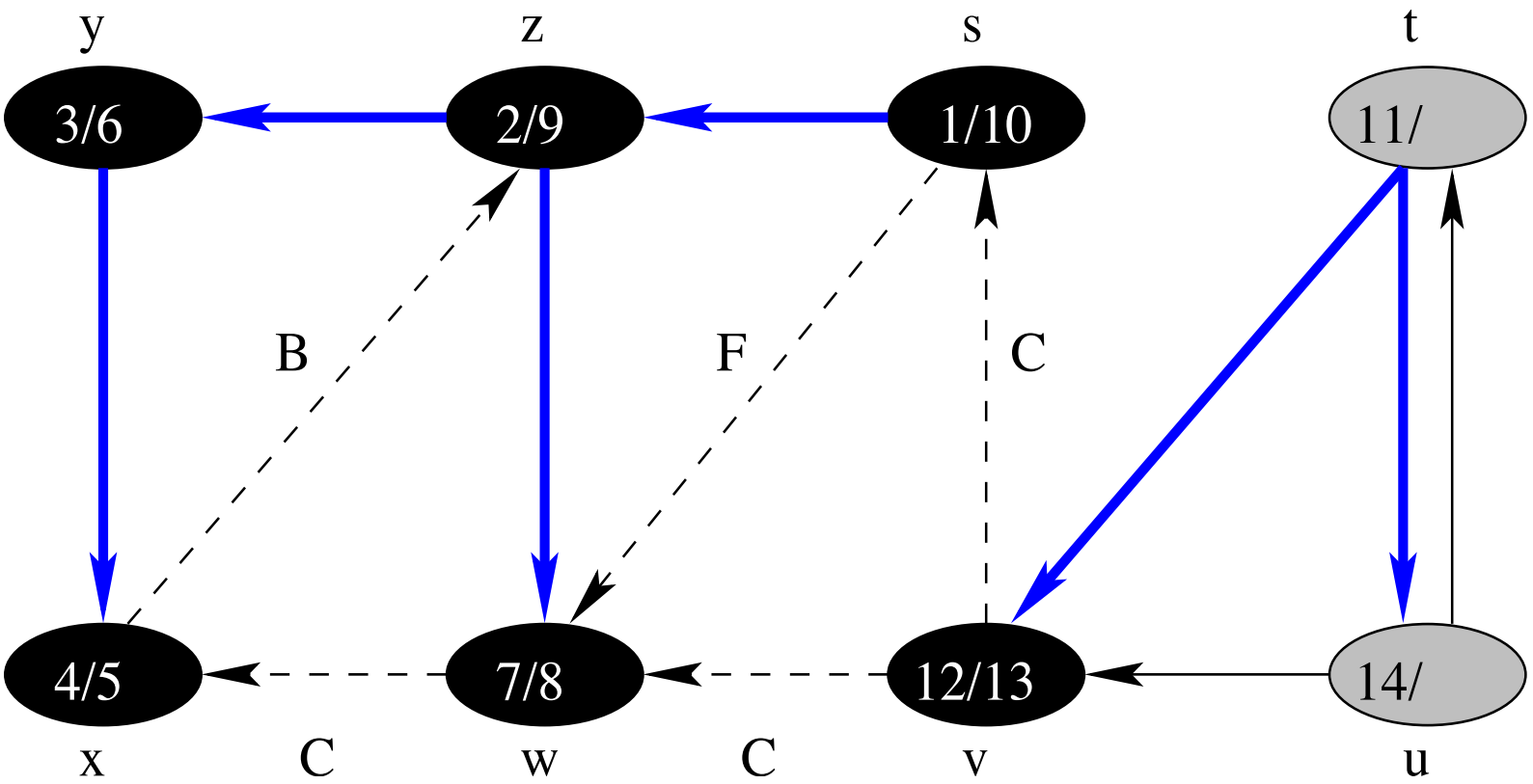
Exemplo



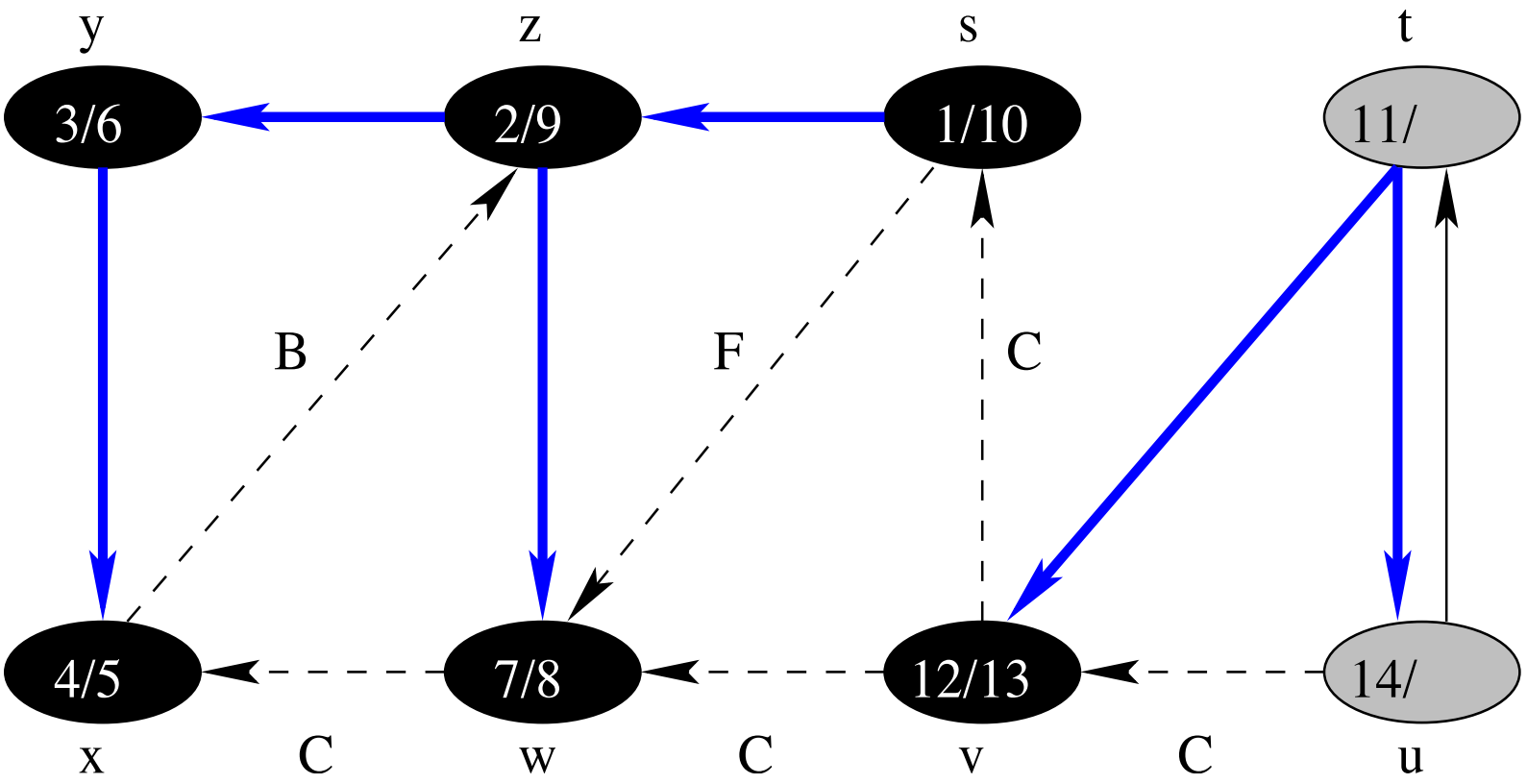
Exemplo



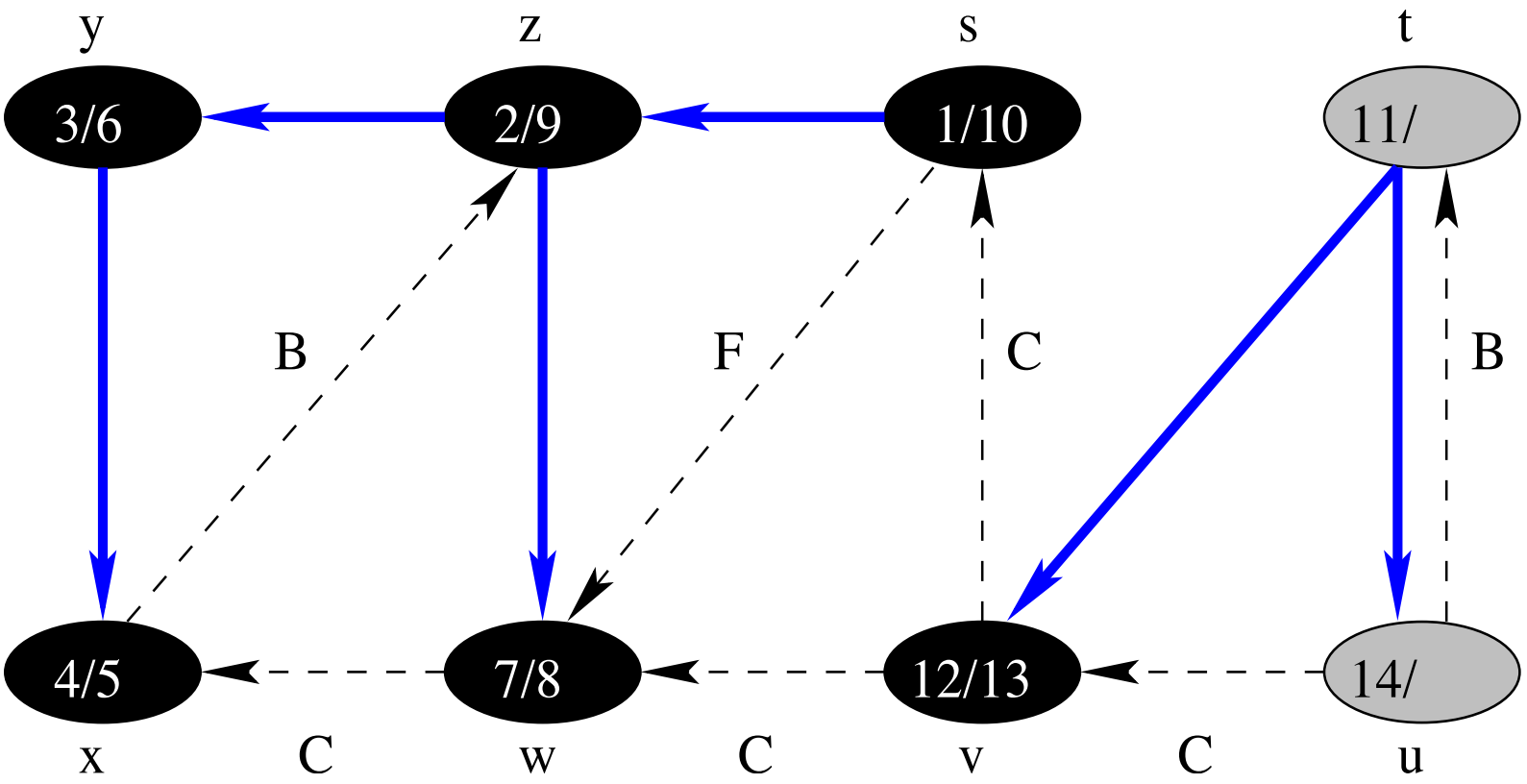
Exemplo



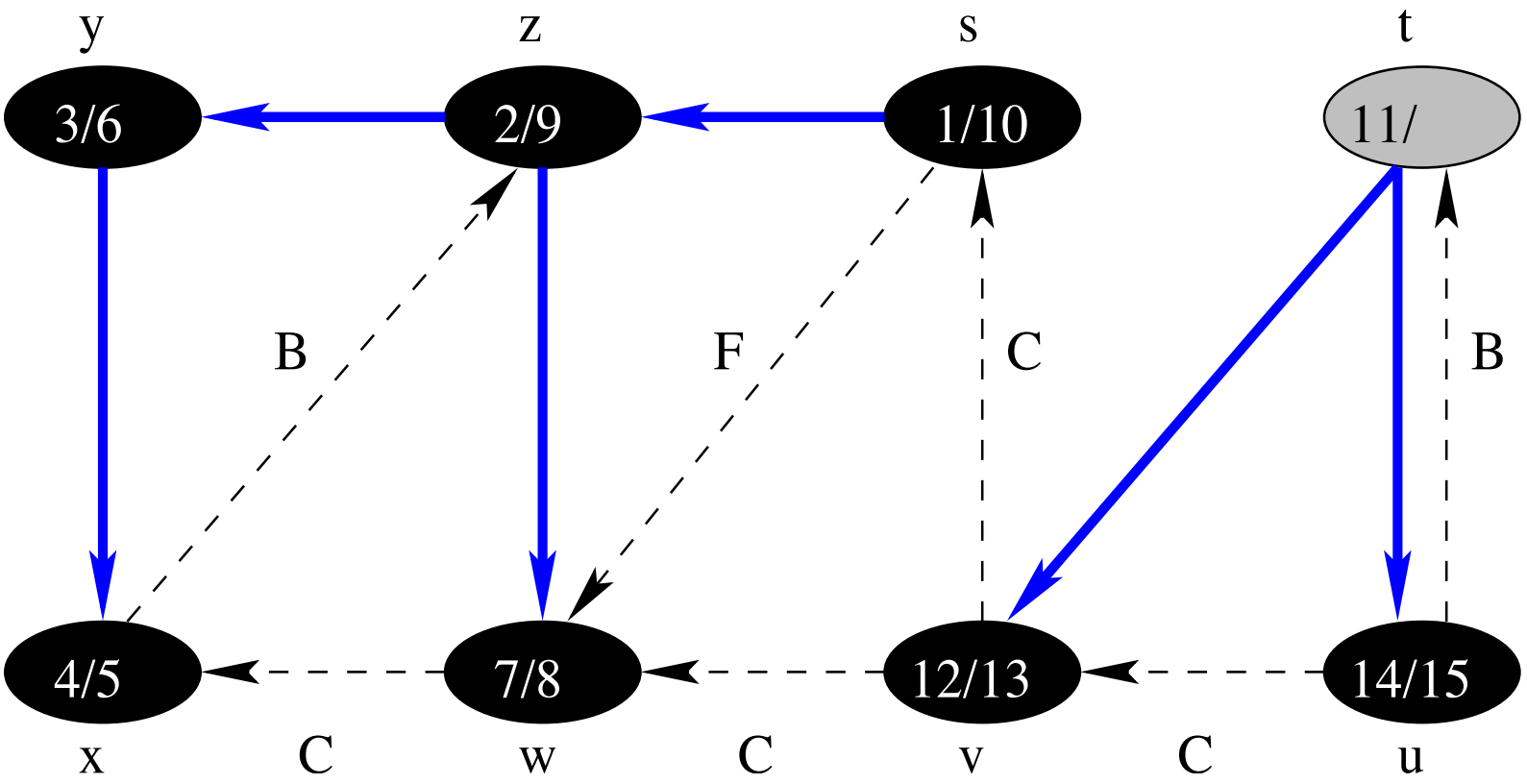
Exemplo



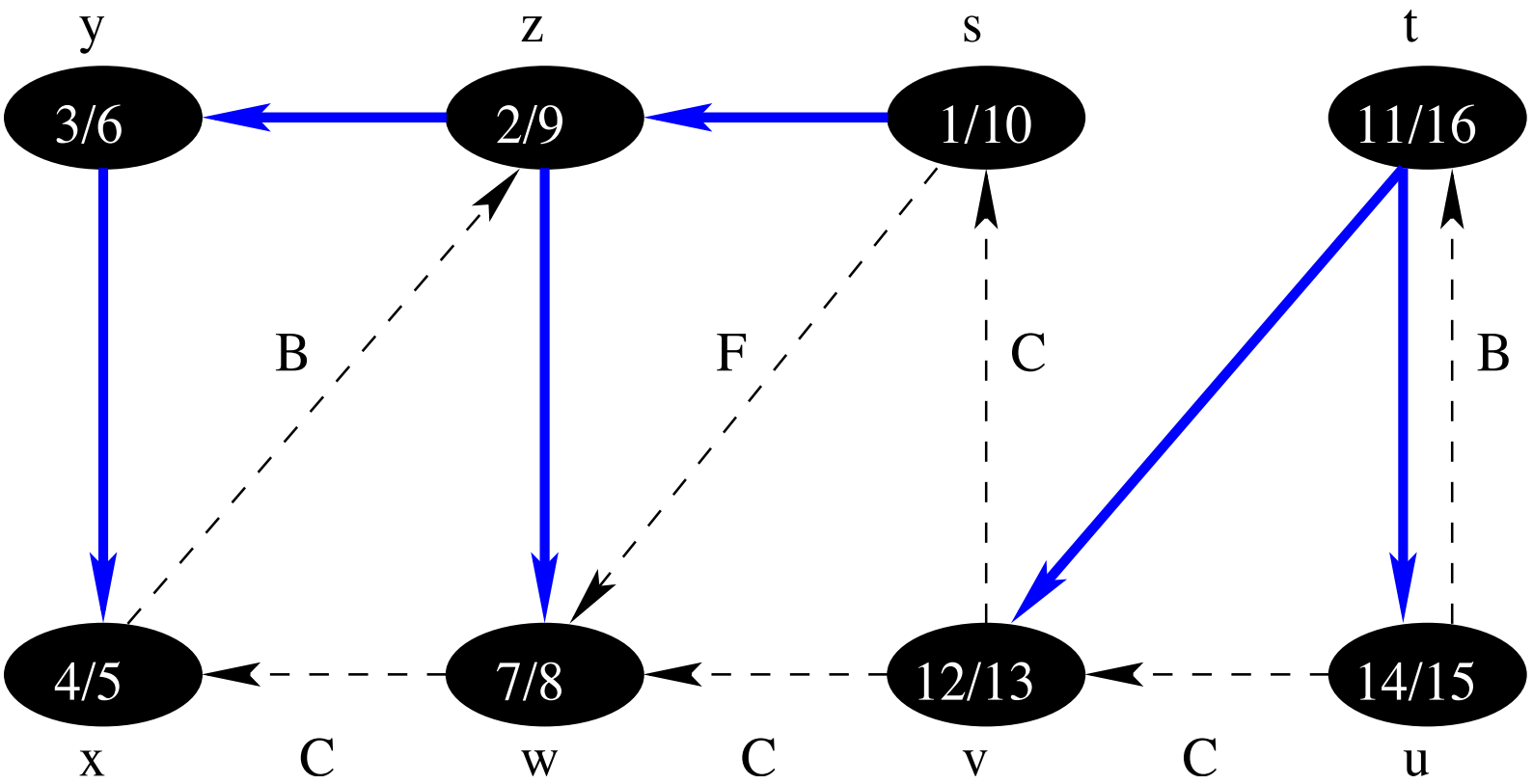
Exemplo



Exemplo



Exemplo



Rótulos versus cores

Para todo $x \in V[G]$ vale que $d[x] < f[x]$.

Além disso

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

Algoritmo DFS

Recebe um grafo G (na forma de listas de adjacências) e devolve
(i) os instantes $d[v], f[v]$ para cada $v \in V$ e
(ii) uma Floresta de Busca em Profundidade.

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

Algoritmo DFS-VISIT

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

DFS-VISIT(u)

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

Algoritmo DFS

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

Consumo de tempo

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

Algoritmo DFS-VISIT

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

Consumo de tempo

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

Complexidade de DFS

- DFS-VISIT(v) é executado exatamente uma vez para cada $v \in V$.
- Em uma execução de DFS-VISIT(v), o laço das linhas 4-7 é executado $|\mathrm{Adj}[u]|$ vezes.
  Assim, o custo total de todas as chamadas é

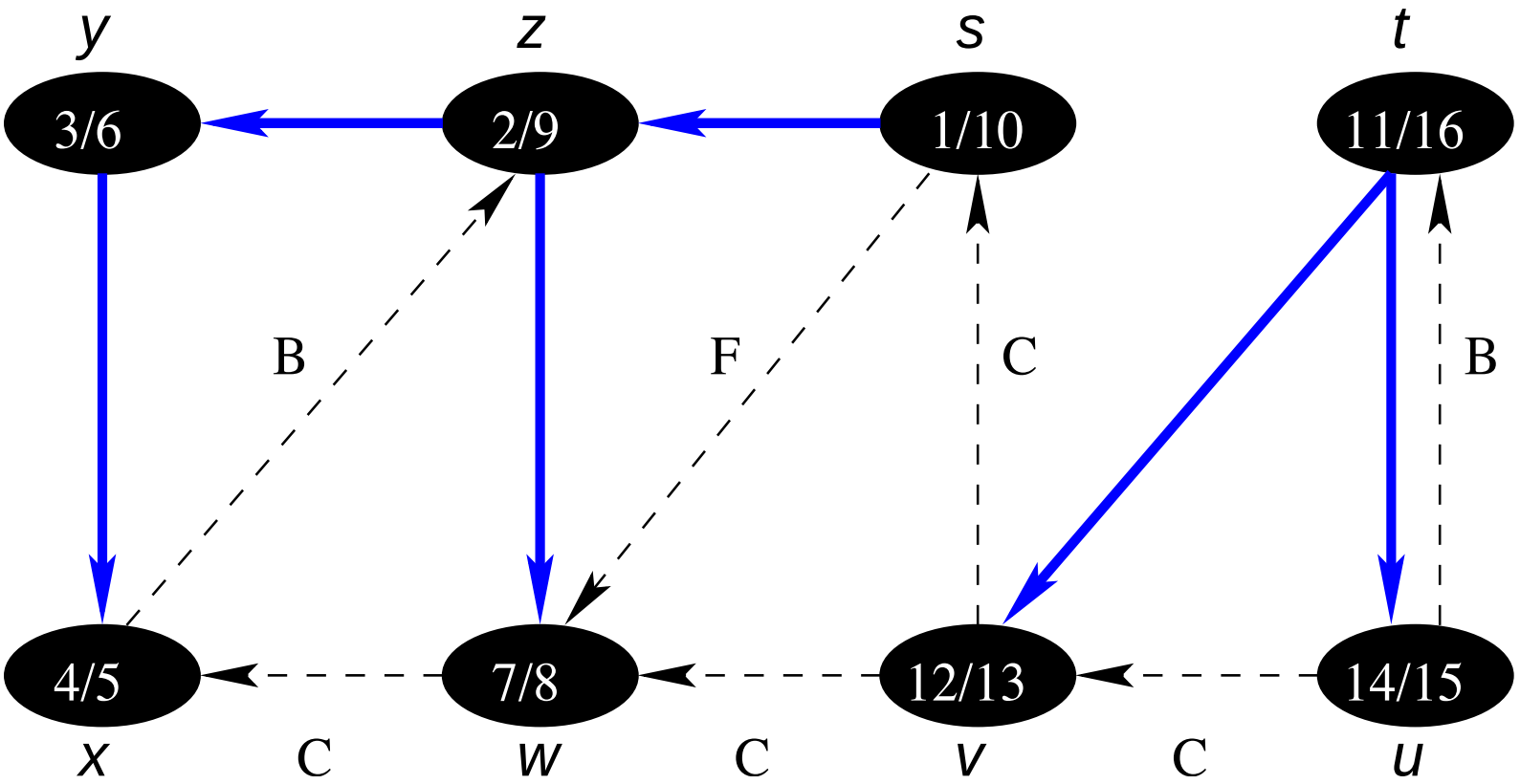
  $\sum_{v \in V} |\mathrm{Adj}(v)| = \Theta(E)$.

