Projeto e Análise de Algoritmos

A. G. Silva

Baseado nos materiais de
Roman – USP
Souza, Silva, Lee, Rezende, Miyazawa – Unicamp
Monteforte – UFABC

18 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)*.

Algoritmos gulosos: conceitos básicos

- São aqueles que, a cada decisão:
  - Sempre escolhem a alternativa que parece mais promissora naquele instante: critério guloso – decisão localmente ótima!
  - Nunca reconsideram essa decisão
    - Uma escolha que foi feita nunca é revista
    - Não há *backtracking*

Árvore Geradora Mínima - Algoritmo de Prim (revisão)

Árvore Geradora Mínima

Problema da Árvore Geradora Mínima

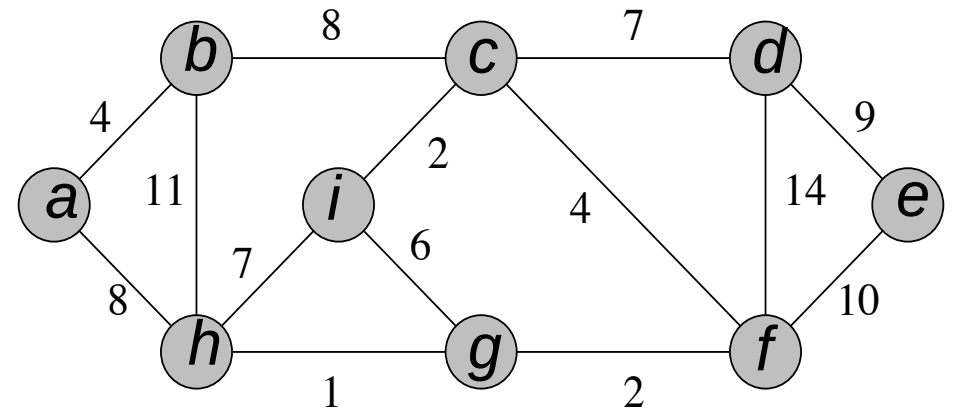
Entrada: grafo conexo $G = (V, E)$ com pesos $w(u, v)$ para cada aresta (u, v).

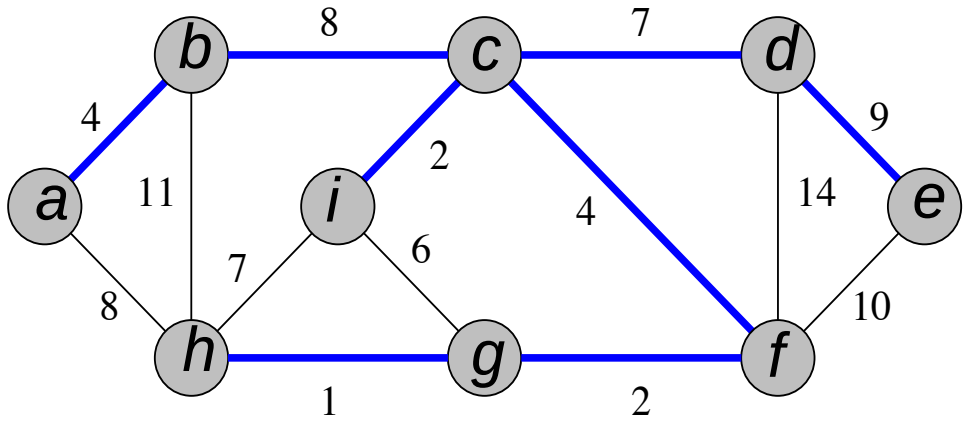
Saída: subgrafo gerador conexo T de G cujo peso total

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

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.