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

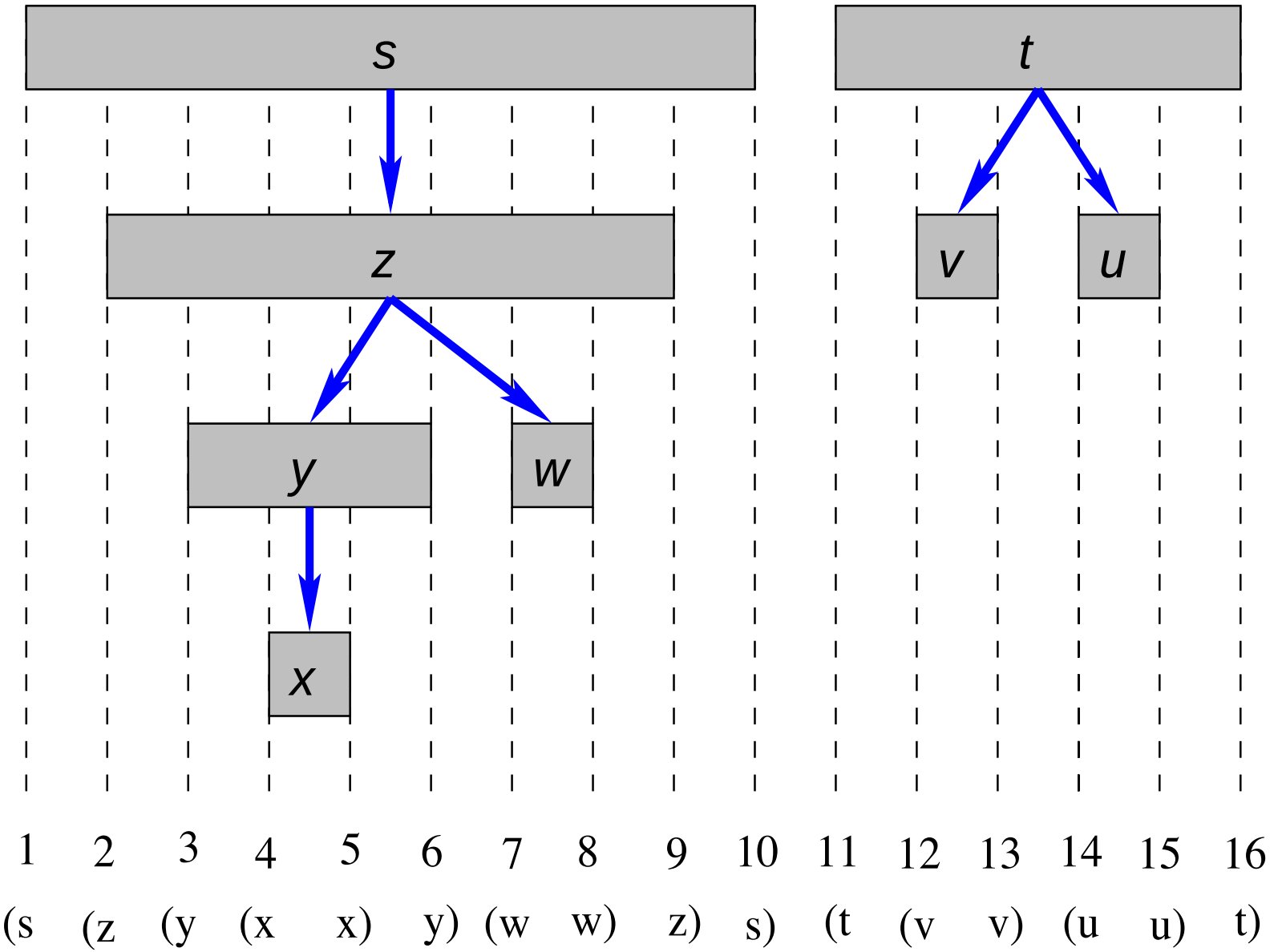
Estrutura de parênteses

- Os rótulos $d[x], f[x]$ têm propriedades muito úteis para serem usadas em outros algoritmos.
- Eles refletem a ordem em que a busca em profundidade foi executada.
- Eles fornecem informação de como é a “cara” (estrutura) do grafo.

Estrutura de parênteses



Estrutura de parênteses



Estrutura de parênteses

Teorema (Teorema dos Parênteses)

Em uma busca em profundidade sobre um grafo $G = (V, E)$, para quaisquer vértices u e v, ocorre exatamente uma das situações abaixo:

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

Estrutura de parênteses

Corolário. (Intervalos encaixantes para descendentes)

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

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

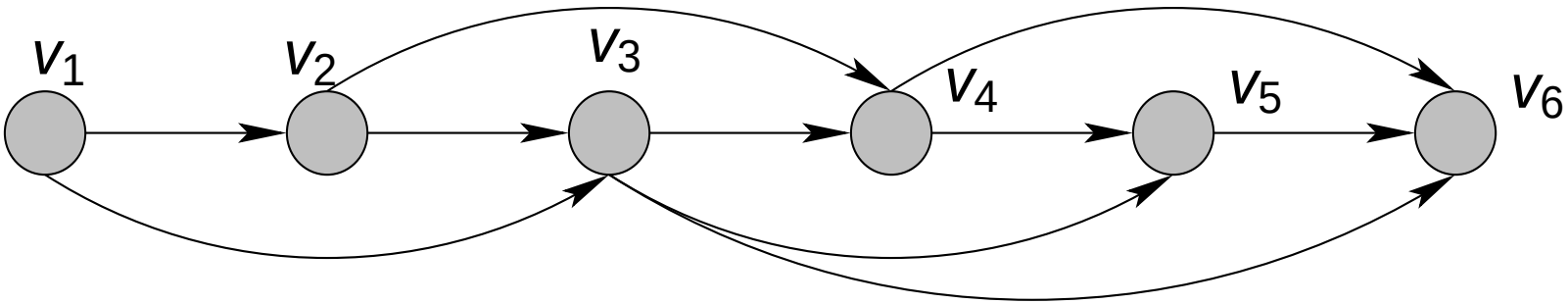
Ordenação Topológica

Ordenação Topológica

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

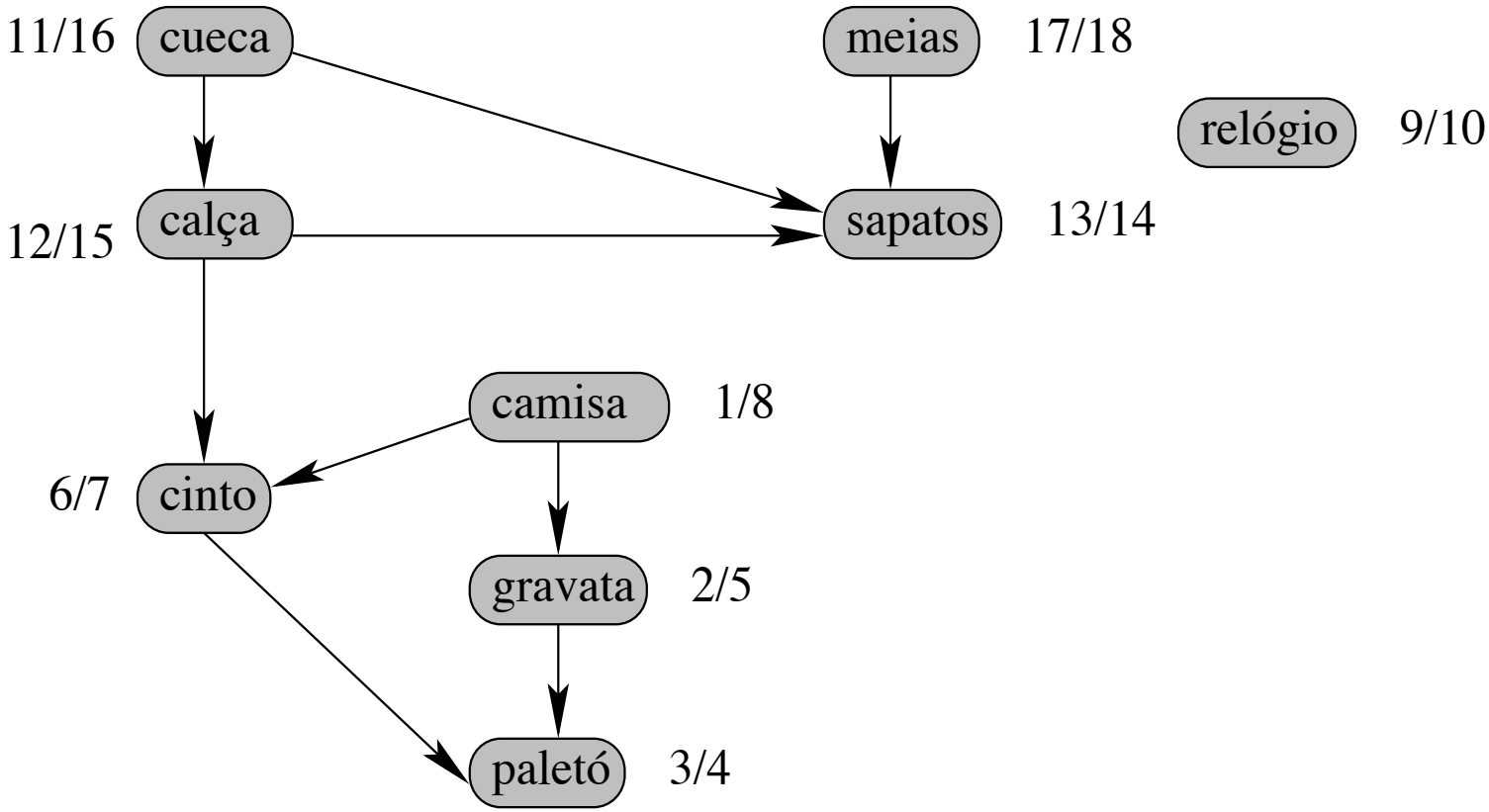
$$v_1 \quad v_2 \quad v_3 \quad \ldots \quad v_{n-2} \quad v_{n-1} \quad v_n$$

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

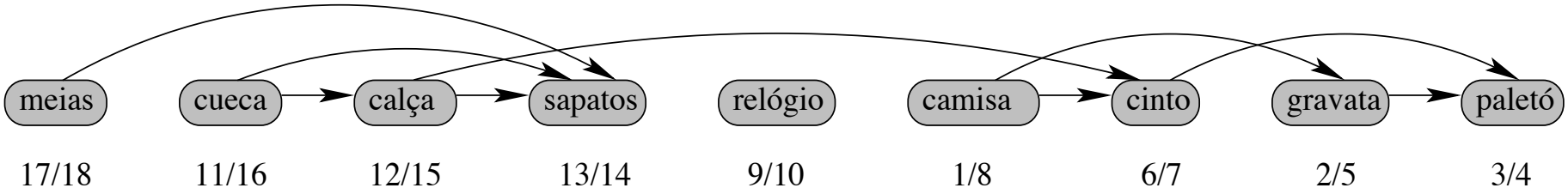


Ordenação Topológica

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

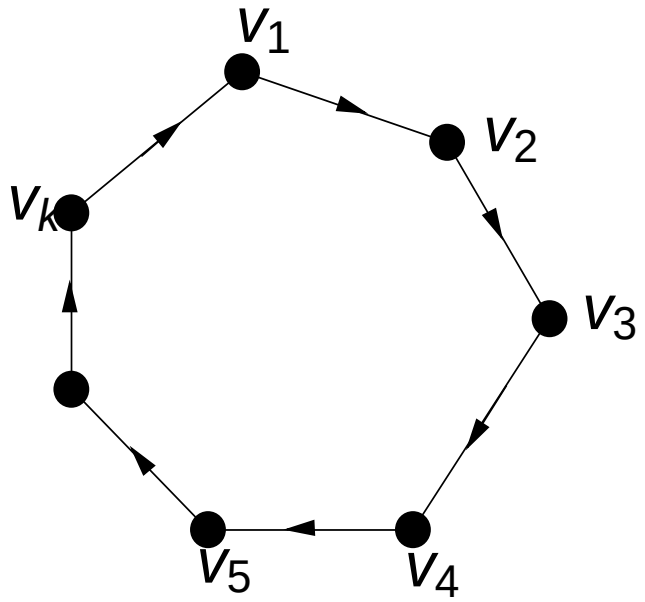


Ordenação Topológica



Ordenação Topológica

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



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

Grafo Orientado Acíclico

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

Prova.

Obviamente, se G possui uma ordenação topológica então G é acíclico.

Vamos mostrar a recíproca.

Definição

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

Grafo Orientado Acíclico

Lema. Todo grafo orientado acíclico possui uma fonte e um sorvedouro.

Baseado no resultado acima pode-se projetar um algoritmo para obter uma ordenação topológica de um grafo orientado acíclico G.

- Encontre uma fonte v_1 de G.
- Recursivamente encontre uma ordenação topológica $v_2, \ldots, v_n$ de $G - v_1$.
- Devolva $v_1, v_2, \ldots, v_n$.

Complexidade: $O(V^2)$ (análise grosseira)
Pode-se fazer melhor: O(V+E) (CLRS 22.4-5)

Ordenação Topológica

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

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

Outro modo de ver a linha 2 é:
Imprima os vértices em ordem decrescente de $f[v]$.

Complexidade de tempo

Conclusão:

A complexidade de tempo de TOPOLOGICAL-SORT é $O(V + E)$

Componentes fortemente conexos

Componentes fortemente conexos (CFC)

- Uma aplicação clássica de busca em profundidade:

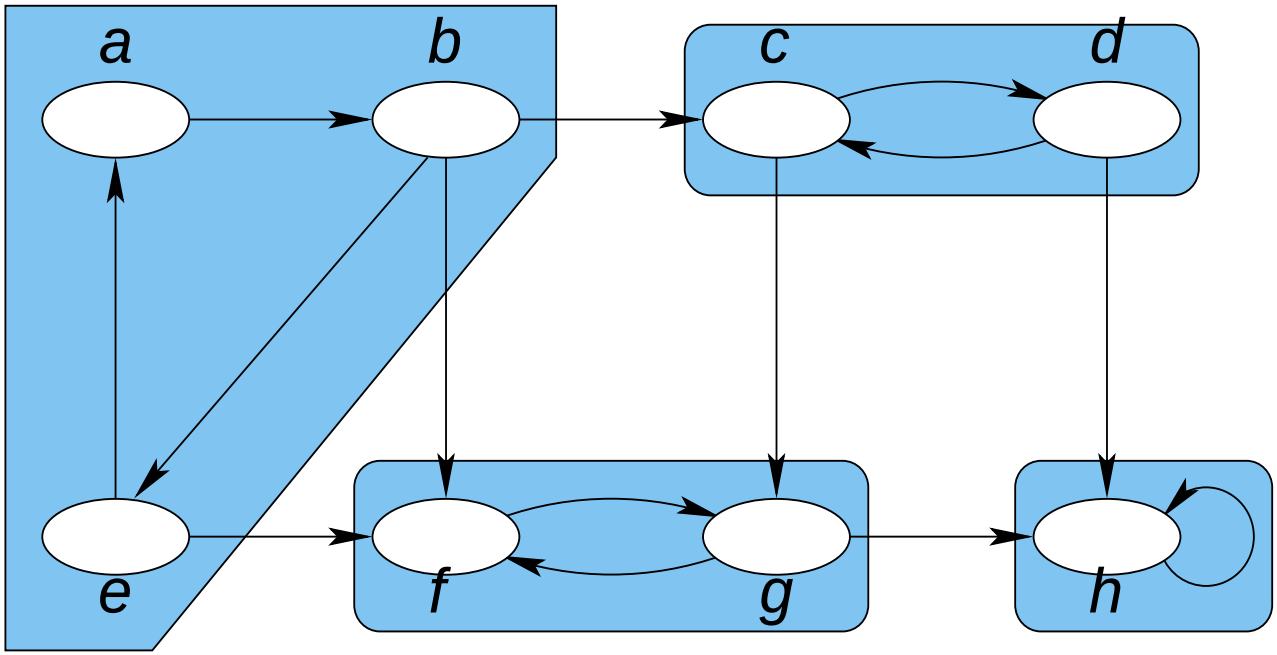
  decompor um grafo orientado em seus componentes fortemente conexos.
- Muitos algoritmos em grafos começam com tal decomposição.
- O algoritmo opera separadamente em cada componente fortemente conexo.
- As soluções são combinadas de alguma forma.

Componentes fortemente conexos

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

1. Para todo par de vértices u e v em C, existe um caminho (orientado) de u a v e vice-versa.
2. C é maximal com respeito à propriedade (1).

Componentes fortemente conexos



Um grafo orientado e seus componentes fortemente conexos.

Grafo transposto

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

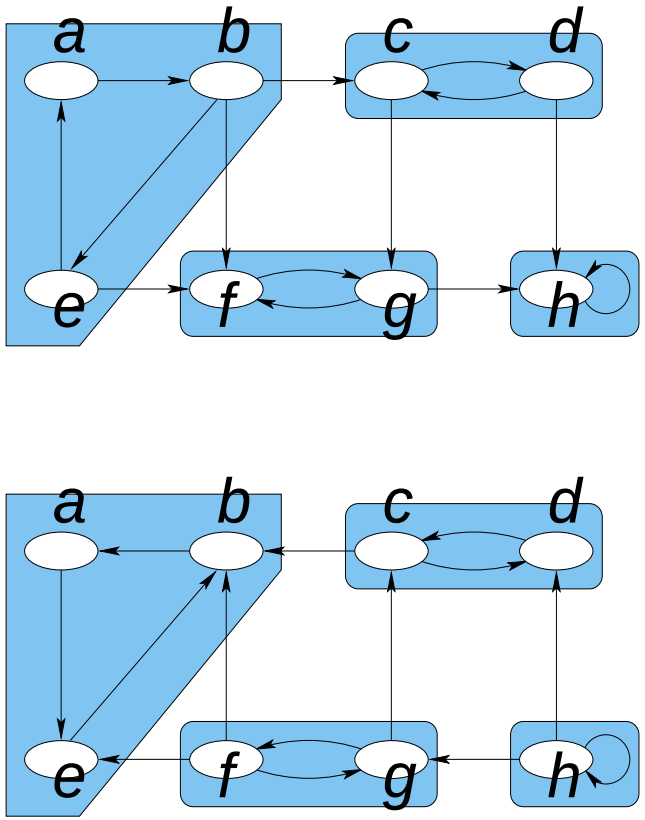
O grafo transposto de G é o grafo $G^T = (V^T, E^T)$ tal que

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

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

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

Grafo transposto



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

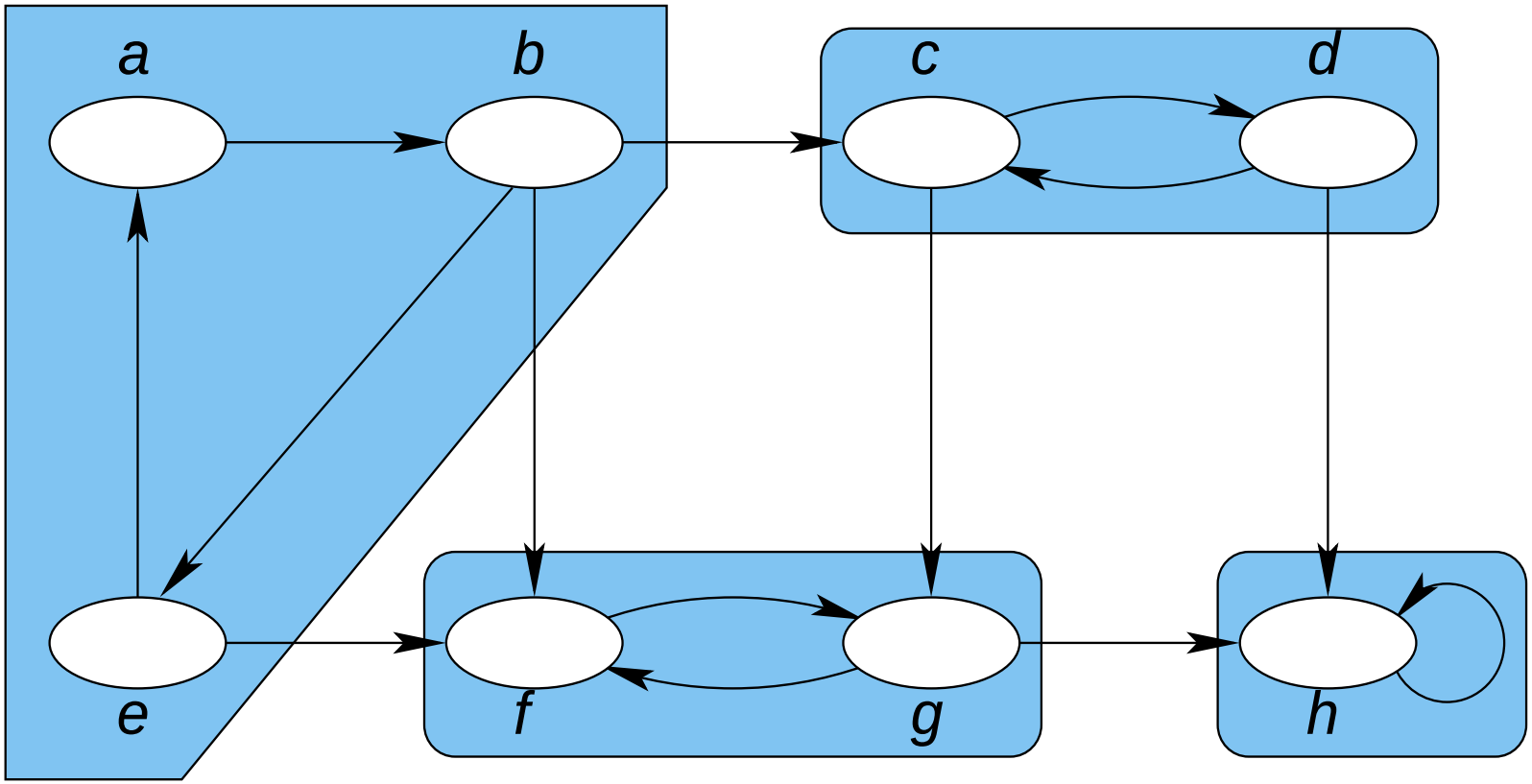
Algoritmo

COMPONENTES-FORTEMENTE-CONEXOS(G)

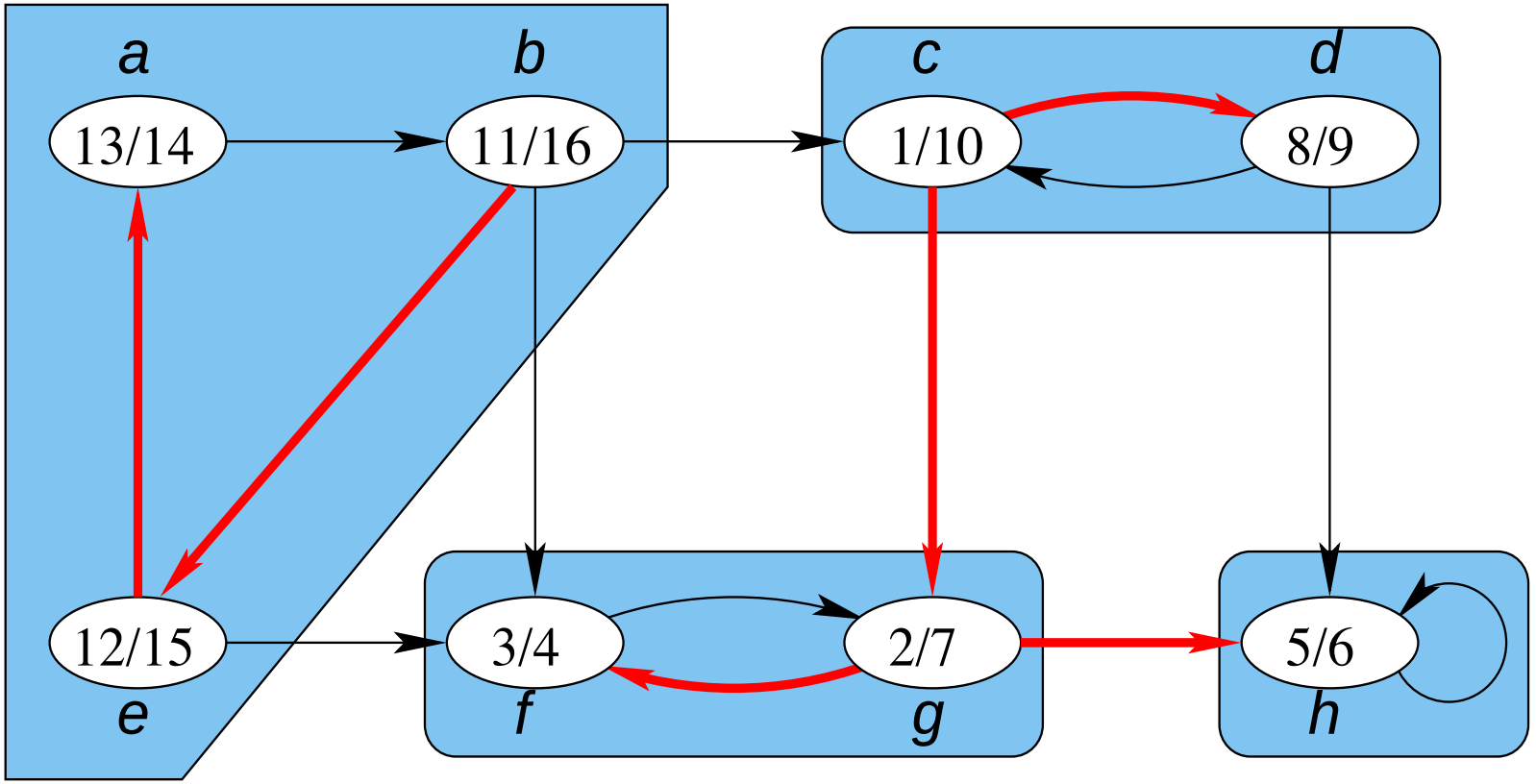
1. Execute DFS(G) para obter $f[v]$ para $v \in V$.
2. Execute DFS(G^T) considerando os vértices em ordem decrescente de $f[v]$.
3. Devolva os conjuntos de vértices de cada árvore da Floresta de Busca em Profundidade obtida.

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

Exemplo CLRS

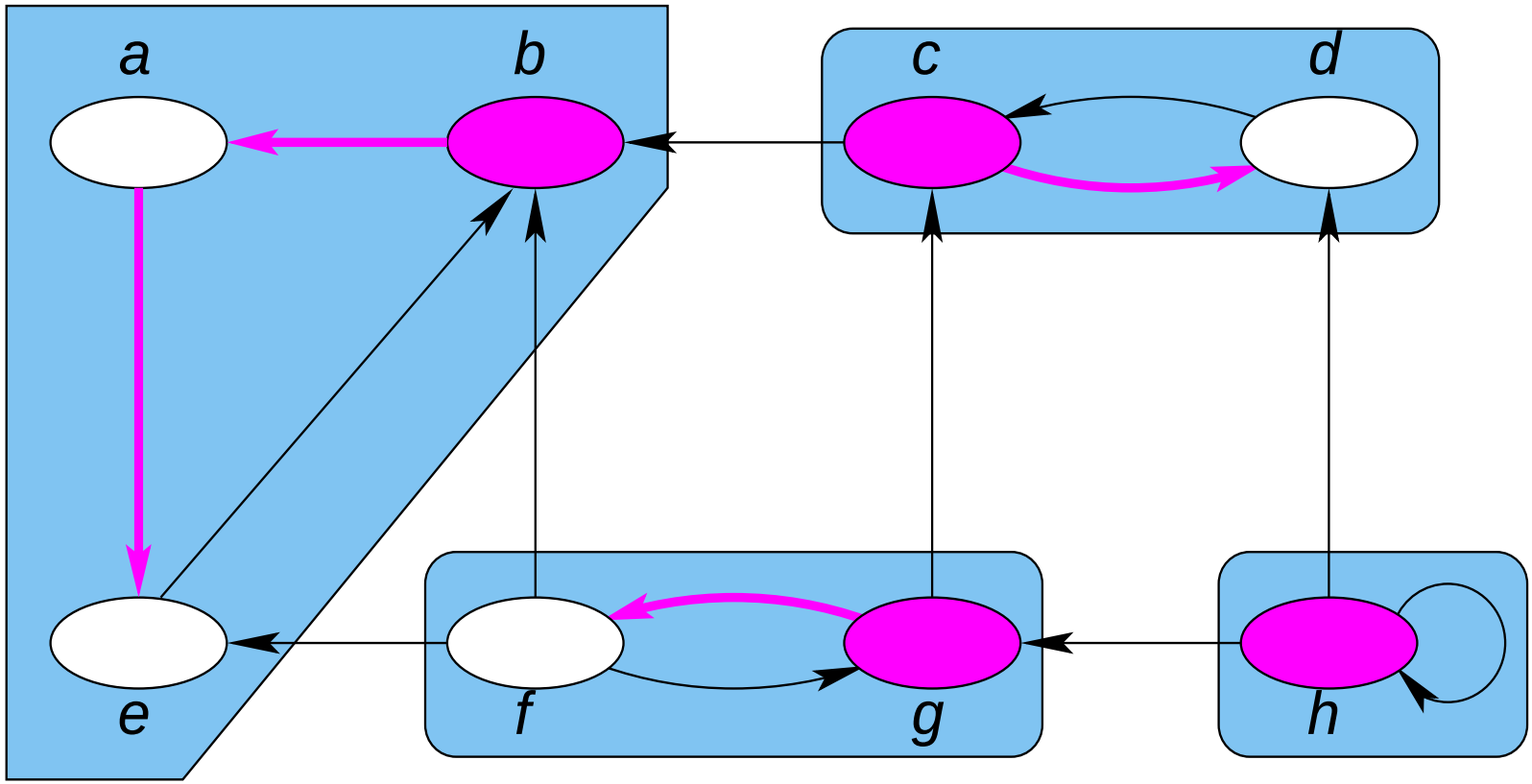


Exemplo CLRS



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

Exemplo CLRS

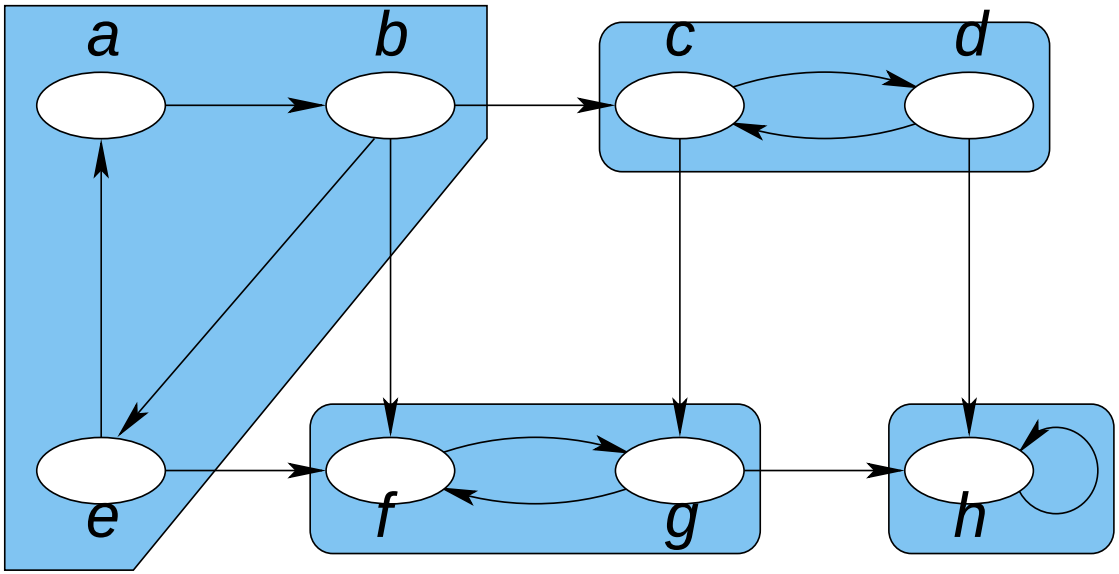


2 Execute DFS(G^T) considerando os vértices em ordem decrescente de $f[v]$.

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

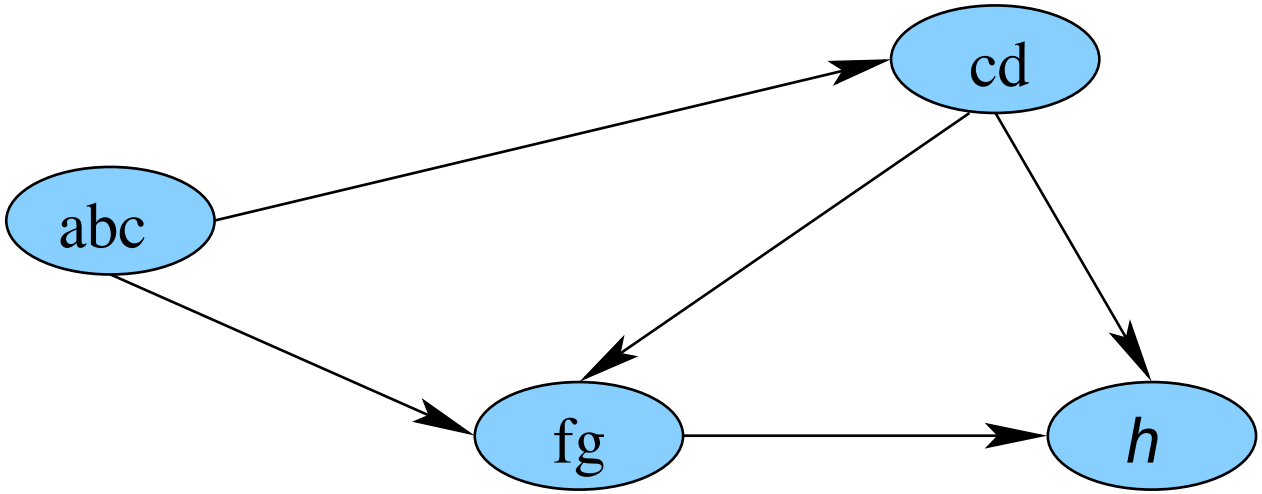
Grafo Componente

A idéia por trás de COMPONENTES-FORTEMENTE-CONEXOS segue de uma propriedade do grafo componente G^{CFC} obtido a partir de G **contraindo-se** seus componentes fortemente conexos.



Grafo Componente

A idéia por trás de COMPONENTES-FORTEMENTE-CONEXOS segue de uma propriedade do grafo componente G^{CFC} obtido a partir de G **contraindo-se** seus componentes fortemente conexos.



G^{CFC} é acíclico.

Os componentes fortementes conexos são visitados em ordem topológica com relação a G^{CFC}!

Algoritmos gulosos

Algoritmos Gulosos: Conceitos Básicos

- Tipicamente algoritmos gulosos são utilizados para resolver problemas de **otimização**.
- Uma característica comum dos problemas onde se aplicam algoritmos gulosos é a existência **subestrutura ótima**, semelhante à programação dinâmica!
- Programação dinâmica: tipicamente os subproblemas são resolvidos à otimalidade **antes** de se proceder à escolha de um elemento que irá compor a solução ótima.
- Algoritmo Guloso: primeiramente é feita a escolha de um elemento que irá compor a solução ótima e só **depois** um subproblema é resolvido.

Algoritmos Gulosos: Conceitos Básicos

- Um algoritmo guloso sempre faz a escolha que parece ser a *"melhor"* a cada iteraçãousando um **critério guloso**.
  É uma decisão **localmente** ótima.
- Propriedade da **escolha gulosa**: garante que a cada iteração é tomada uma decisão que irá levar a um ótimo global.
- Em um algoritmo guloso uma escolha que foi feita **nunca é revista**, ou seja, não há qualquer tipo de *backtracking*.
- Em geral é fácil projetar ou descrever um algoritmo guloso.
  O **difícil** é provar que ele funciona!

Árvore Geradora Mínima

Árvore Geradora Mínima

- Suponha que queremos resolver o seguinte problema: dado um conjunto de computadores, onde cada par de computadores pode ser ligado usando uma quantidade de fibra ótica, encontrar uma rede interconectando-os que use a menor quantidade de fibra ótica possível.

- Este problema pode ser modelado por um problema em grafos não orientados ponderados onde os vértices representam os computadores, as arestas representam as conexões que podem ser construídas e o peso/custo de uma aresta representa a quantidade de fibra ótica necessária.

Árvore Geradora Mínima

- Nessa modelagem, o problema que queremos resolver é encontrar um **subgrafo gerador** (que contém todos os vértices do grafo original), **conexo** (para garantir a interligação entre todos os computadores) e cuja soma dos custos de suas arestas seja a menor possível.
- Obviamente, o problema só tem solução se o grafo for conexo. Daqui pra frente vamos supor que o grafo de entrada é conexo.
- Além disso, o sugrafo gerador procurado é sempre uma árvore (supondo que os pesos s ão positivos).

Árvore Geradora Mínima

Problema da Árvore Geradora Mínima

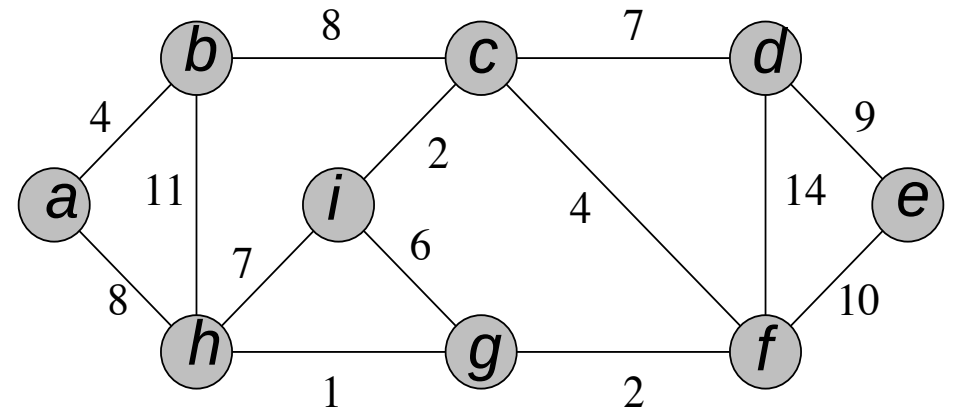
Entrada: grafo conexo $G = (V, E)$ com pesos $w(u, v)$ para cada aresta (u, v).

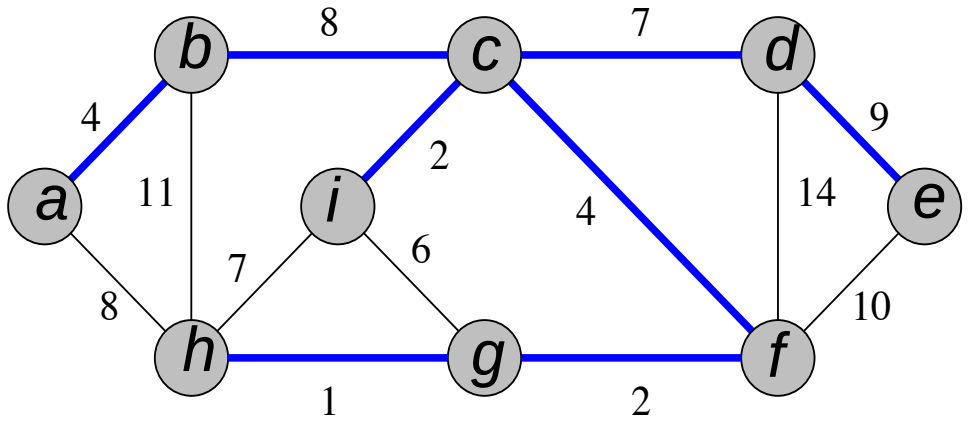
Saída: subgrafo gerador conexo T de G cujo peso total

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

seja o menor possível.

Exemplo


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
4
7
6
8
10
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
4
7
6
8
10
1
2

Árvore Geradora Mínima

- Veremos dois algoritmos para resolver o problema:
  - algoritmo de Prim
  - algoritmo de Kruskal
- Ambos algoritmos usam estratégia gulosa. Eles são exemplos clássicos de algoritmos gulosos.

Algoritmo genérico

- A estratégia gulosa usada baseia-se em um algoritmo genérico que constrói uma AGM incrementalmente.
- O algoritmo mantém um conjunto de arestas A que satisfaz o seguinte invariante:

  No início de cada iteração, A está contido em uma AGM.
- Em cada iteração, determina-se uma aresta (u, v) tal que $A' = A \cup \{(u, v)\}$ também satisfaz o invariante.

  Uma tal aresta é chamada aresta segura (para A).

Algoritmo genérico

AGM-GENÉRICO(G, w)
1 $A \leftarrow \emptyset$
2 **enquanto** A não é uma árvore geradora
3 Encontre uma aresta (u, v) segura para A
4 $A \leftarrow A \cup \{(u, v)\}$
5 **devolva** A

Obviamente o “algoritmo” está correto!

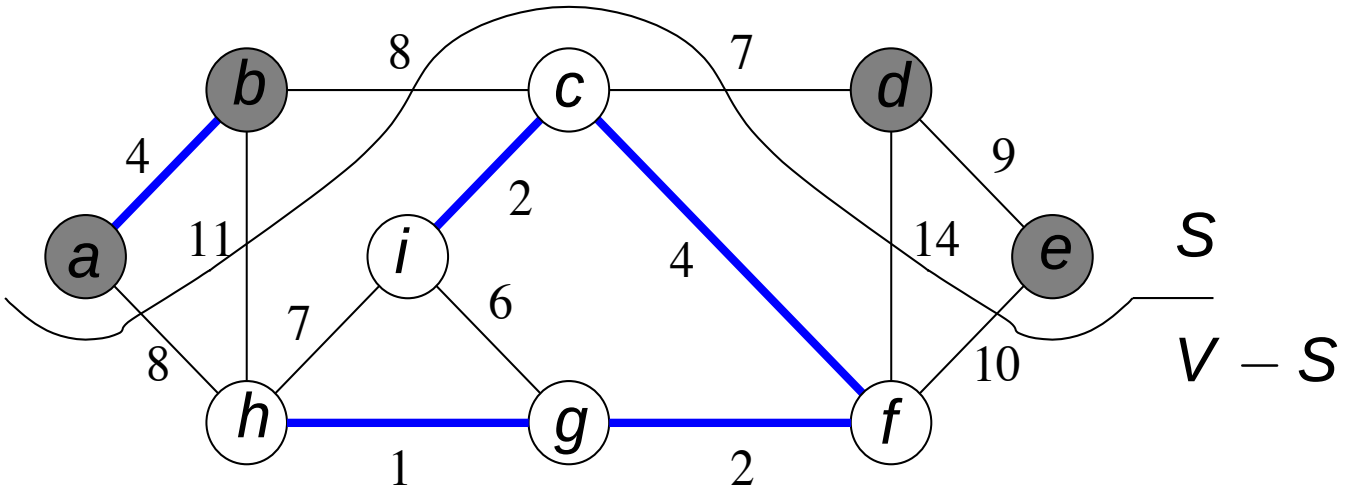
Note que nas linhas 2–4 A está propriamente contido em uma AGM, digamos T. Logo, existe uma aresta segura (u, v) em $T - A$.

Naturalmente, para que isso seja um algoritmo de verdade, é preciso especificar como encontrar uma aresta segura.

Como encontrar arestas seguras

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

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



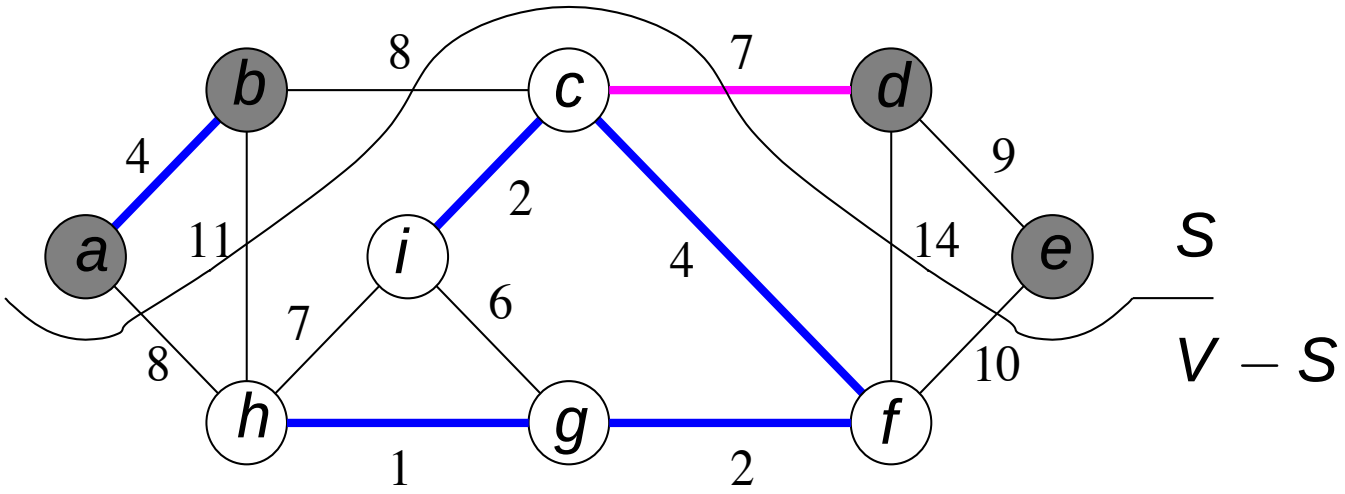
Um corte $\delta(S)$ respeita um conjunto A de arestas se não contém nenhuma aresta de A.

Como encontrar arestas seguras

Uma aresta de um corte $\delta(S)$ é leve se tem o menor peso entre as arestas do corte.

Teorema 23.1: (CLRS)

Seja G um grafo com pesos nas arestas dado por w. Seja A um subconjunto de arestas contido em uma AGM. Seja $\delta(S)$ um corte que respeita A e (u, v) uma aresta leve desse corte. Então (u, v) é uma **aresta segura**.



Como encontrar arestas seguras

Corolário 23.2 (CLRS)

Seja G um grafo com pesos nas arestas dado por w. Seja A um subconjunto de arestas contido em uma AGM. Seja C um componente (árvore) de $G_A = (V, A)$. Se (u, v) é uma aresta leve de $\delta(C)$, então (u, v) é segura para A.

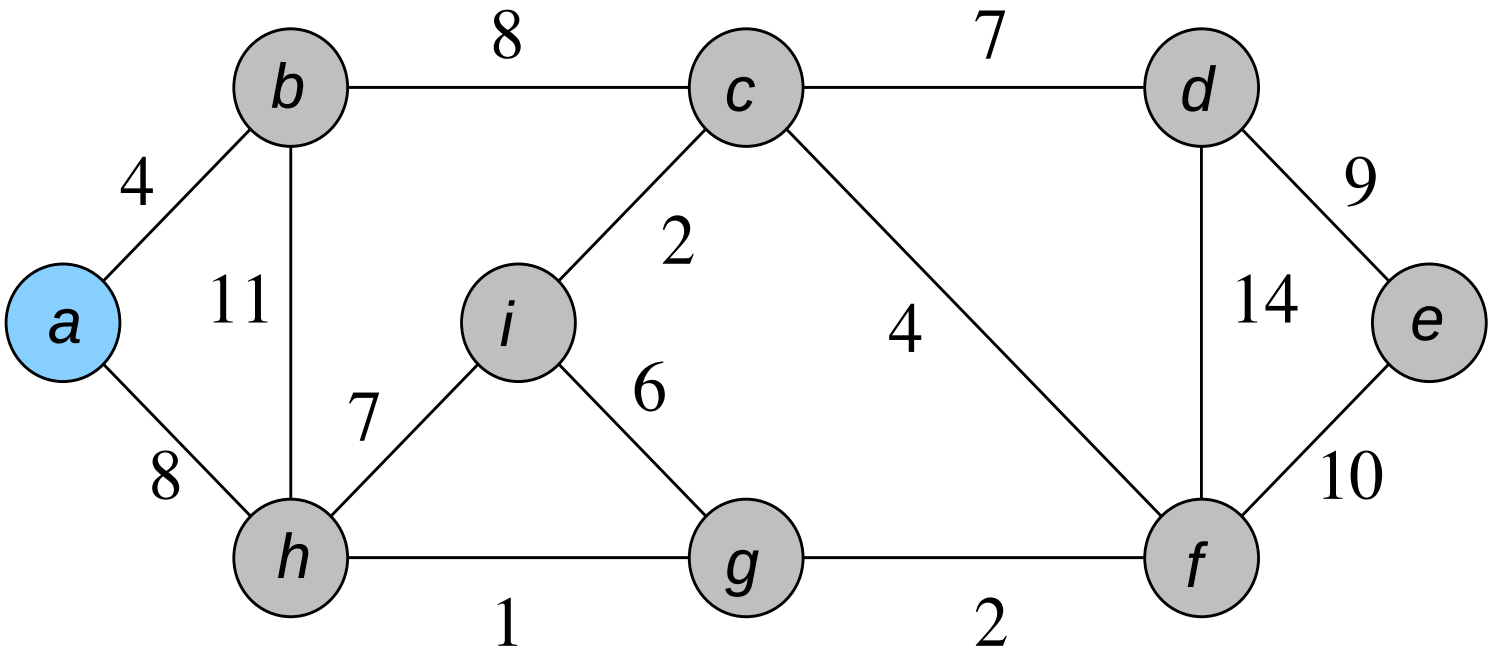
Os algoritmos de Prim e Kruskal são especializações do algoritmo genérico e fazem uso do Corolário 23.2.