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)
```

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 $O(V \lg V + E \lg V) = O(E \lg V)$.

Árvore Geradora Mínima - Algoritmo de Kruskal

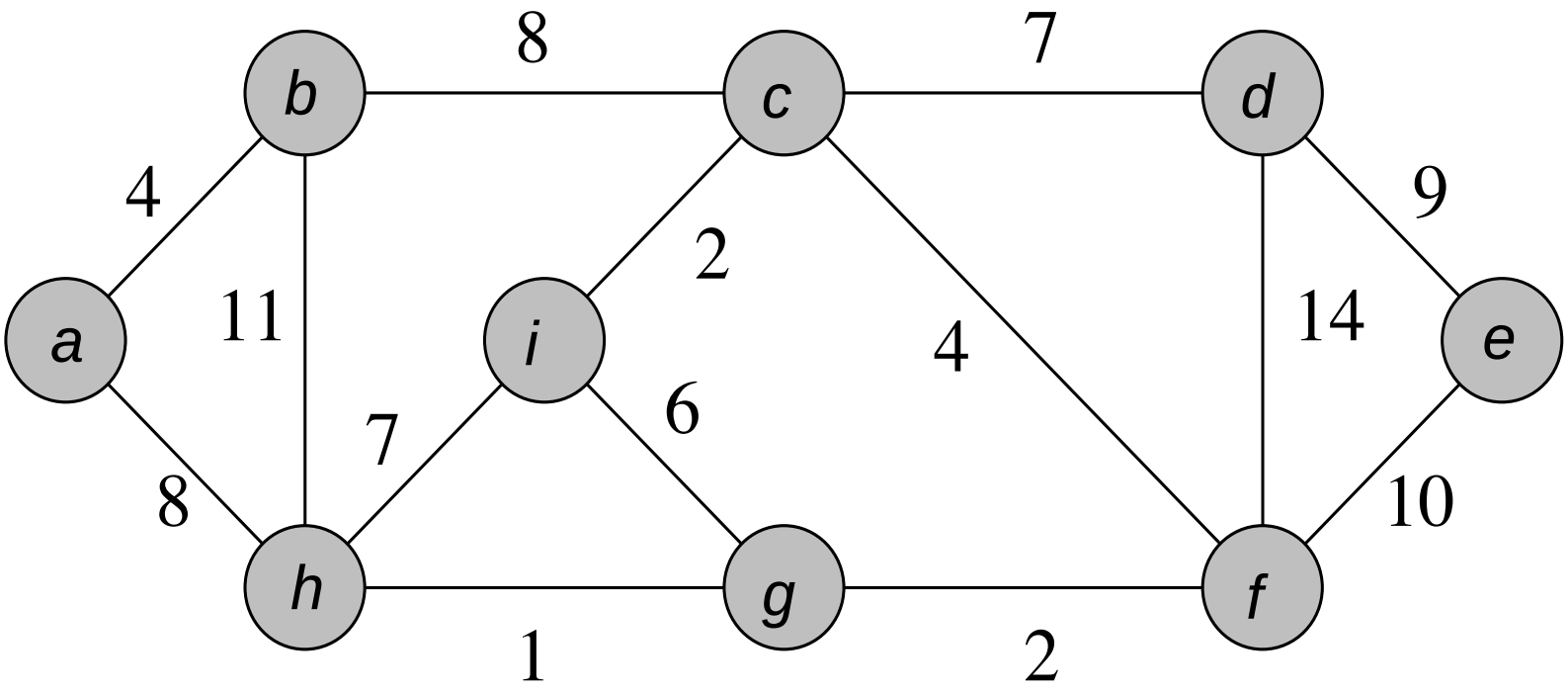
O algoritmo de Kruskal

- 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 $F = (V, A)$.

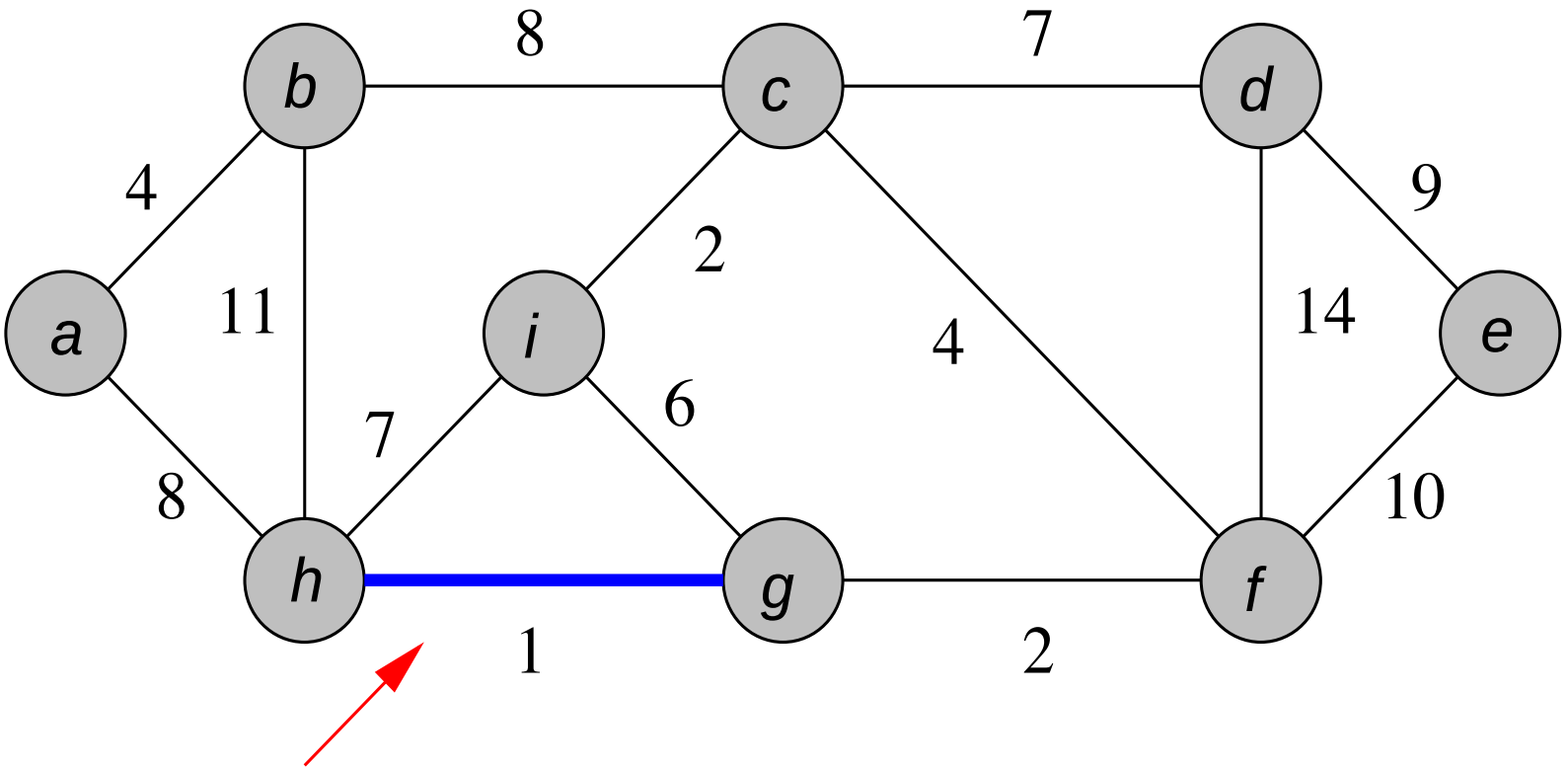