O algoritmo de Prim

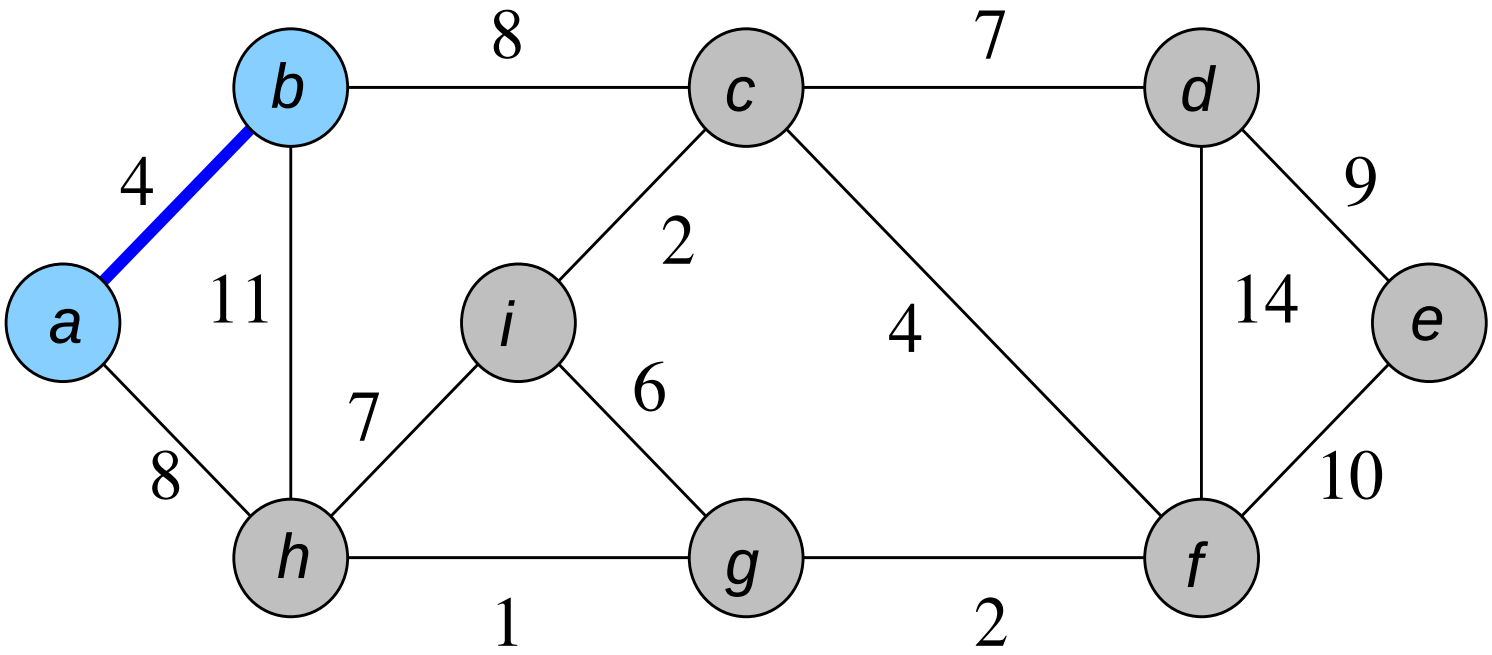
- No algoritmo de Prim, o conjunto A é uma árvore com raiz r (escolhido arbitrariamente no início). Inicialmente, A é vazio.
- Em cada iteração, o algoritmo considera o corte $\delta(C)$ onde C é o conjunto de vértices que são extremos de A.
- Ele encontra uma aresta leve (u, v) neste corte e acrescenta-a ao conjunto A e começa outra iteração até que A seja uma árvore geradora.

Um detalhe de implementação importante é como encontrar eficientemente uma aresta leve no corte.

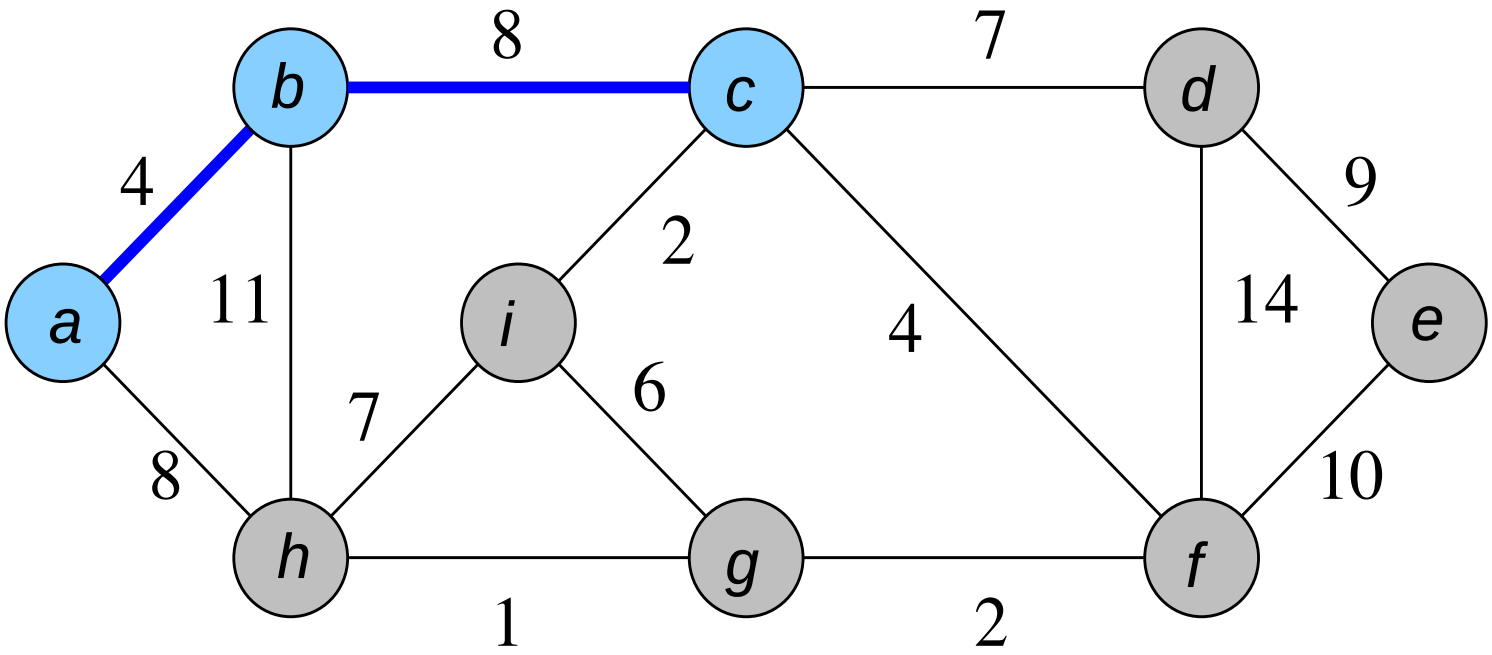
O algoritmo de Prim



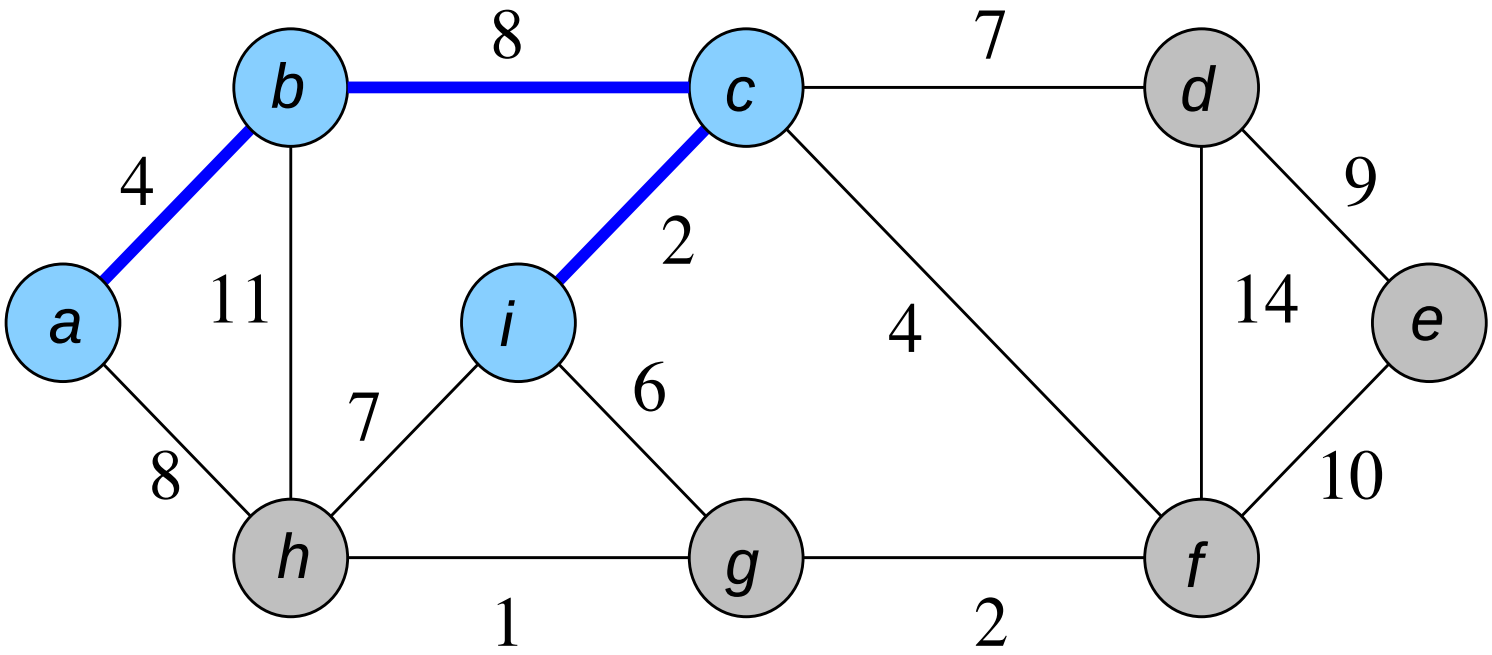
O algoritmo de Prim



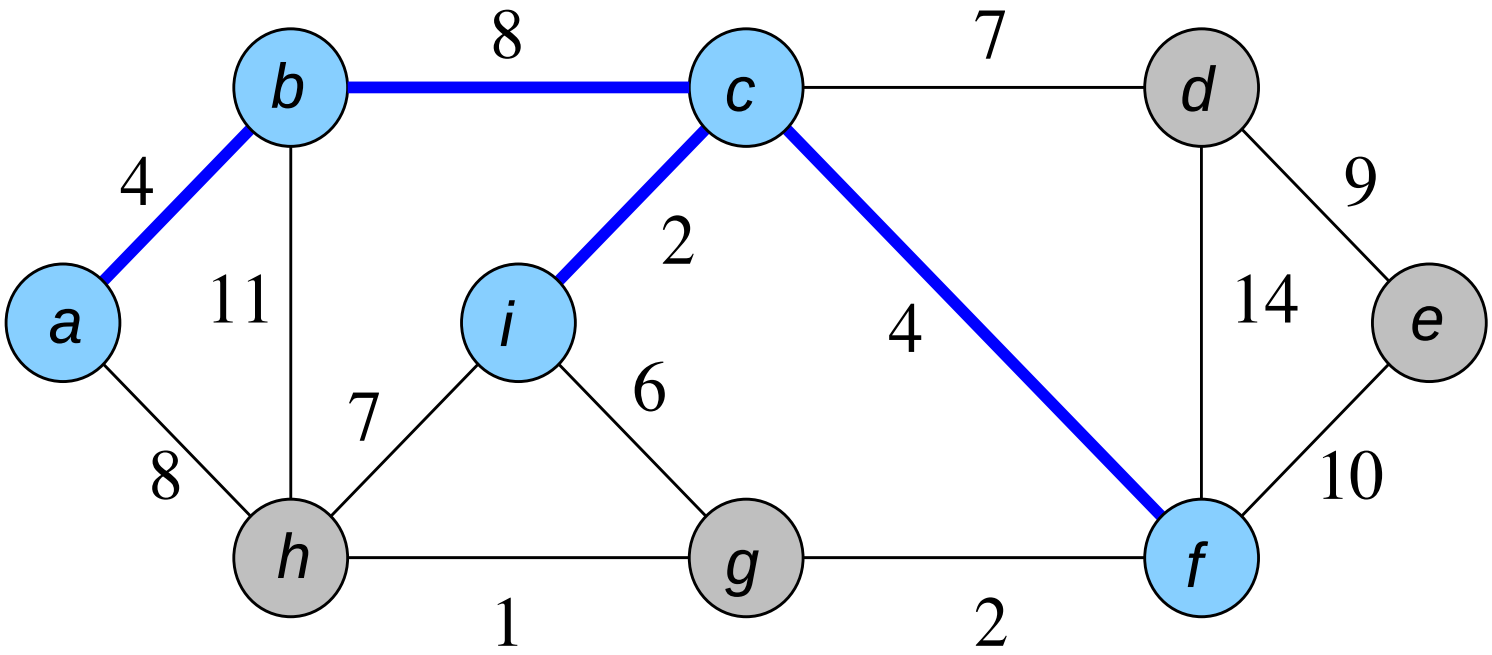
O algoritmo de Prim



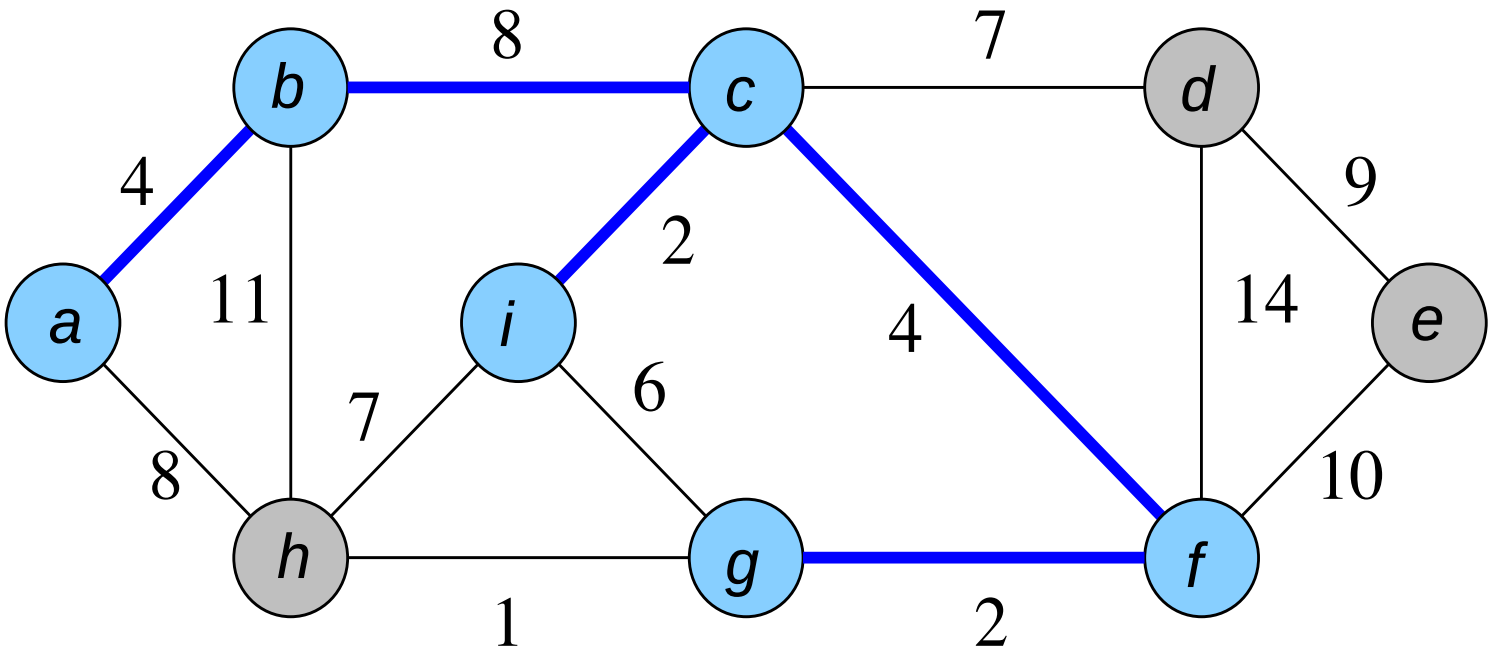
O algoritmo de Prim



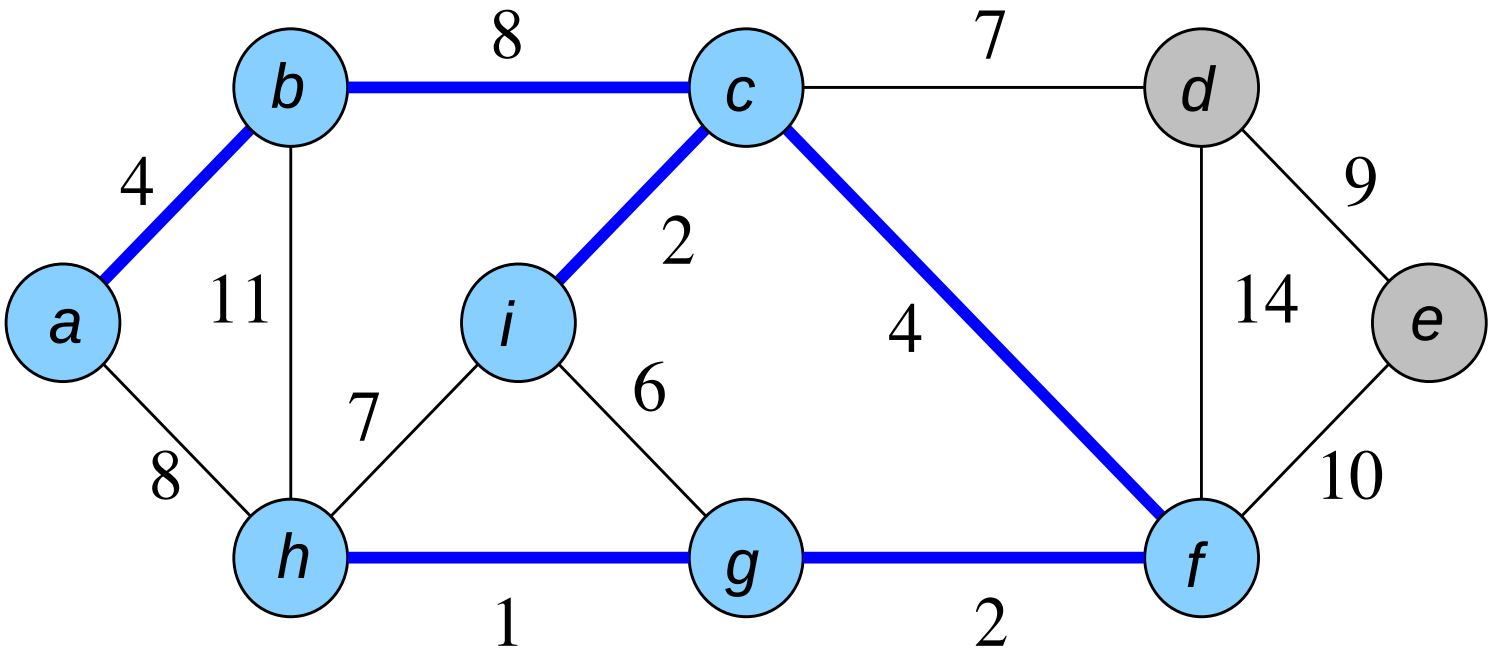
O algoritmo de Prim



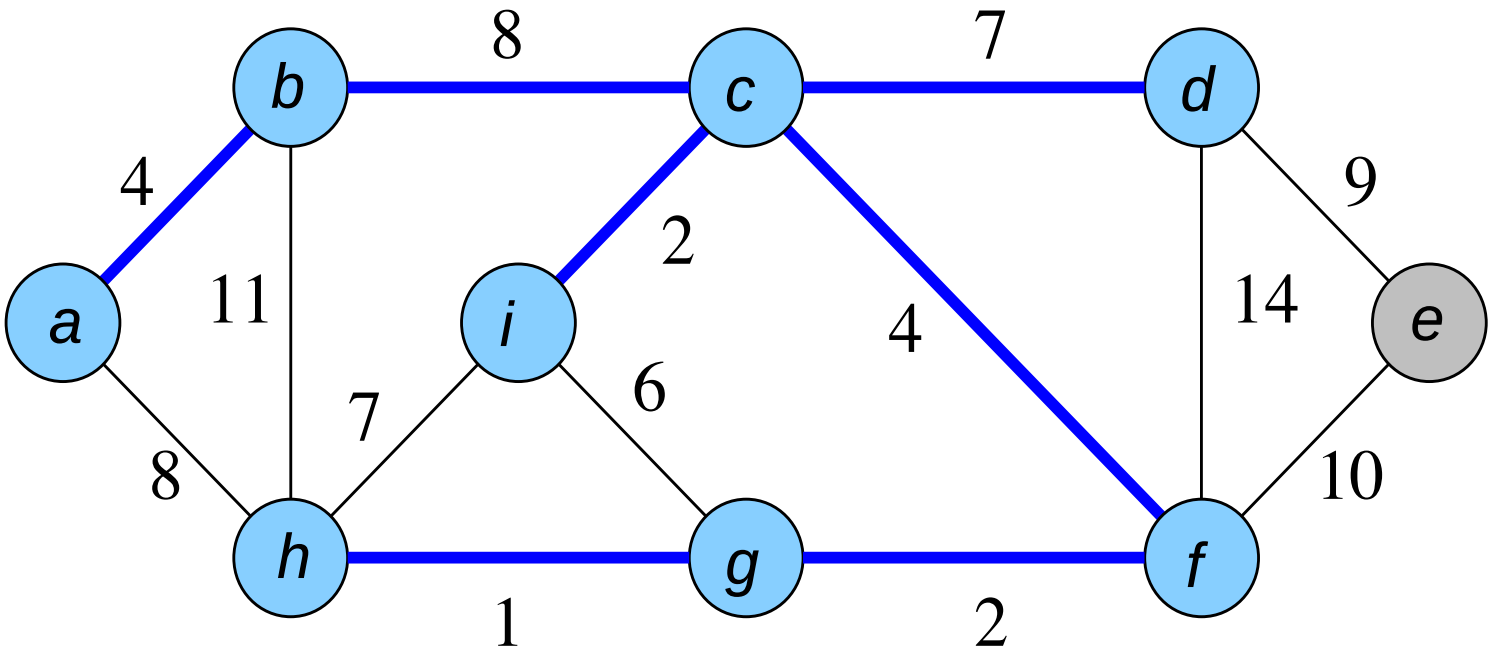
O algoritmo de Prim



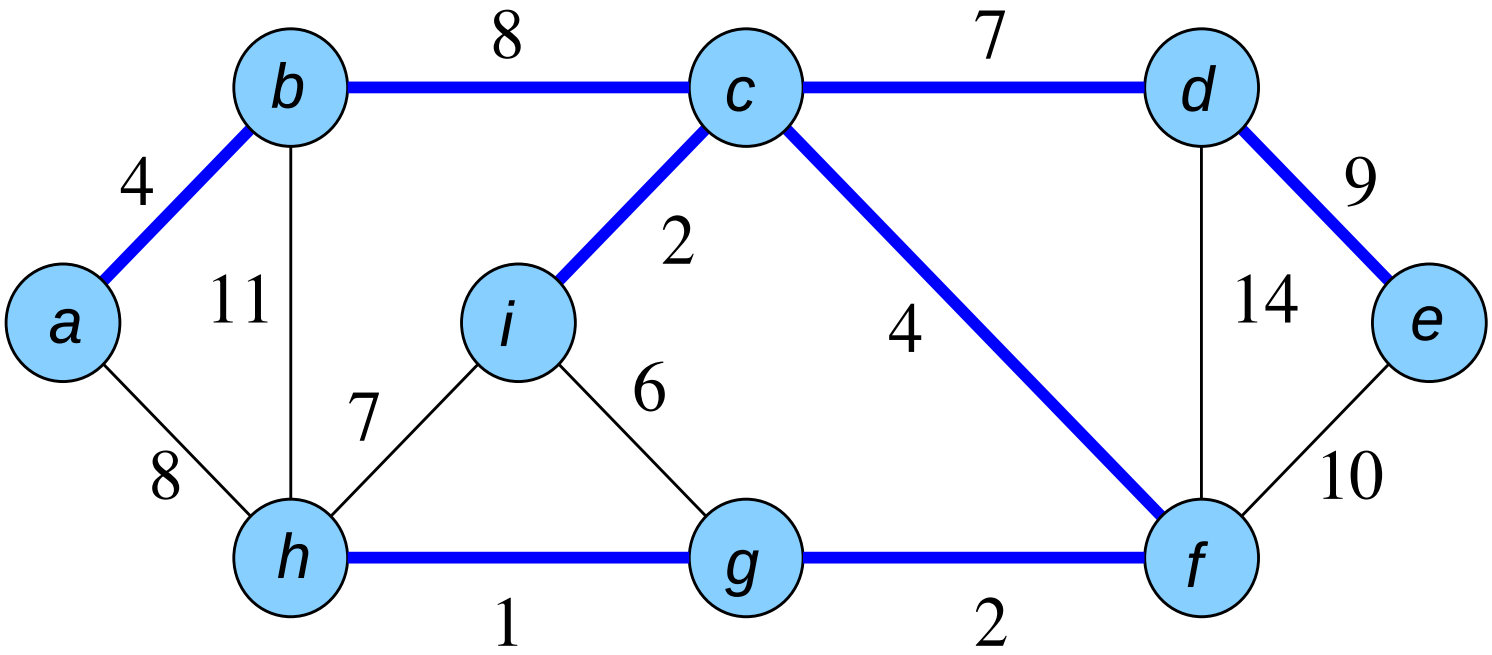
O algoritmo de Prim



O algoritmo de Prim



O algoritmo de Prim



O algoritmo de Prim

O algoritmo mantém durante sua execução as seguintes informações:

- Todos os vértices que não estão na árvore estão em uma fila de prioridade (de mínimo) Q.
- Cada vértice v em Q tem uma chave $key[v]$ que indica o menor peso de qualquer aresta ligando v a algum vértice da árvore. Se não existir nenhuma aresta, então $key[v] = \infty$.
- A variável $\pi[u]$ indica o pai de u na árvore. Então

$$A = \{(u, \pi[u]) : u \in V - \{r\} - Q\}.$$

O algoritmo de Prim

```
AGM-PRIM(G, w, r)
 1  para cada u ∈ V[G]
 2       faça key[u] ← ∞
 3            π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  enquanto Q ≠ ∅ faça
 7       u ← EXTRACT-MIN(Q)
 8       para cada v ∈ Adj[u]
 9            se v ∈ Q e w(u, v) < key[v]
10                 então π[v] ← u
11                      key[v] ← w(u, v)
```

Corretude do algoritmo de Prim

O algoritmo mantém os seguintes invariantes.

No início de cada iteração das linhas 6–11:

- $A = \{(u, \pi[u]) : u \in V - \{r\} - Q\}$.
- O conjunto de vértices da árvore é exatamente $V[G] - Q$.
- Para cada $v \in Q$, se $\pi[v] \neq \text{NIL}$, então $key[v]$ é o peso de uma aresta $(v, \pi[v])$ de menor peso ligando v a um vértice $\pi[v]$ na árvore.

Esse invariantes garantem que o algoritmo sempre escolhe uma aresta segura para acrescentar a A e portanto, o algoritmo está correto.

Complexidade do algoritmo de Prim

Obviamente, a complexidade de AGM-PRIM depende de como a fila de prioridade Q é implementada. Vejamos o que acontece se Q for um min-heap.

- As linhas 1–5 podem ser executadas em tempo $O(V)$ usando BUILD-MIN-HEAP.
- O laço da linha 6 é executado $|V|$ vezes e cada operação EXTRACT-MIN consome tempo $O(\lg V)$, resultando em um tempo total $O(V \lg V)$ para todas as chamadas de EXTRACT-MIN.
- O laço das linhas 8–11 é executado $O(E)$ vezes no total. O teste de pertinência de na fila Q pode ser feito em tempo constante usando um **vetor de bits (booleano)**. Ao atualizar a chave de um vértice, na linha 11, é feita uma chamada implícita a DECREASE-KEY que consome tempo $O(\lg V)$.
- O tempo total é $O(V \lg V + E \lg V) = O(E \lg V)$.

Complexidade do algoritmo de Prim

Pode-se fazer melhor usando uma estrutura de dados chamada *heap de Fibonacci* que guarda $|V|$ elementos e suporta as seguintes operações:

- EXTRACT-MIN – $O(\lg V)$,
- DECREASE-KEY – tempo amortizado $O(1)$.
- INSERT – tempo amortizado $O(1)$.
- Outras operações eficientes que um min-heap não suporta. Por exemplo, UNION.
- Maiores detalhes no CLRS (para quem quiser ver).

Usando um *heap de Fibonacci* para implementar Q melhoramos o tempo para $O(E + V \lg V)$.

Este é um resultado interessante do ponto de vista teórico. Na prática, a implementação anterior comporta-se muito melhor.

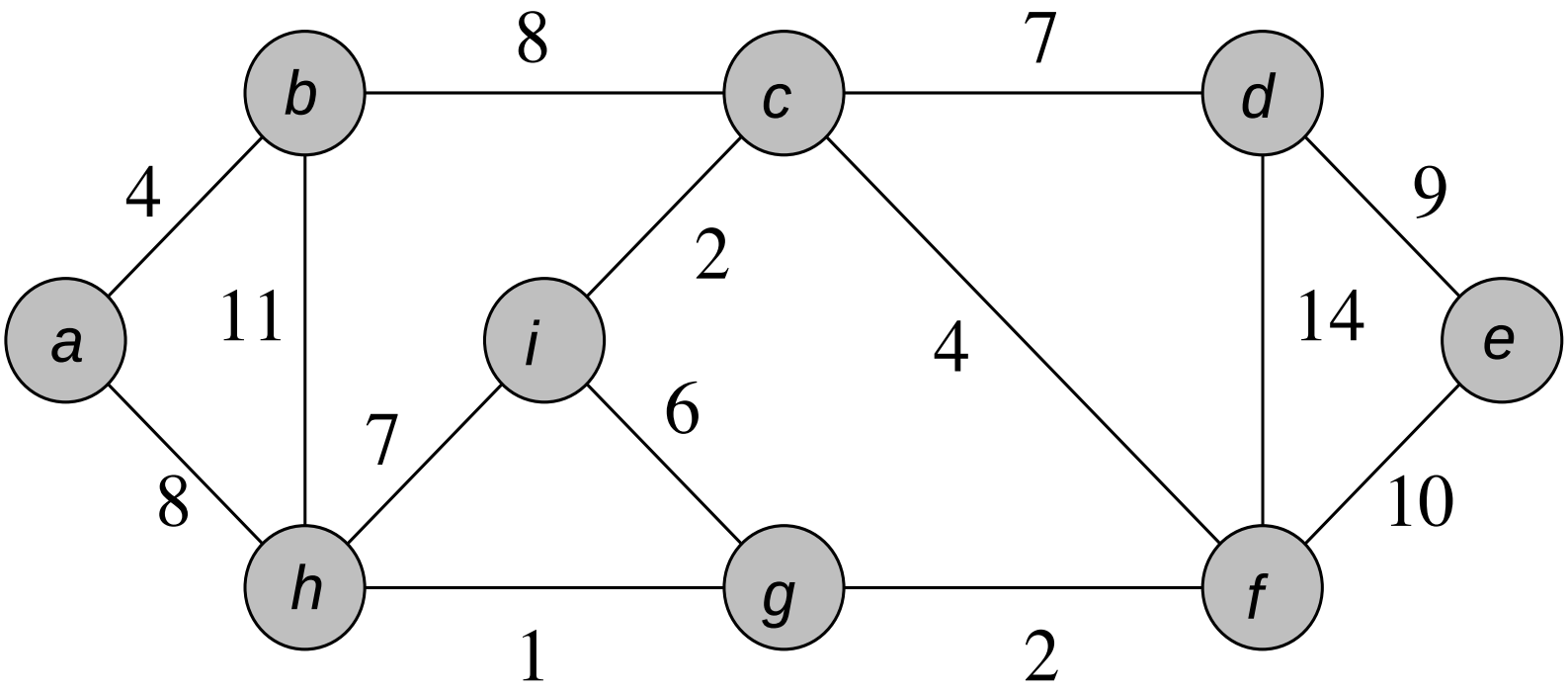
O algoritmo de Kruskal

- No algoritmo de Kruskal o subgrafo $F = (V, A)$ é uma floresta. Inicialmente, A é vazio.
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de menor peso que liga vértices de componentes (árvores) distintos C e C' de $F = (V, A)$.

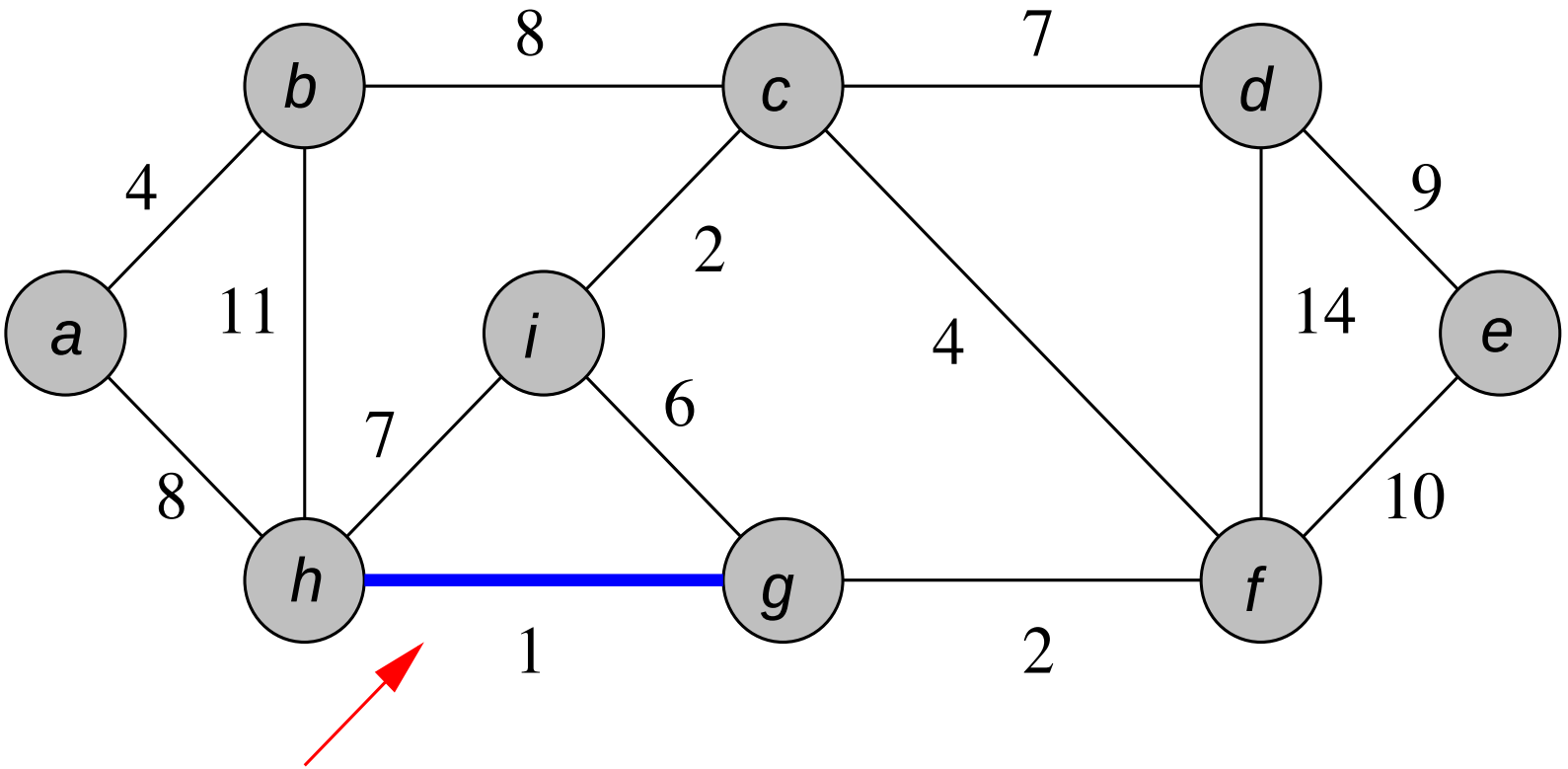
  Note que (u, v) é uma aresta leve do corte $\delta(C)$.
- Ele acrescenta (u, v) ao conjunto A e começa outra iteração até que A seja uma árvore geradora.

Um detalhe de implementação importante é como encontrar a aresta de menor peso ligando componentes distintos.

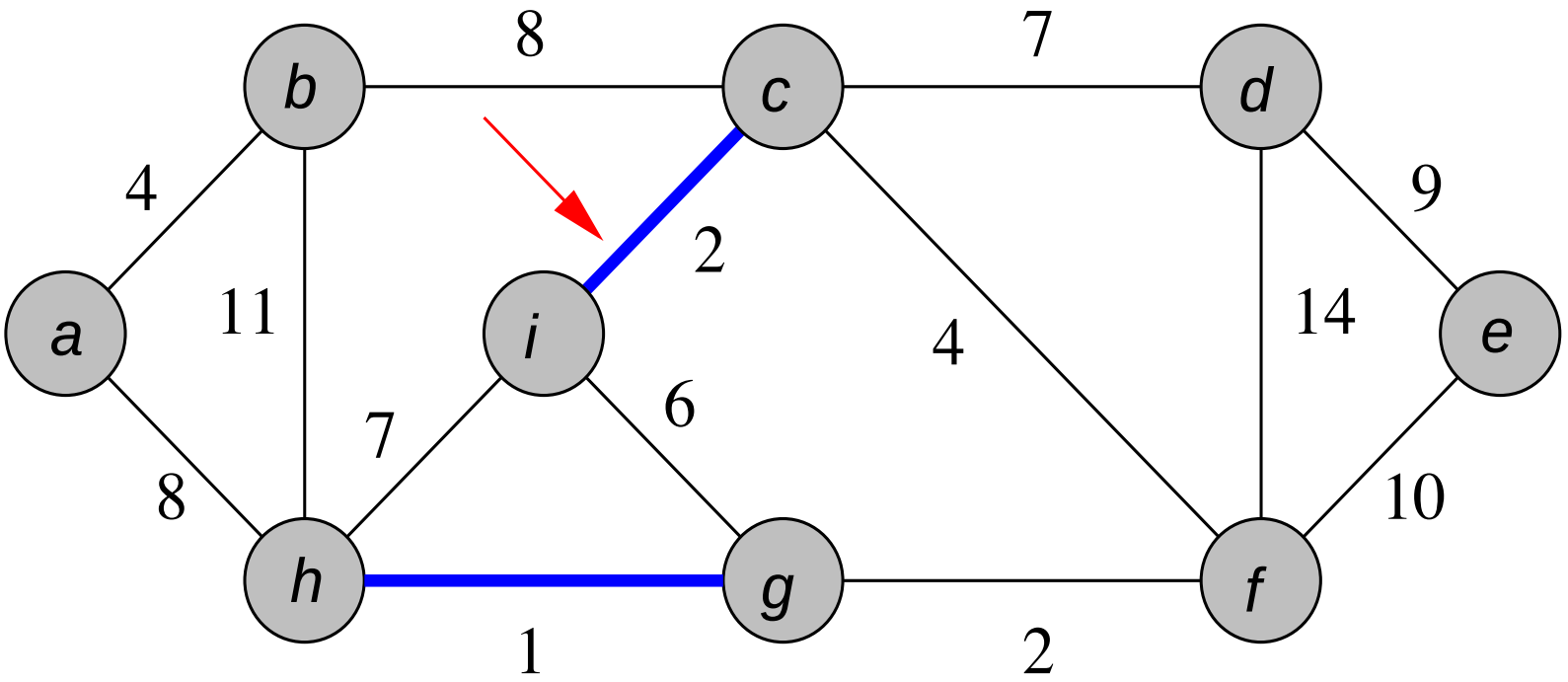
O algoritmo de Kruskal



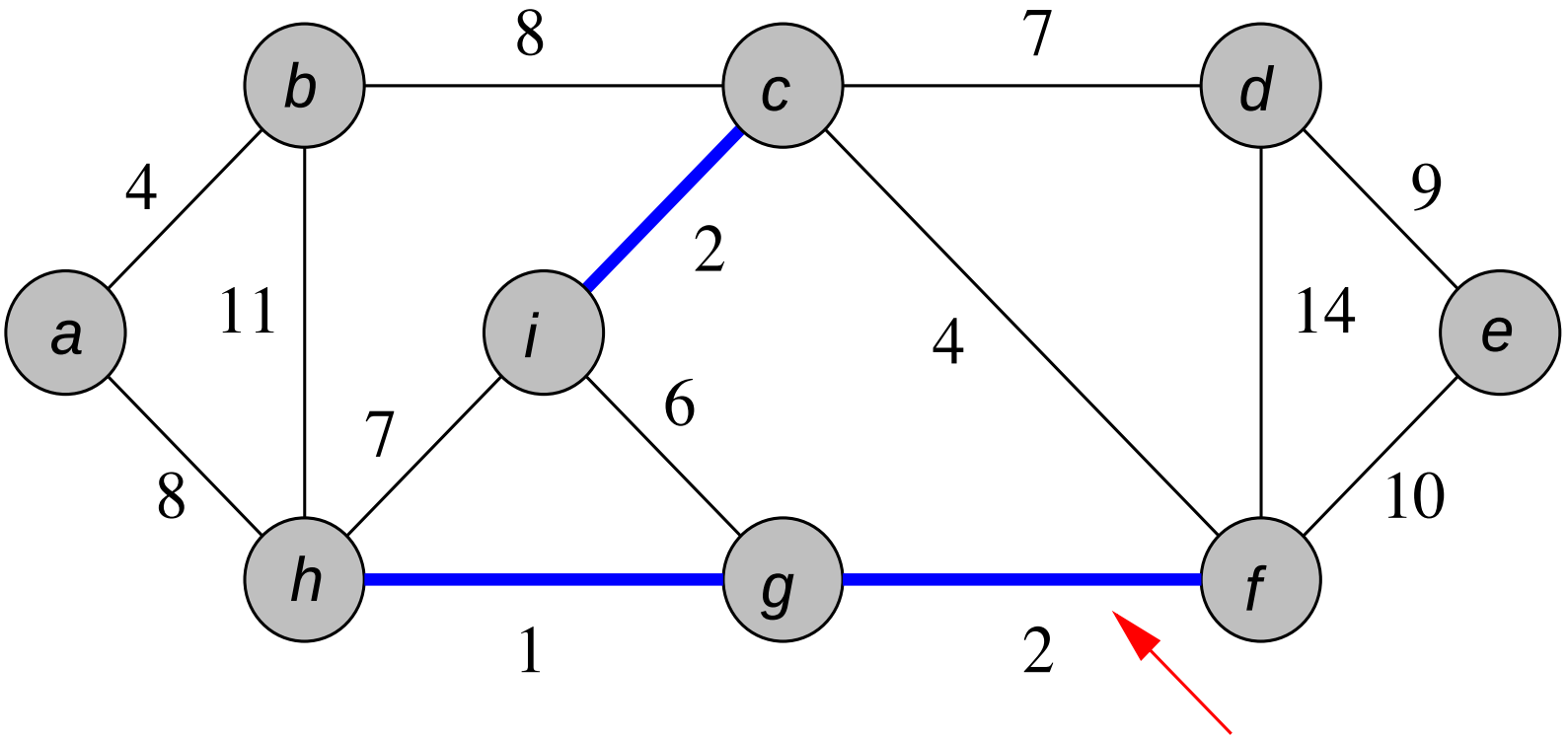
O algoritmo de Kruskal



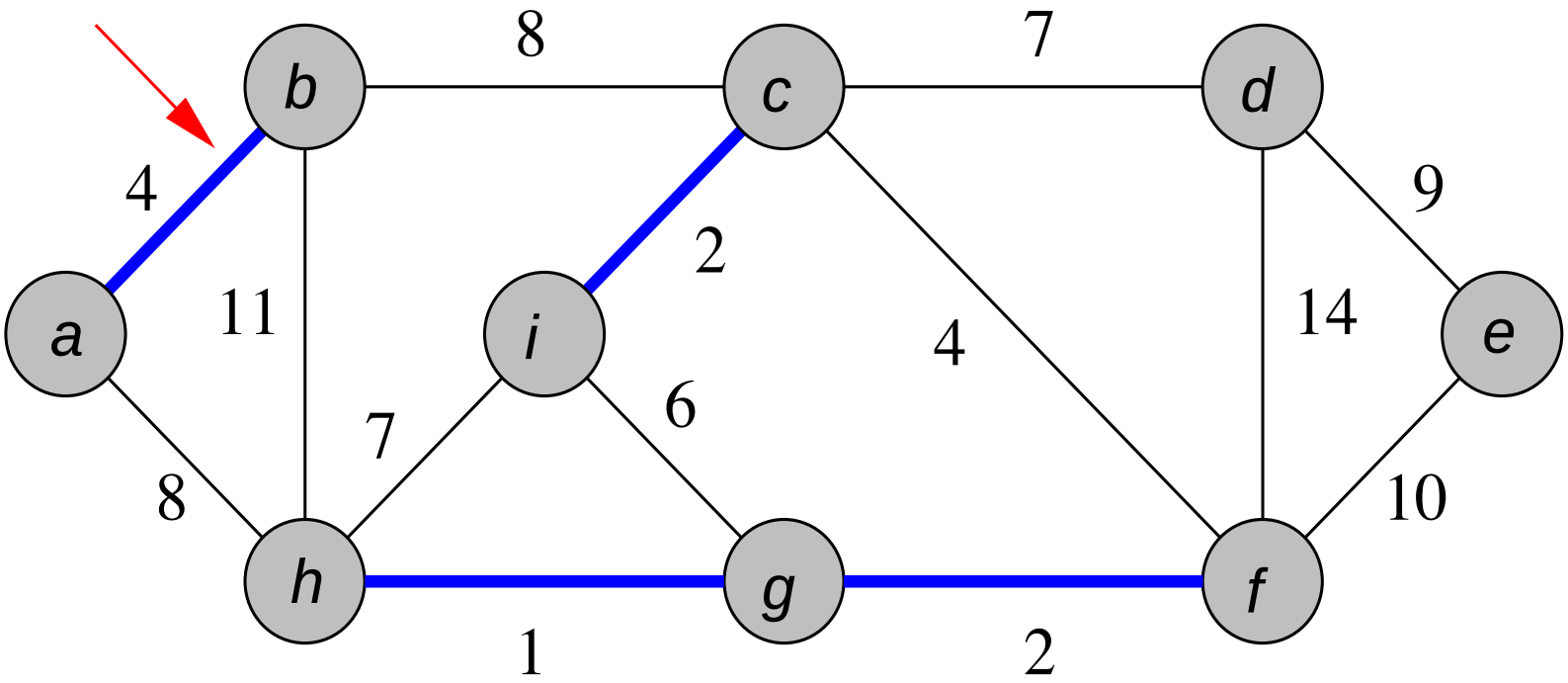
O algoritmo de Kruskal



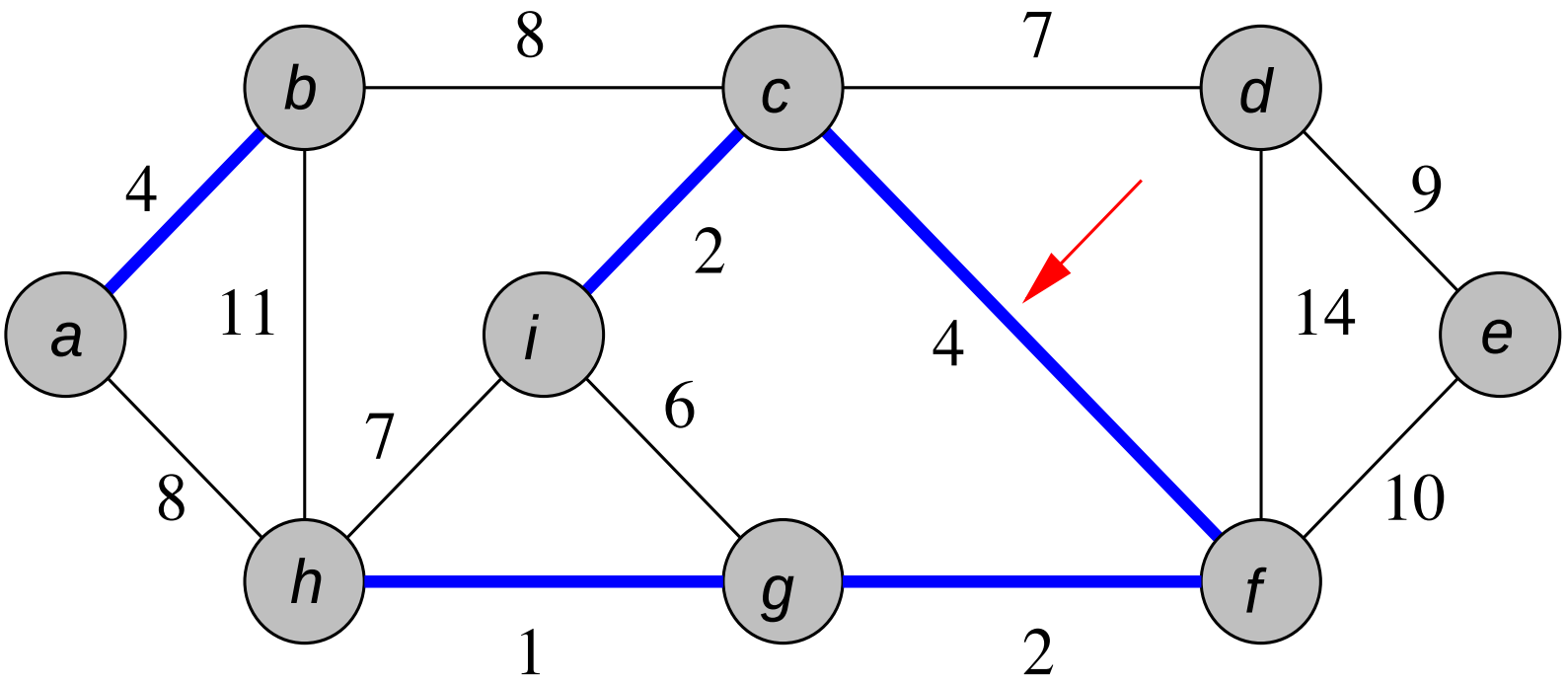
O algoritmo de Kruskal



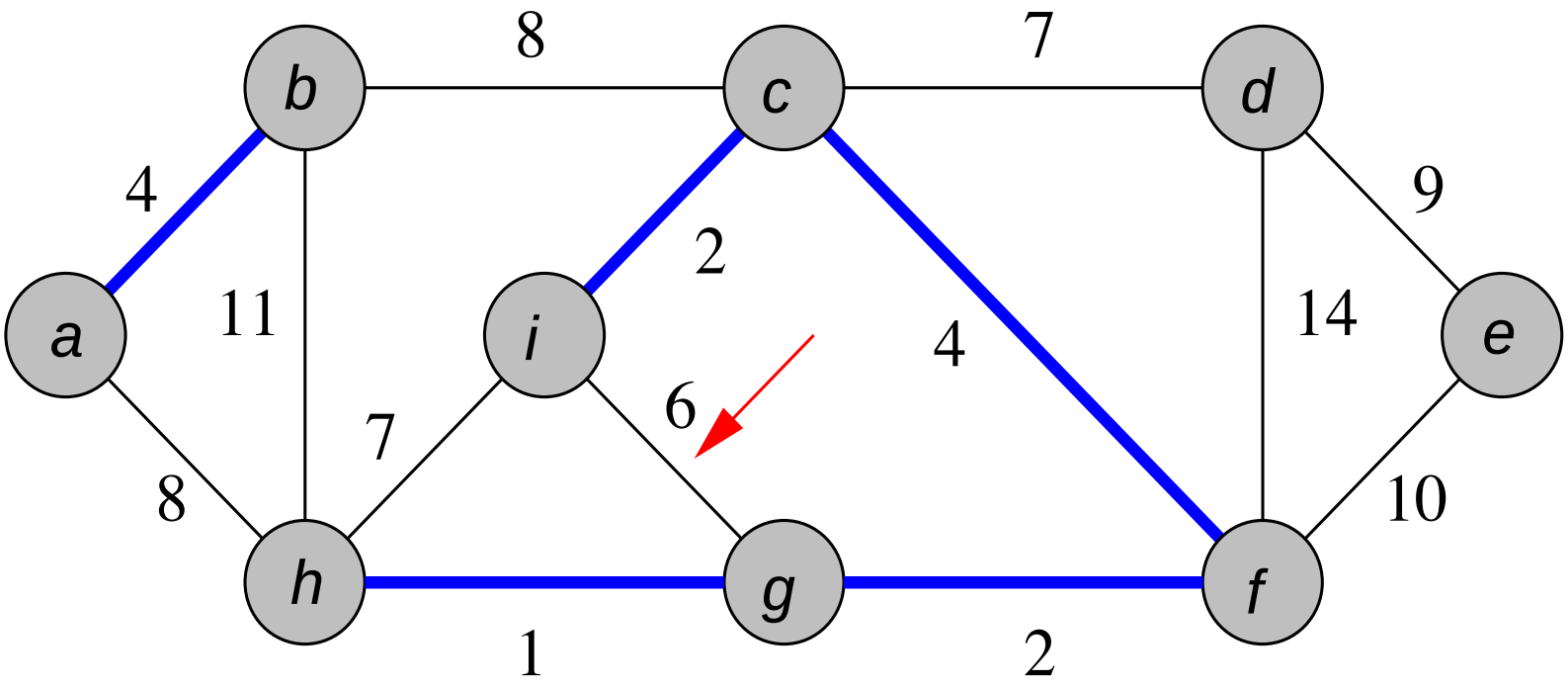
O algoritmo de Kruskal



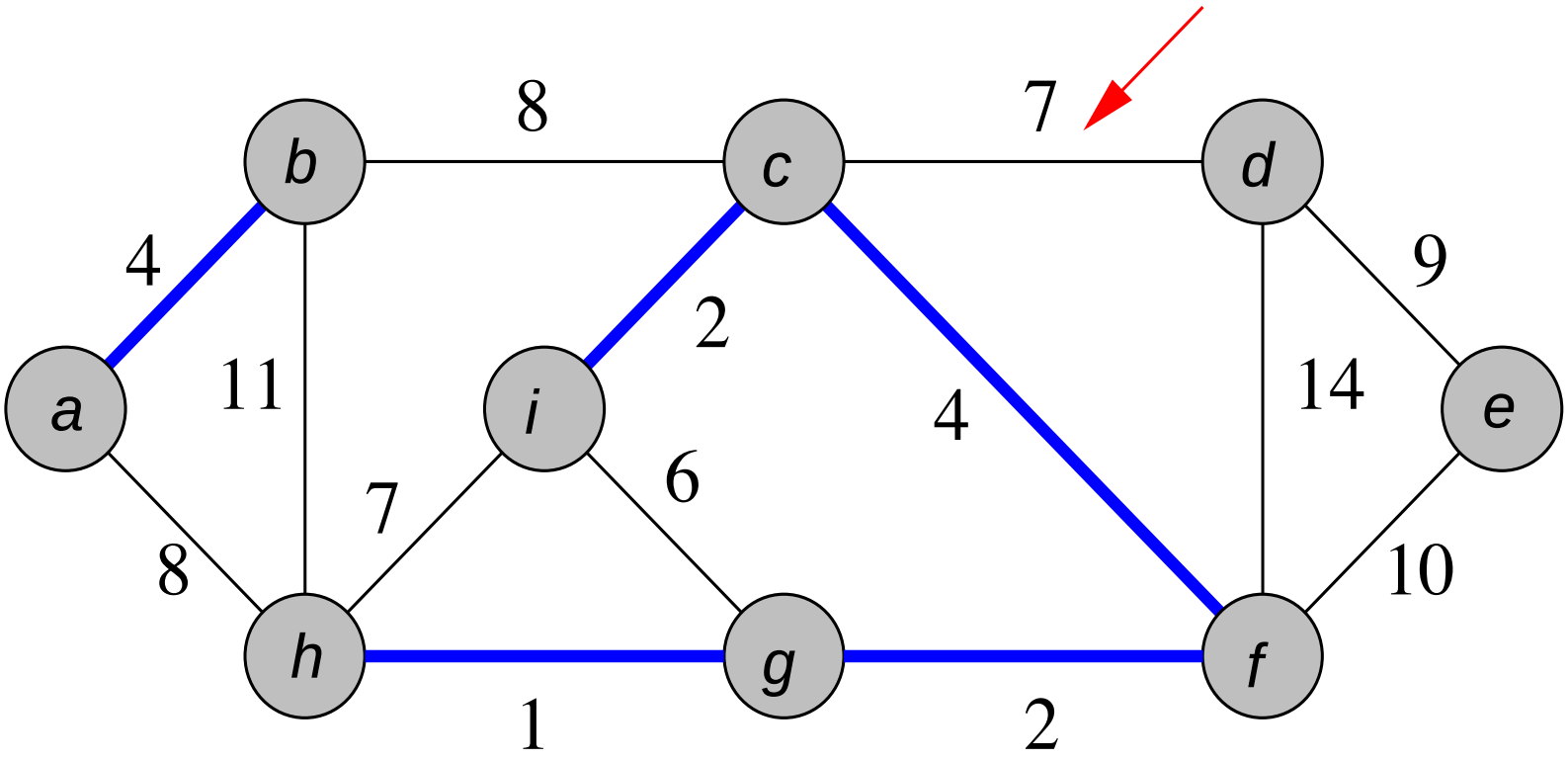
O algoritmo de Kruskal



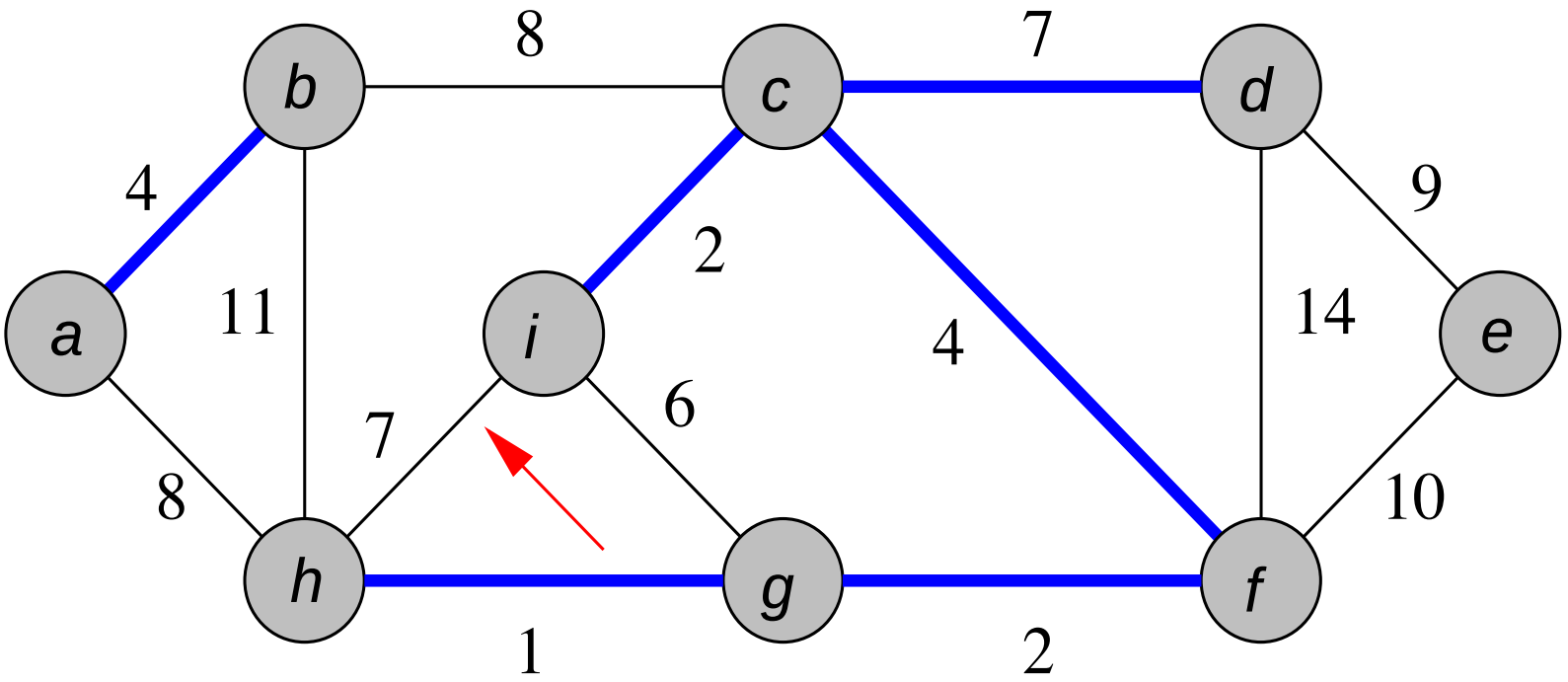
O algoritmo de Kruskal



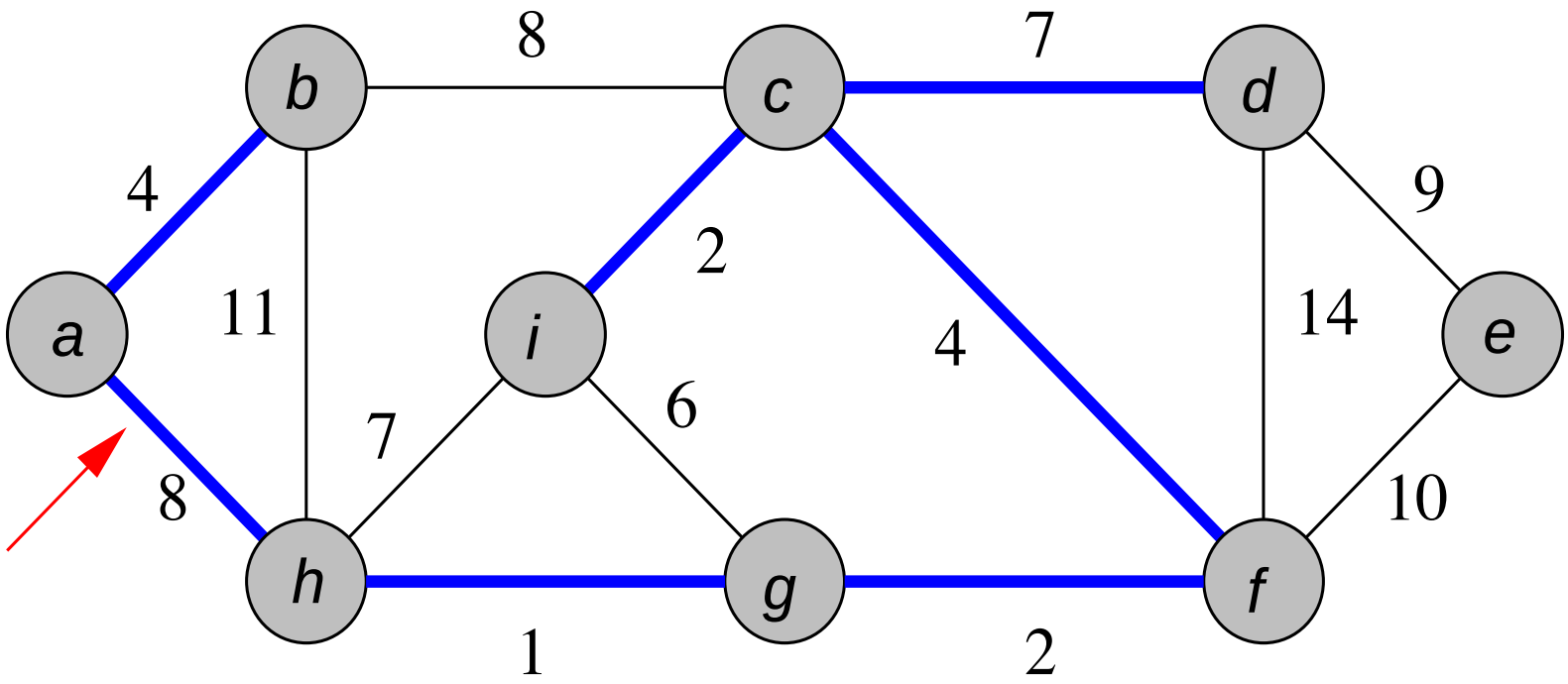
O algoritmo de Kruskal



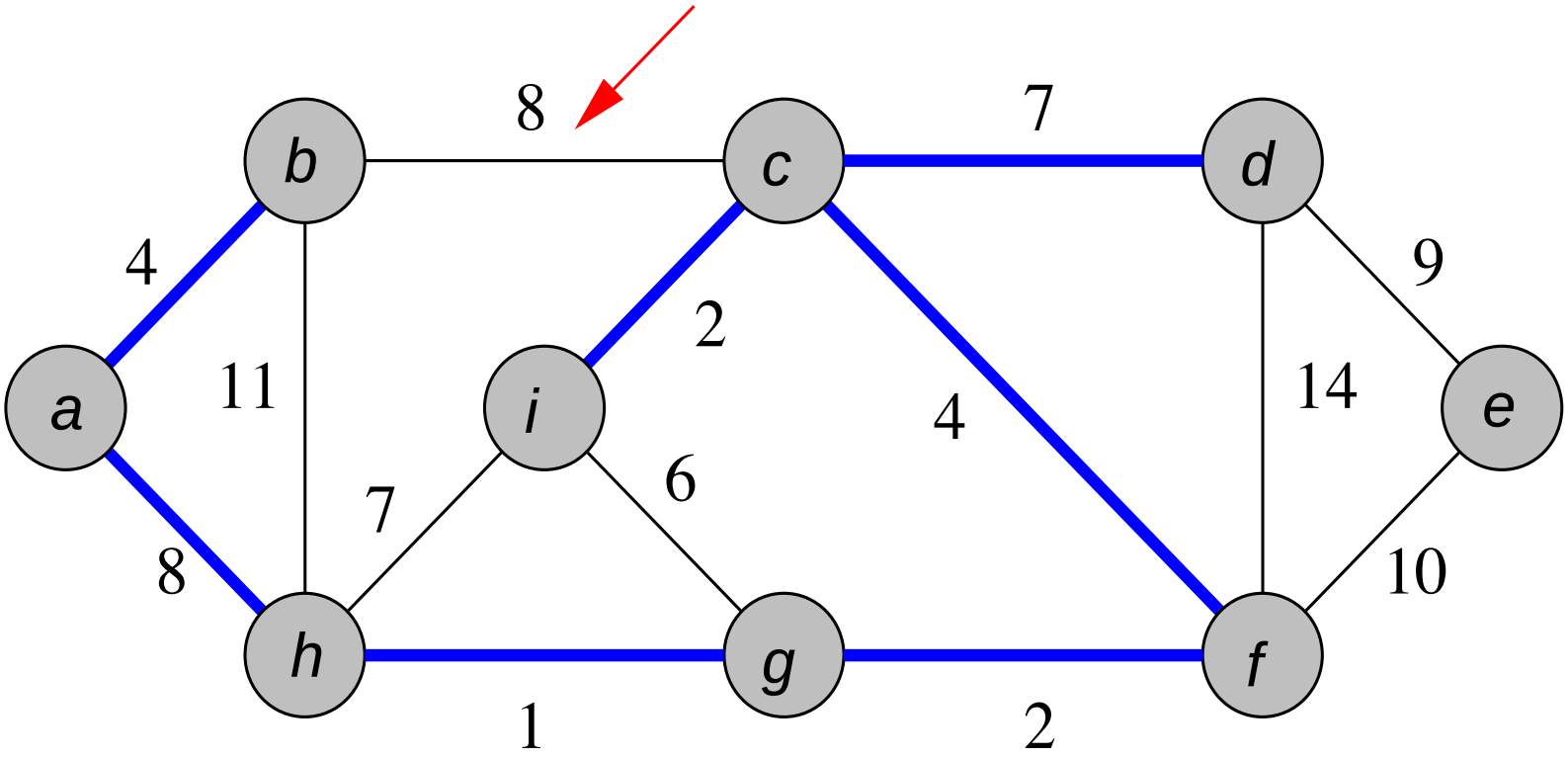
O algoritmo de Kruskal



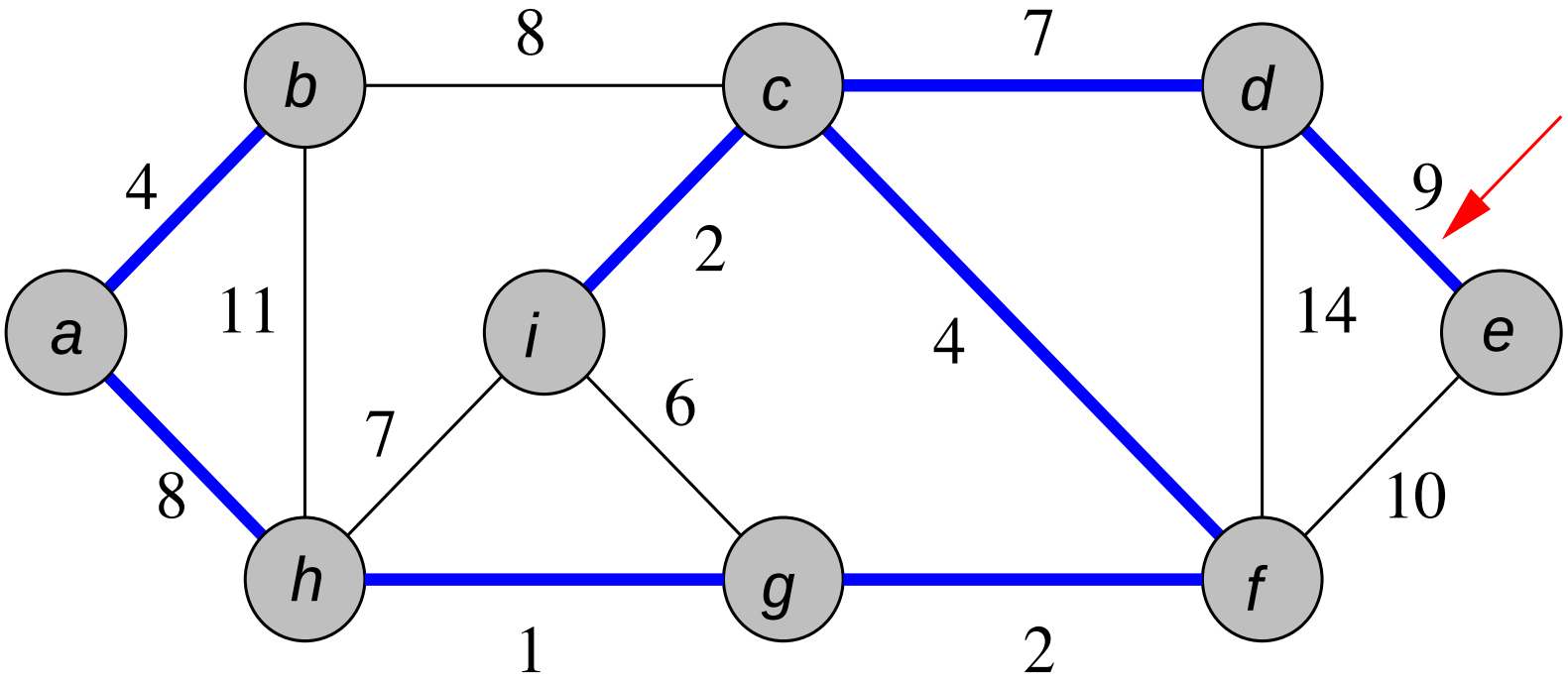
O algoritmo de Kruskal



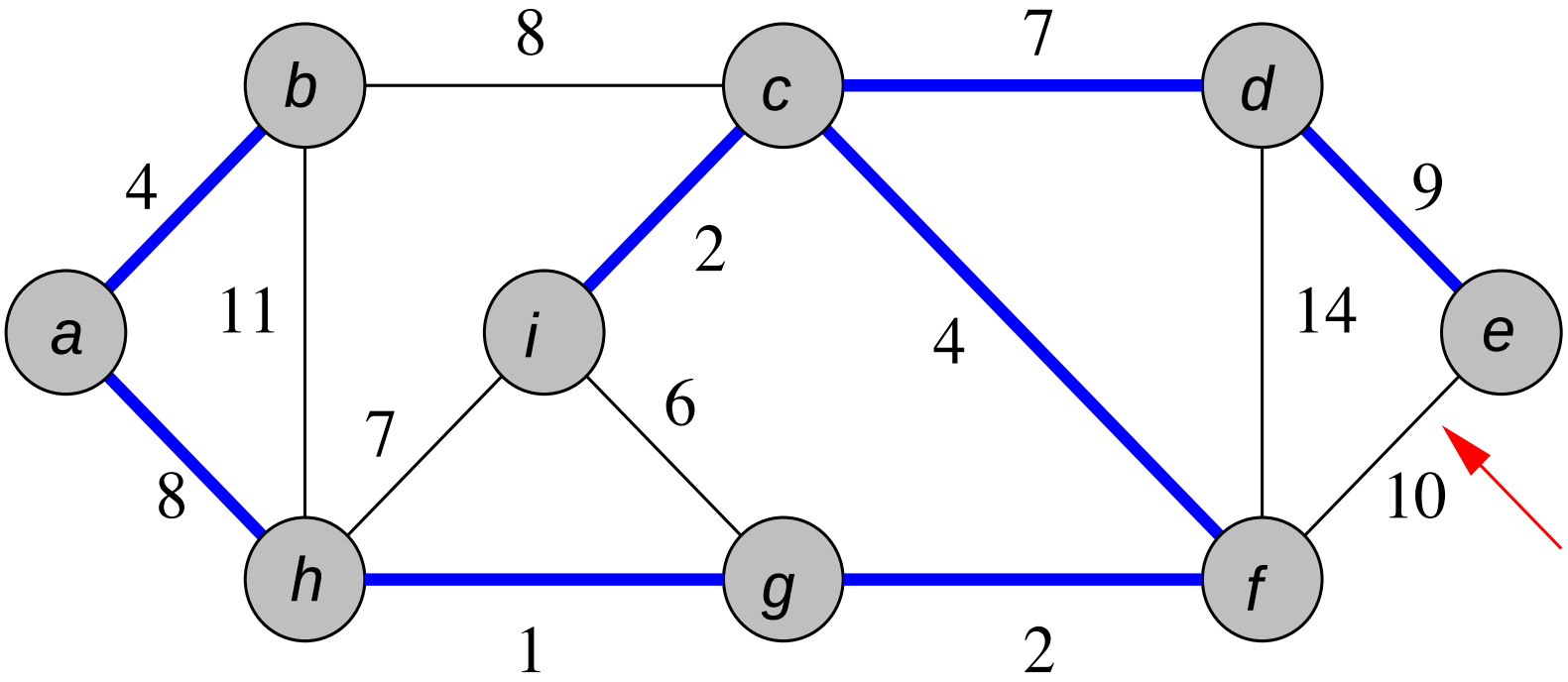
O algoritmo de Kruskal



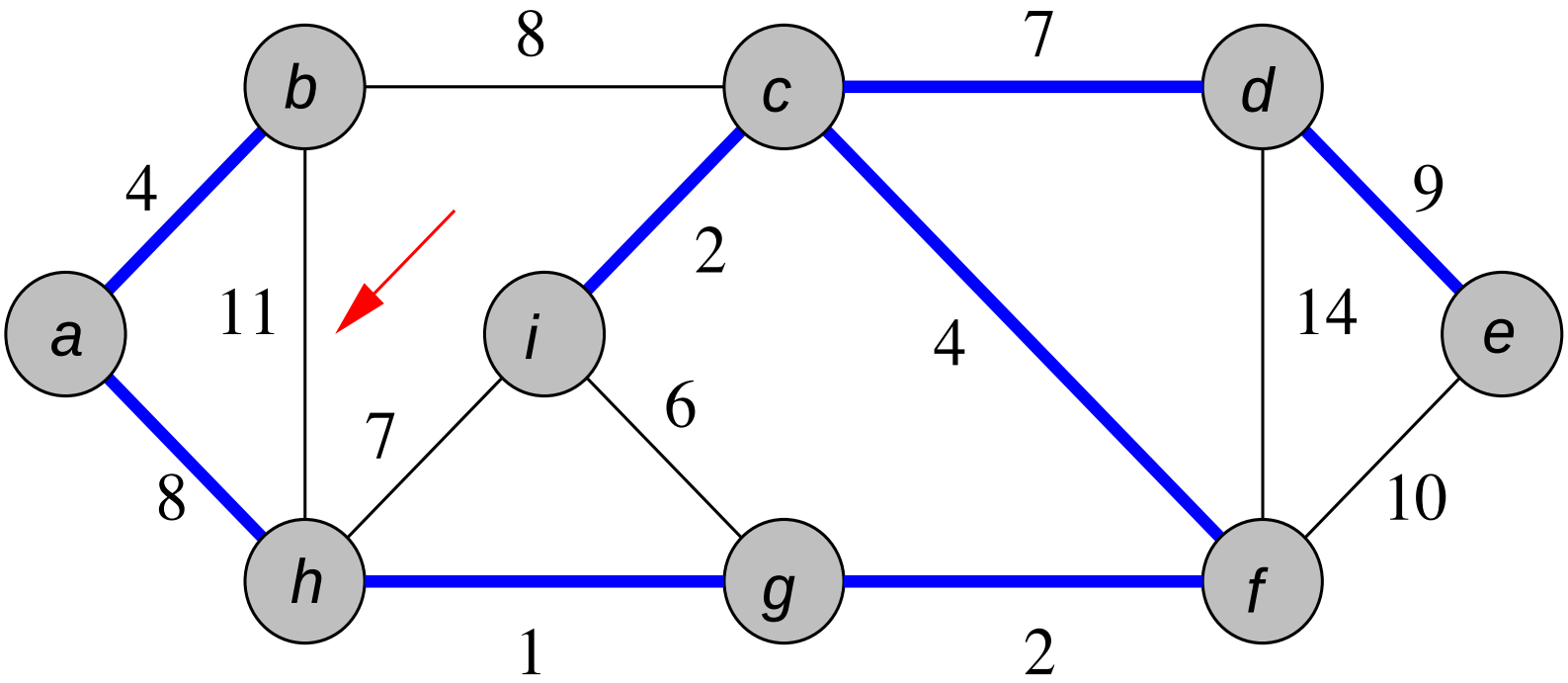
O algoritmo de Kruskal



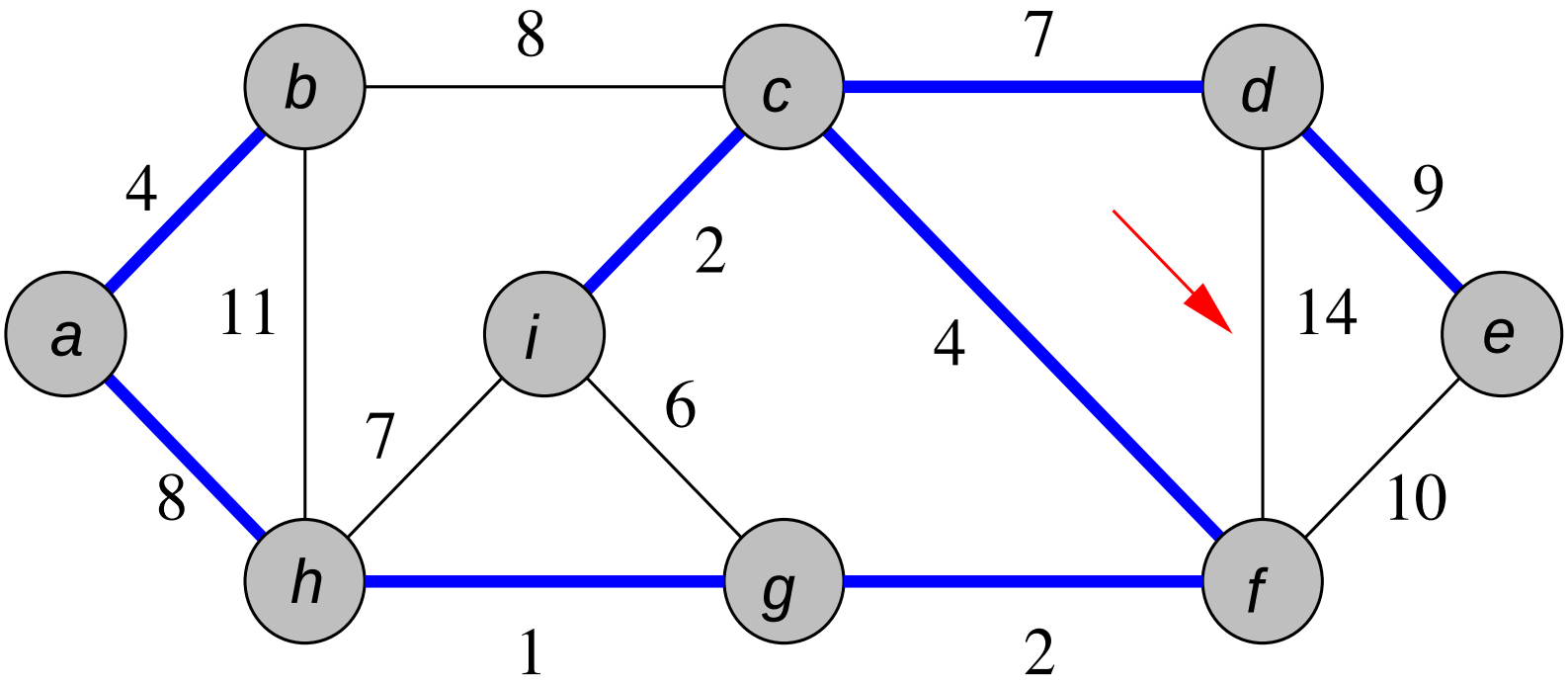
O algoritmo de Kruskal



O algoritmo de Kruskal



O algoritmo de Kruskal



O algoritmo de Kruskal

Eis uma versão 0.0001 do algoritmo de Kruskal.

AGM-KRUSKAL(G, w)
1 $A \leftarrow \emptyset$
2 Ordene as arestas em ordem não-decrescente de peso
3 **para cada** $(u, v) \in E$ nessa ordem **faça**
4 **se** u e v estão em componentes distintos de (V, A)
5 **então** $A \leftarrow A \cup \{(u, v)\}$
6 **devolva** A

Problema: Como verificar eficientemente se u e v estão no mesmo componente da floresta $G_A = (V, A)$?

O algoritmo de Kruskal

Inicialmente $G_A = (V, \emptyset)$, ou seja, G_A corresponde à floresta onde cada componente é um vértice isolado.

Ao longo do algoritmo, esses componentes são modificados pela inclusão de arestas em A.

Uma estrutura de dados para representar $G_A = (V, A)$ deve ser capaz de executar eficientemente as seguintes operações:

- Dado um vértice u, determinar o componente de G_A que contém u e
- dados dois vértices u e v em componentes distintos C e C', fazer a união desses em um novo componente.

ED para conjuntos disjuntos

- Uma estrutura de dados para conjuntos disjuntos mantém uma coleção $\{S_1, S_2, \ldots, S_k\}$ de conjuntos disjuntos dinâmicos (isto é, eles mudam ao longo do tempo).
- Cada conjunto é identificado por um representante que é um elemento do conjunto.

  Quem é o representante é irrelevante, mas se o conjunto não for modificado, então o representante não pode mudar.

ED para conjuntos disjuntos

Uma estrutura de dados para conjuntos disjuntos deve ser capaz de executar as seguintes operações:

- MAKE-SET(x): cria um novo conjunto $\{x\}$.
- UNION(x, y): une os conjuntos (disjuntos) que contém x e y, digamos S_x e S_y, em um novo conjunto $S_x \cup S_y$.

  Os conjuntos S_x e S_y são descartados da coleção.
- FIND-SET(x) devolve um apontador para o representante do (único) conjunto que contém x.

Componentes conexos

```
CONNECTED-COMPONENTS(G)
1   para cada vértice v ∈ V[G] faça
2       MAKE-SET(v)
3   para cada aresta (u, v) ∈ E[G] faça
4       se FIND-SET(u) ≠ FIND-SET(v)
5           então UNION(u, v)

SAME-COMPONENT(u, v)
1   se FIND-SET(u) = FIND-SET(v)
2       então devolva SIM
3       senão devolva NÃO
```

Componentes conexos

"Complexidade" de CONNECTED-COMPONENTS

- $|V|$ chamadas a MAKE-SET
- $2|E|$ chamadas a FIND-SET
- $\leq |V| - 1$ chamadas a UNION

O algoritmo de Kruskal

Eis a versão completa!

AGM-KRUSKAL(G, w)
1 $A \leftarrow \emptyset$
2 **para cada** $v \in V[G]$ **faça**
3 MAKE-SET(v)
4 Ordene as arestas em ordem não-decrescente de peso
5 **para cada** $(u, v) \in E$ nessa ordem **faça**
6 **se** FIND-SET$(u) \neq$ FIND-SET(v)
7 **então** $A \leftarrow A \cup \{(u, v)\}$
8 UNION(u, v)
9 **devolva** A

O algoritmo de Kruskal

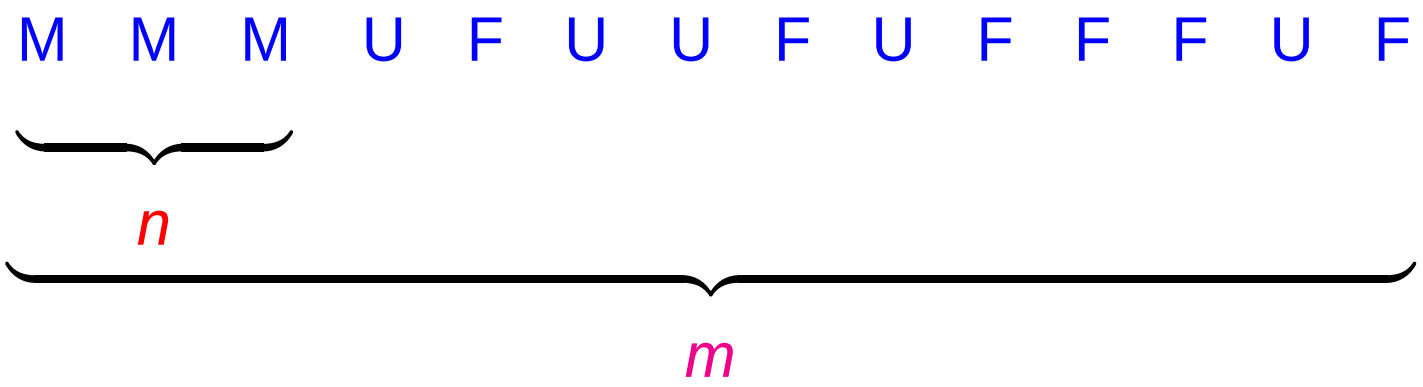
"Complexidade" de AGM-KRUSKAL

- Ordenação: $O(E \lg E)$
- $|V|$ chamadas a MAKE-SET
- $2|E|$ chamadas a FIND-SET
- $\leq |V| - 1$ chamadas a UNION

A complexidade depende de como essas operações são implementadas.

ED para conjuntos disjuntos

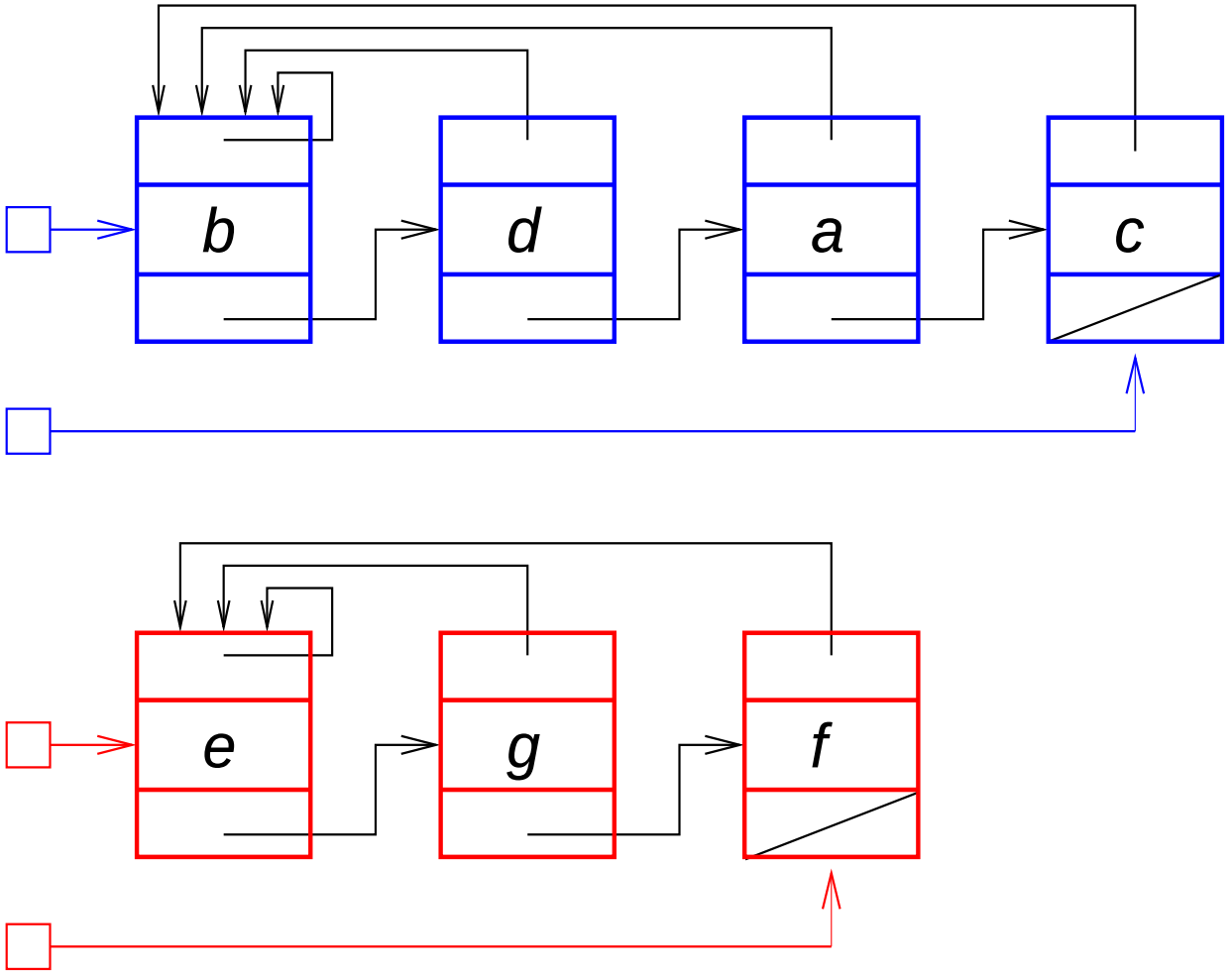
Seqüência de operações MAKE-SET, UNION e FIND-SET



Vamos medir a complexidade das operações em termos de n e m.

Que estrutura de dados usar?
Ou seja, como representar os conjuntos?

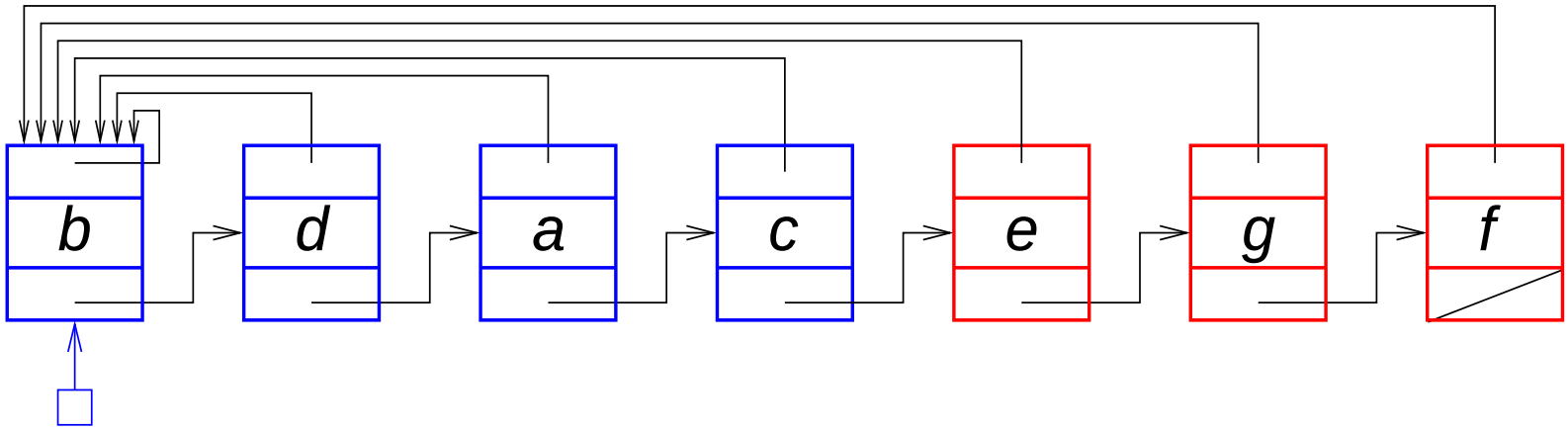
Representação por listas ligadas



- Cada conjunto tem um representante (início da lista)
- Cada nó tem um campo que aponta para o representante
- Guarda-se um apontador para o fim da lista

Representação por listas ligadas

- MAKE-SET(x) – $O(1)$
- FIND-SET(x) – $O(1)$
- UNION(x, y) – concatena a lista de x no final da lista de y



$O(n)$ no pior caso

É preciso atualizar os apontadores para o representante.

Um exemplo de pior caso

Operação	Número de atualizações
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
$\vdots$	$\vdots$
UNION(x_{n-1}, x_n)	n-1

Número total de operações: $2n - 1$

Custo total: $\Theta(n^2)$

Custo amortizado de cada operação: $O(n)$

Uma heurística muito simples

No exemplo anterior, cada chamada de UNION requer em média tempo $\Theta(n)$ pois concatemos a maior lista no final da menor.

Uma idéia simples para evitar esta situação é sempre concatenar a menor lista no final da maior (*weighted-union heuristic*.)

Para implementar isto basta guardar o tamanho de cada lista.

Uma única execução de UNION pode gastar tempo $O(n)$, mas na média o tempo é bem menor (próximo slide).

Uma heurística muito simples

Teorema. Usando a representação por listas ligadas e *weighted-union heuristic*, uma seqüência de m operações MAKE-SET, UNION e FIND-SET gasta tempo $O(m + n \lg n)$.
Prova.

O tempo total em chamadas a MAKE-SET e FIND-SET é $O(m)$.

Sempre que o apontador para o representante de um elemento x é atualizado, o tamanho da lista que contém x (pelo menos) dobra.

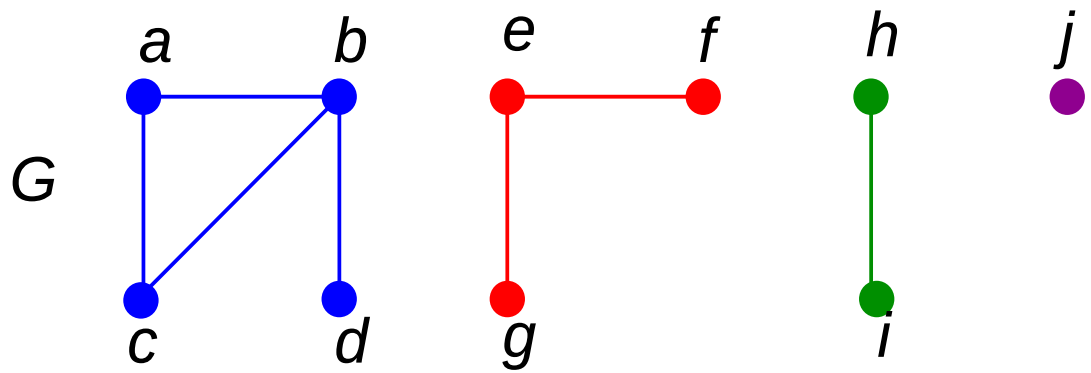
Após ser atualizado $\lceil \lg k \rceil$ vezes, a lista tem tamanho pelo menos k. Como k tem que ser menor que n, cada apontador é atualizado no máximo $O(\lg n)$ vezes.

Assim, o tempo total em chamadas a UNION é $O(n \lg n)$.

Representação por *disjoint-set forests*

- Veremos agora a representação por *disjoint-set forests*.
- Implementações ingênuas não são assintoticamente melhores do que a representação por listas ligadas.
- Usando duas heurísticas — union by rank e path compression — obtemos a representação por *disjoint-set forests* mais eficiente conhecida.

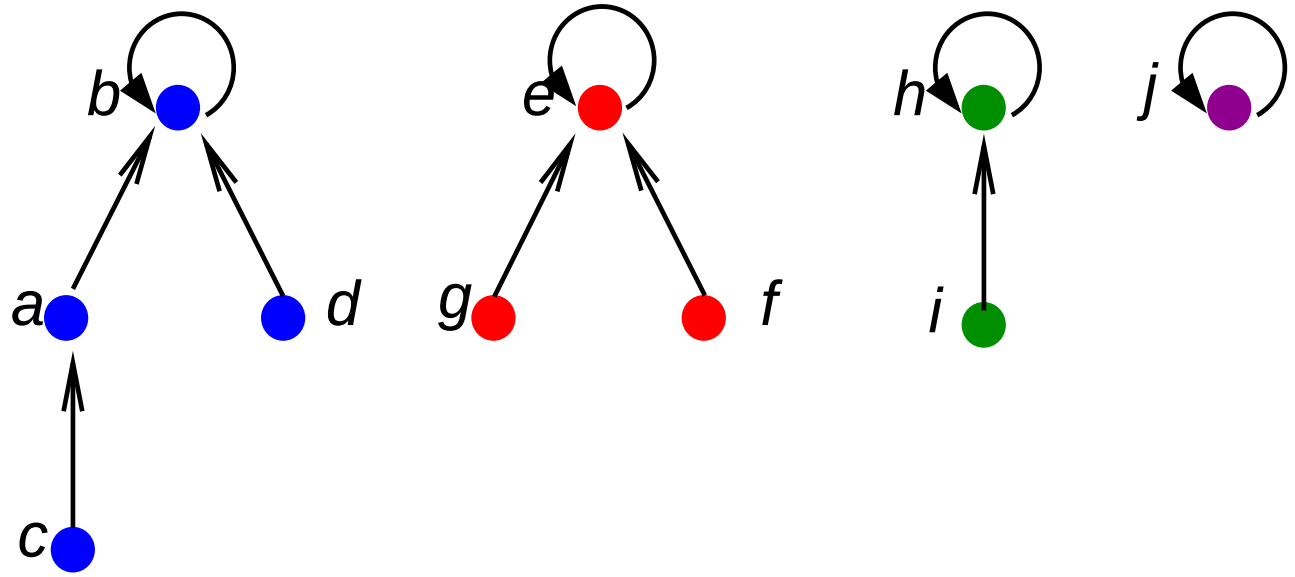
Representação por *disjoint-set forests*



Grafo com vários componentes.

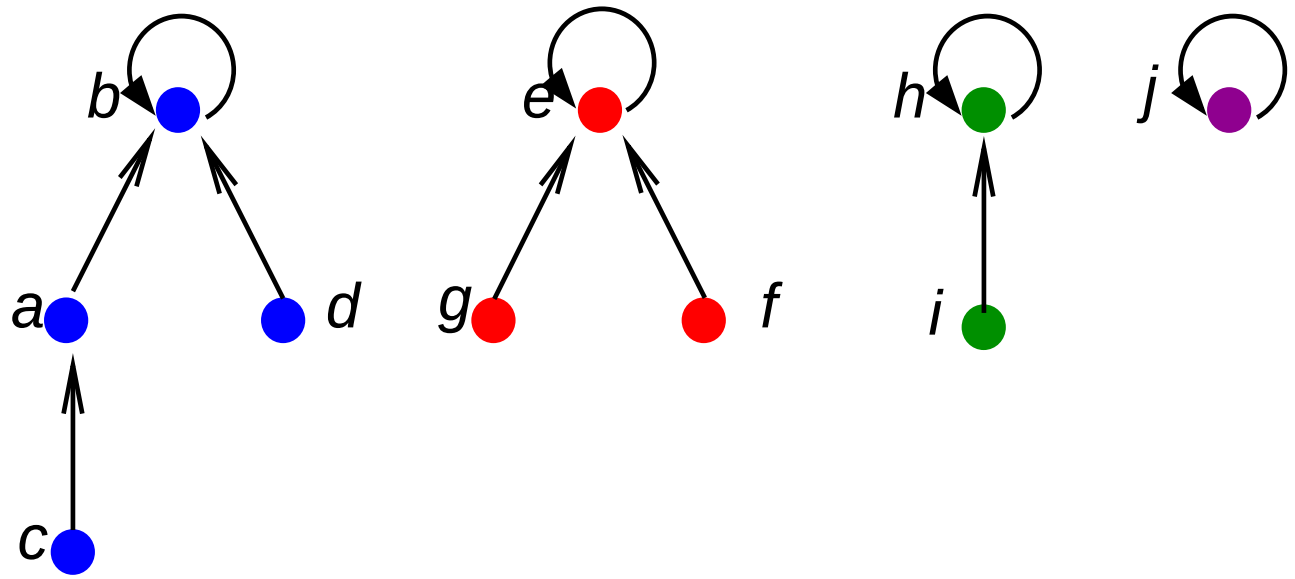
Como é a representação dos componentes na estrutura de dados *disjoint-set forests*?

Representação por *disjoint-set forests*



- Cada conjunto corresponde a uma árvore enraizada.
- Cada elemento aponta para seu pai.
- A raiz é o representante do conjunto e aponta para si mesma.

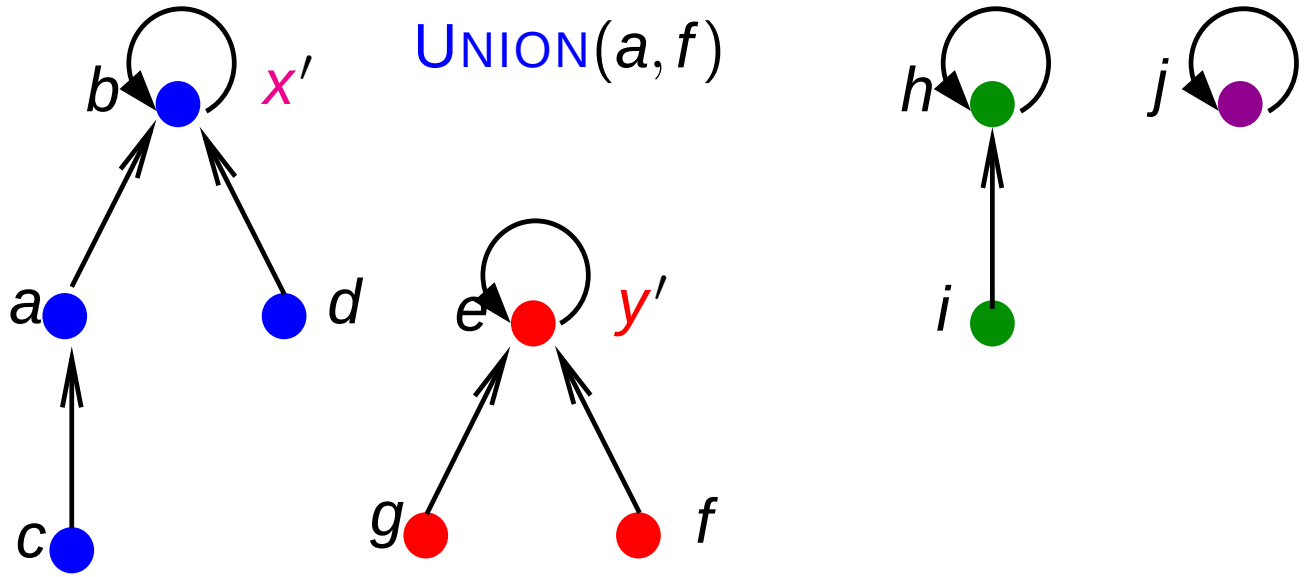
Representação por *disjoint-set forests*



MAKE-SET(x)
1 $\quad \text{pai}[x] \leftarrow x$

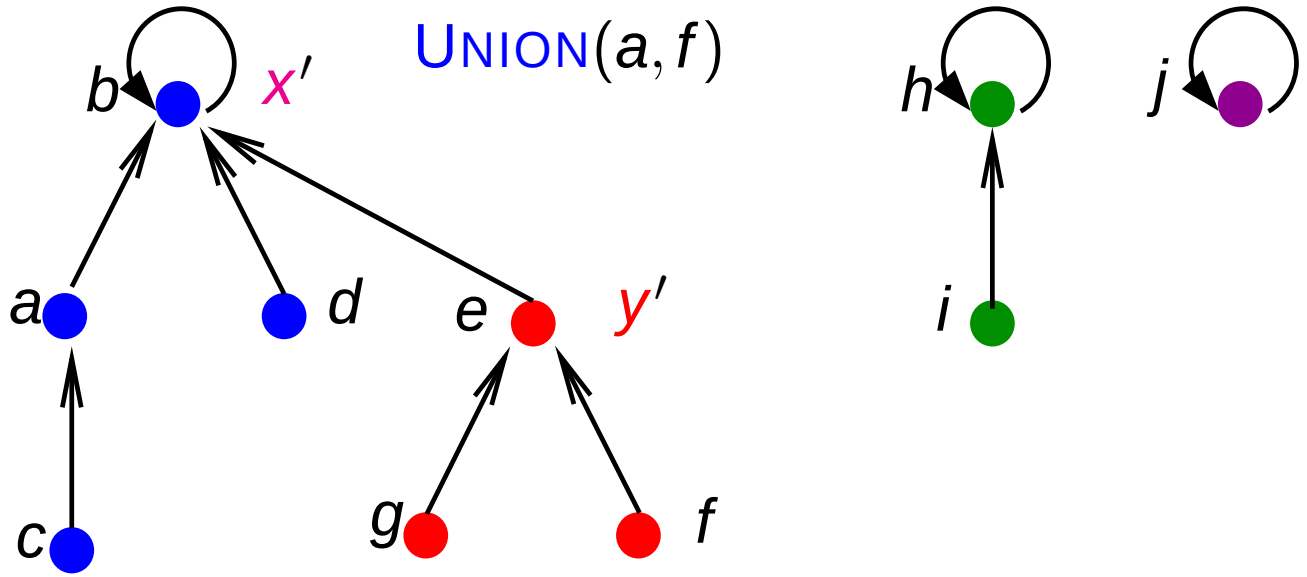
FIND-SET(x)
1 **se** $x = \text{pai}[x]$
2 $\quad$ **então devolva** x
3 $\quad$ **senão devolva** FIND-SET($\text{pai}[x]$)

Representação por *disjoint-set forests*



UNION(x, y)
1 $x' \leftarrow$ FIND-SET(x)
2 $y' \leftarrow$ FIND-SET(y)
3 pai[y'] $\leftarrow x'$

Representação por *disjoint-set forests*



UNION(x, y)
1 $x' \leftarrow$ FIND-SET(x)
2 $y' \leftarrow$ FIND-SET(y)
3 pai[y'] $\leftarrow x'$

Representação por *disjoint-set forests*

Com a implementação descrita até agora, não há melhoria assintótica em relação à representação por listas ligadas.

É fácil descrever uma seqüência de $n - 1$ chamadas a UNION que resultam em uma cadeia linear com n nós.

Pode-se melhorar (muito) isso usado duas heurísticas:

- union by rank
- path compression

Union by rank

- A idéia é emprestada do weighted-union heuristic.
- Cada nó x possui um "posto" $\mathrm{rank}[x]$ que é um limitante superior para a altura de x.
- Em union by rank a raiz com menor rank aponta para a raiz com maior rank.

Union by rank

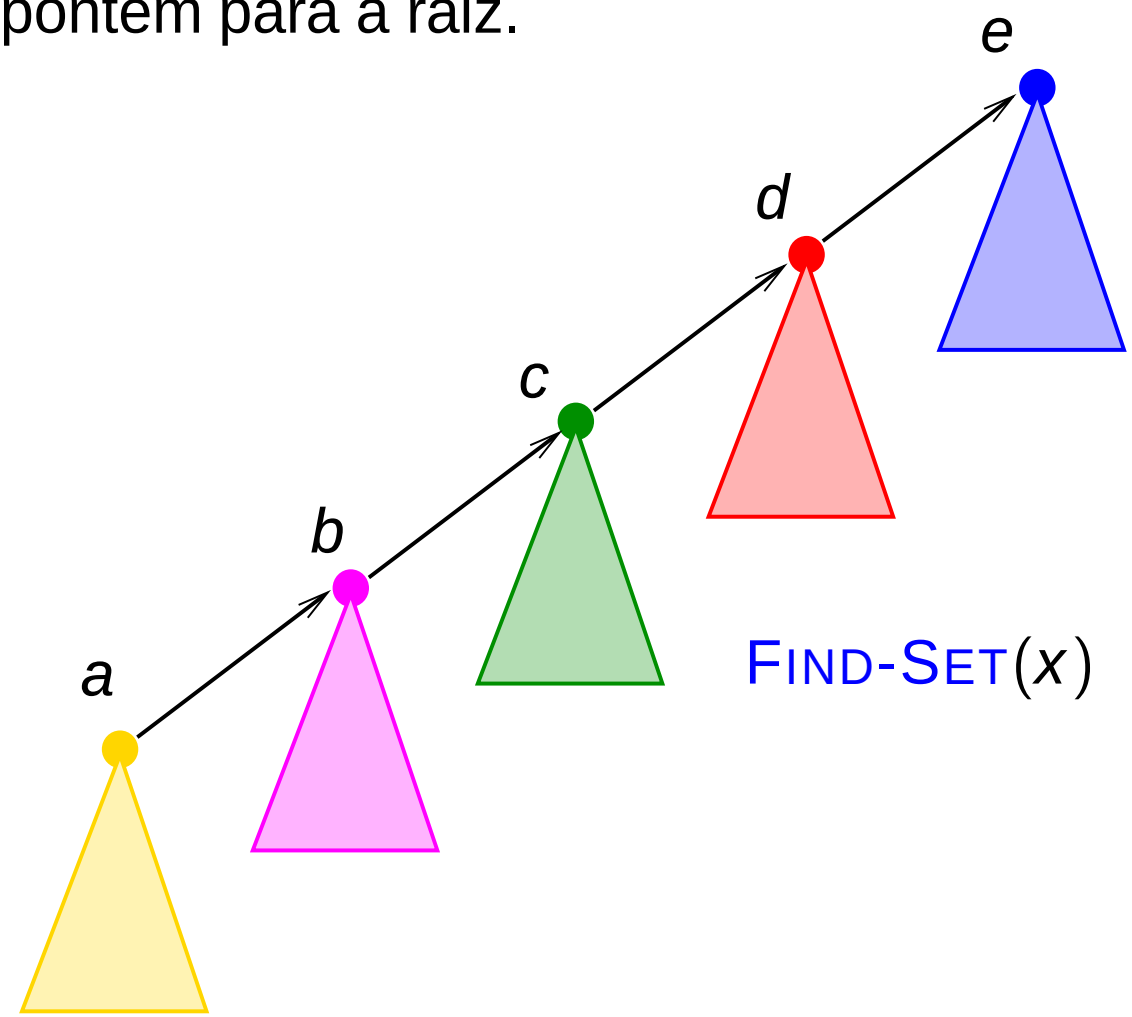
```
MAKE-SET(x)
1   pai[x] ← x
2   rank[x] ← 0

UNION(x, y)
1   LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y) ▷ x e y são raízes
1   se  rank[x] > rank[y]
2       então pai[y] ← x
3       senão pai[x] ← y
4                 se  rank[x] = rank[y]
5                     então  rank[y] ← rank[y] + 1
```

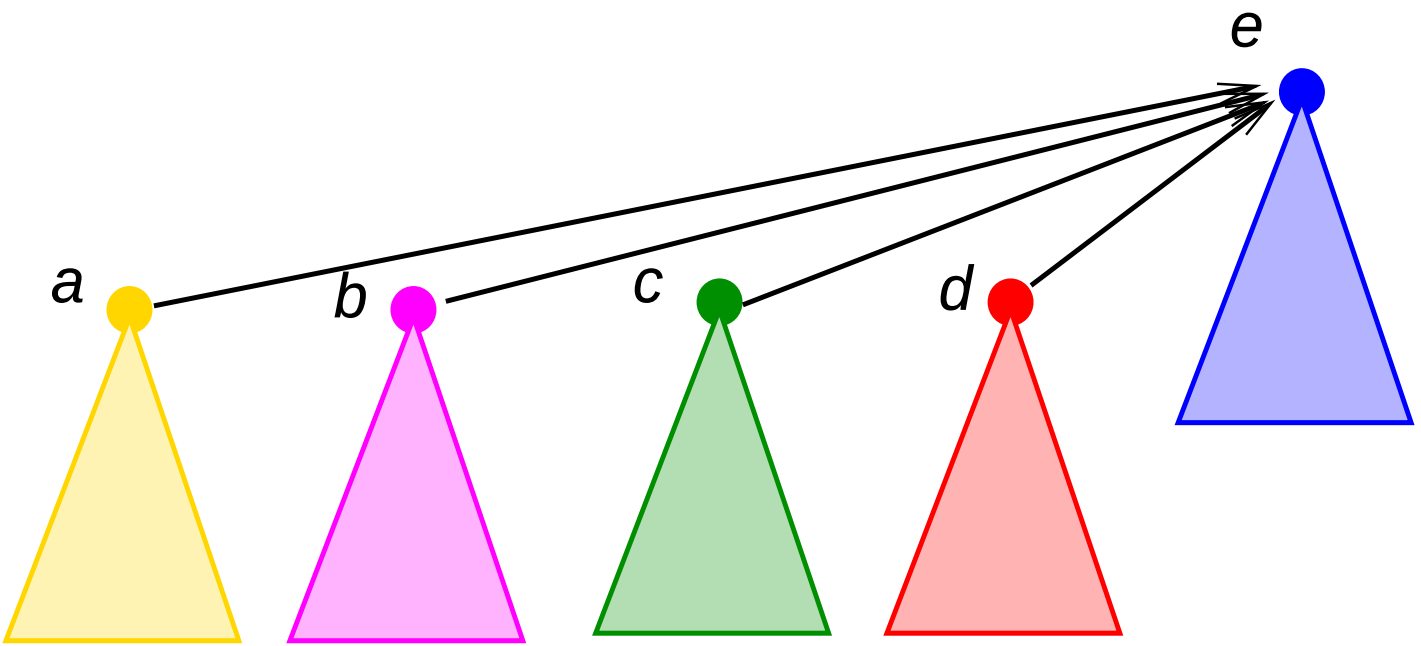
Path compression

A idéia é muito simples: ao tentar determinar o representante (raiz da árvore) de um nó fazemos com que todos os nós no caminho apontem para a raiz.



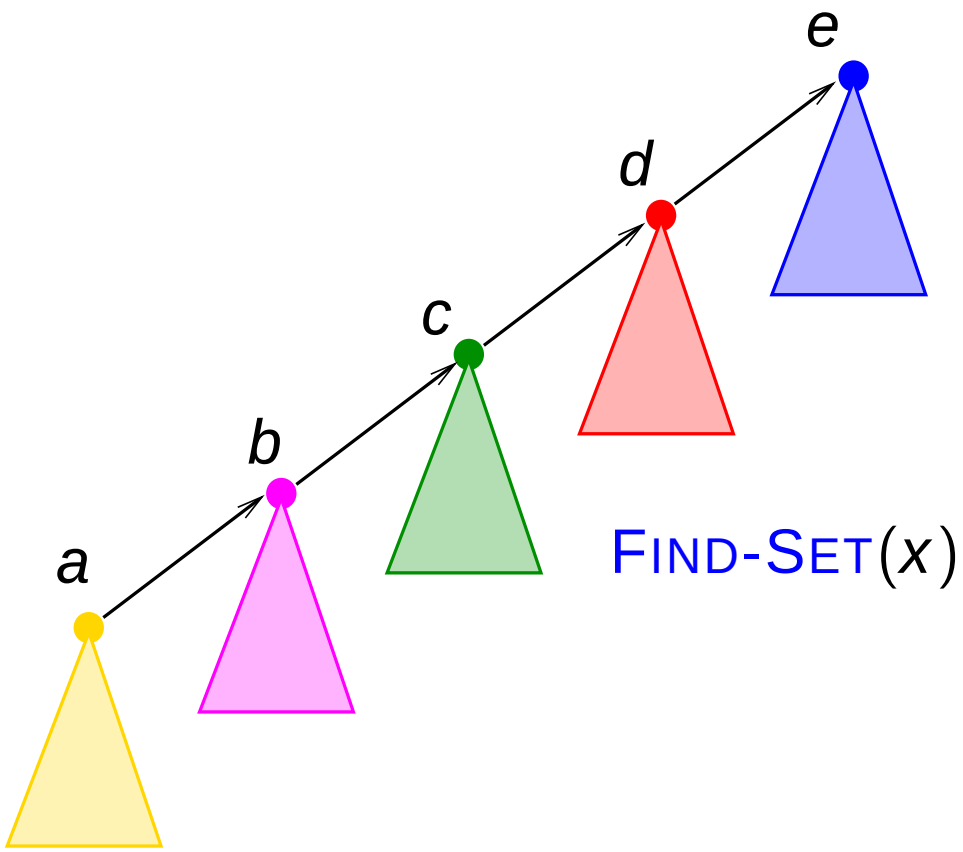
Path compression

A idéia é muito simples: ao tentar determinar o representante (raiz da árvore) de um nó fazemos com que todos os nós no caminho apontem para a raiz.



FIND-SET(x)

Path compression



FIND-SET(x)
1 **se** $x \neq \text{pai}[x]$
2 **então** $\text{pai}[x] \leftarrow$ FIND-SET($\text{pai}[x]$)
3 **devolva** $\text{pai}[x]$

Análise de union by rank com path compression

Vamos descrever (sem provar) a complexidade de uma seqüência de operações MAKE-SET, UNION e FIND-SET quando union by rank e path compression são usados.

Para $k \geq 0$ e $j \geq 1$ considere a função

$$A_k(j) = \begin{cases} j+1 & \text{se } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{se } k \geq 1, \end{cases}$$

onde $A_{k-1}^{(j+1)}(j)$ significa que $A_{k-1}(j)$ foi iterada $j+1$ vezes.

Função $A_k(j)$

- $A_{k-1}^{(j+1)}(j)$ utiliza notação de iteração funcional. O parâmetro k é o nível da função.
- Especificamente:

$$A_{k-1}^{(0)}(j) = j$$

$$A_{k-1}^{i}(j) = A_{k-1}(\ A_{k-1}^{(i-1)}(j)\ ) \qquad \text{para} \quad i \geq 1$$

- Exemplificando:
  - $A_0(1) = 1 + 1 = 2$
  - $A_1(1) = 2 \cdot 1 + 1 = 3$
  - $A_2(1) = 2^{1+1} \cdot (1 + 1) - 1 = 7$
  - $A_3(1) = A_2^{(2)}(1)$
    $= A^2(A^2(1))$
    $= A^2(7)$
    $= 2^8 \cdot 8 - 1$
    $= 2^{11} - 1$
    $= 2047$

Análise de union by rank com path compression

Ok. Você não entendeu o que esta função faz...

Tudo que você precisa saber é que ela cresce muito rápido.

$$A_0(1) = 2$$
$$A_1(1) = 3$$
$$A_2(1) = 7$$
$$A_3(1) = 2047$$
$$A_4(1) = 16^{512}$$

Em particular, $A_4(1) = 16^{512} \gg 10^{80}$ que é número estimado de átomos do universo...

Análise de union by rank com path compression

Considere agora inversa da função $A_k(n)$ definida como

$$\alpha(n) = \min\{k \ : A_k(1) \geq n\}.$$

Usando a tabela anterior temos

$$\alpha(n) = \begin{cases} 0 & \text{para } 0 \leq n \leq 2, \\ 1 & \text{para } n = 3, \\ 2 & \text{para } 4 \leq n \leq 7, \\ 3 & \text{para } 8 \leq n \leq 2047, \\ 4 & \text{para } 2048 \leq n \leq A_4(1). \end{cases}$$

Ou seja, para efeitos práticos $\alpha(n) \leq 4$.

Análise de union by rank com path compression

Teorema. Uma seqüência de m operações MAKE-SET, UNION e FIND-SET pode ser executada em uma ED para *disjoint-set forests* com union by rank e path compression em tempo $O(m\alpha(n))$ no pior caso.

Isto significa (na prática) que o tempo total é linear e que o custo amortizado por operação é uma constante.

Vamos voltar agora à implementação do algoritmo de Kruskal.

O algoritmo de Kruskal (de novo)

AGM-KRUSKAL(G, w)
1 $A \leftarrow \emptyset$
2 **para cada** $v \in V[G]$ **faça**
3 MAKE-SET(v)
4 Ordene as arestas em ordem não-decrescente de peso
5 **para cada** $(u, v) \in E$ nessa ordem **faça**
6 **se** FIND-SET$(u) \neq$ FIND-SET(v)
7 **então** $A \leftarrow A \cup \{(u, v)\}$
8 UNION(u, v)
9 **devolva** A

Complexidade:

- Ordenação: $O(E \lg E)$
- $|V|$ chamadas a MAKE-SET
- $|E| + |V| - 1 = O(E)$ chamadas a UNION e FIND-SET

O algoritmo de Kruskal (de novo)

- Ordenação: $O(E \lg E)$
- $|V|$ chamadas a MAKE-SET
- $O(E)$ chamadas a UNION e FIND-SET

Usando a representação *disjoint-set forests* com union by rank e path compression, o tempo gasto com as operações é $O((V + E)\alpha(V)) = O(E\,\alpha(V))$.

Como $\alpha(V) = O(\lg V) = O(\lg E)$ o passo que consome mais tempo no algoritmo de Kruskal é a ordenação.

Logo, a complexidade do algoritmo é $O(E \lg E) = O(E \lg V)$.