  Note que (u, v) é uma aresta leve do corte $\delta(C)$.
- Ele acrescenta (u, v) ao conjunto A e começa outra iteração até que A seja uma árvore geradora.

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

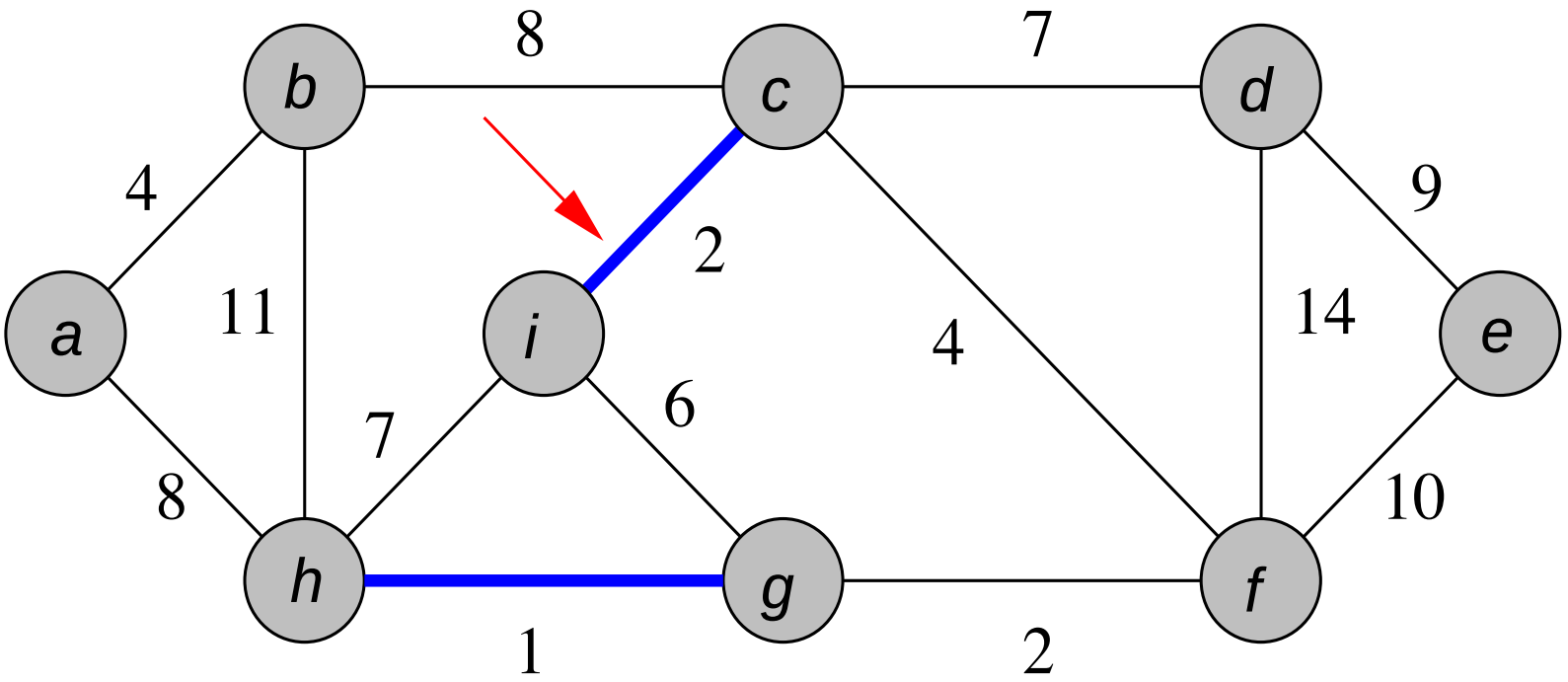
O algoritmo de Kruskal



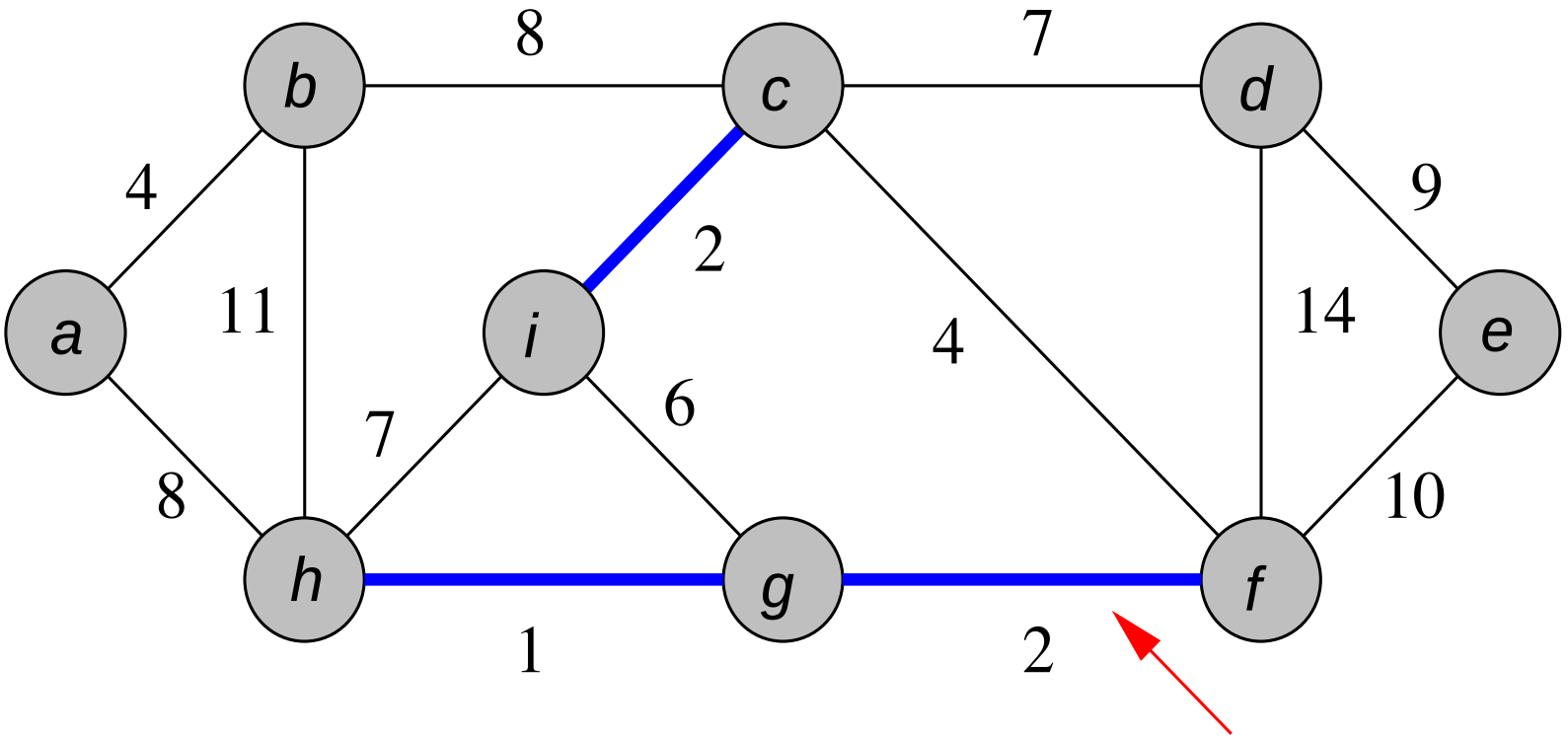
O algoritmo de Kruskal



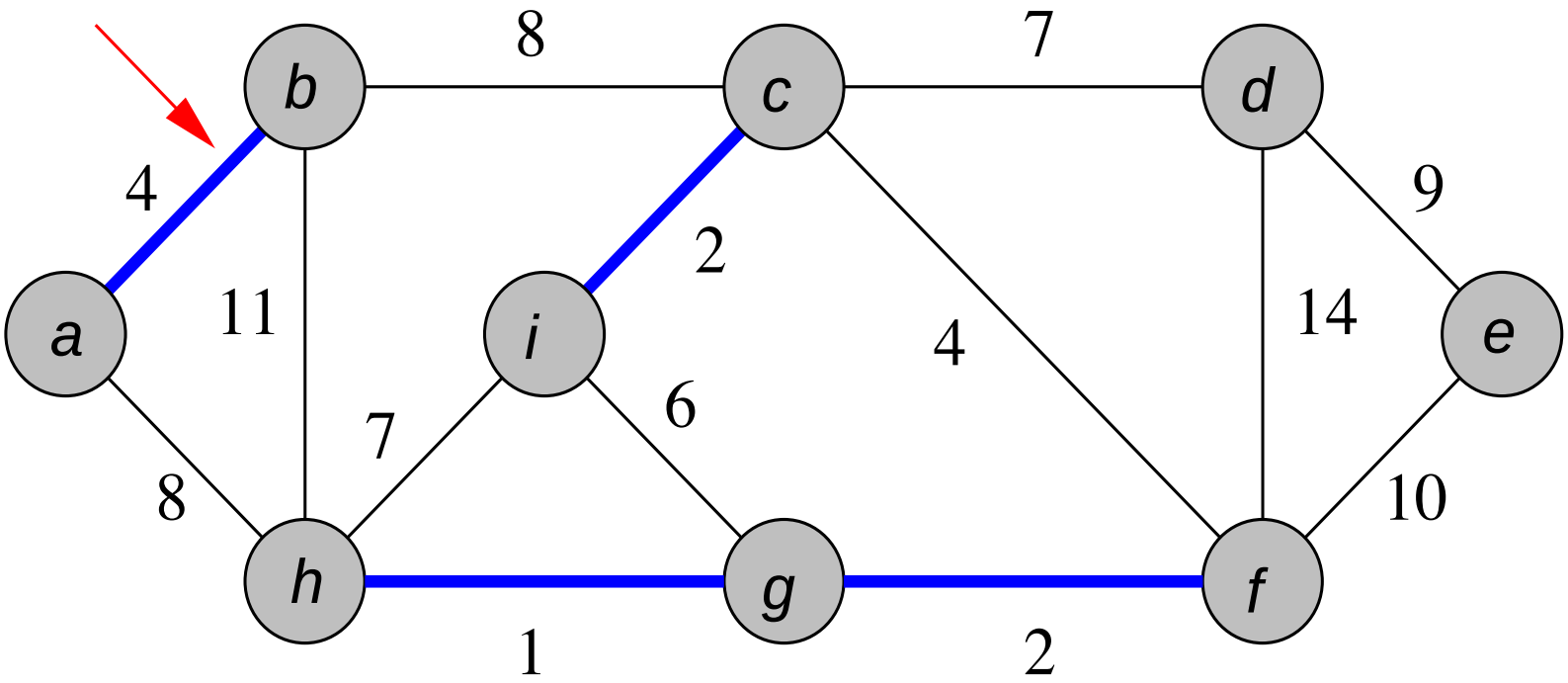
O algoritmo de Kruskal



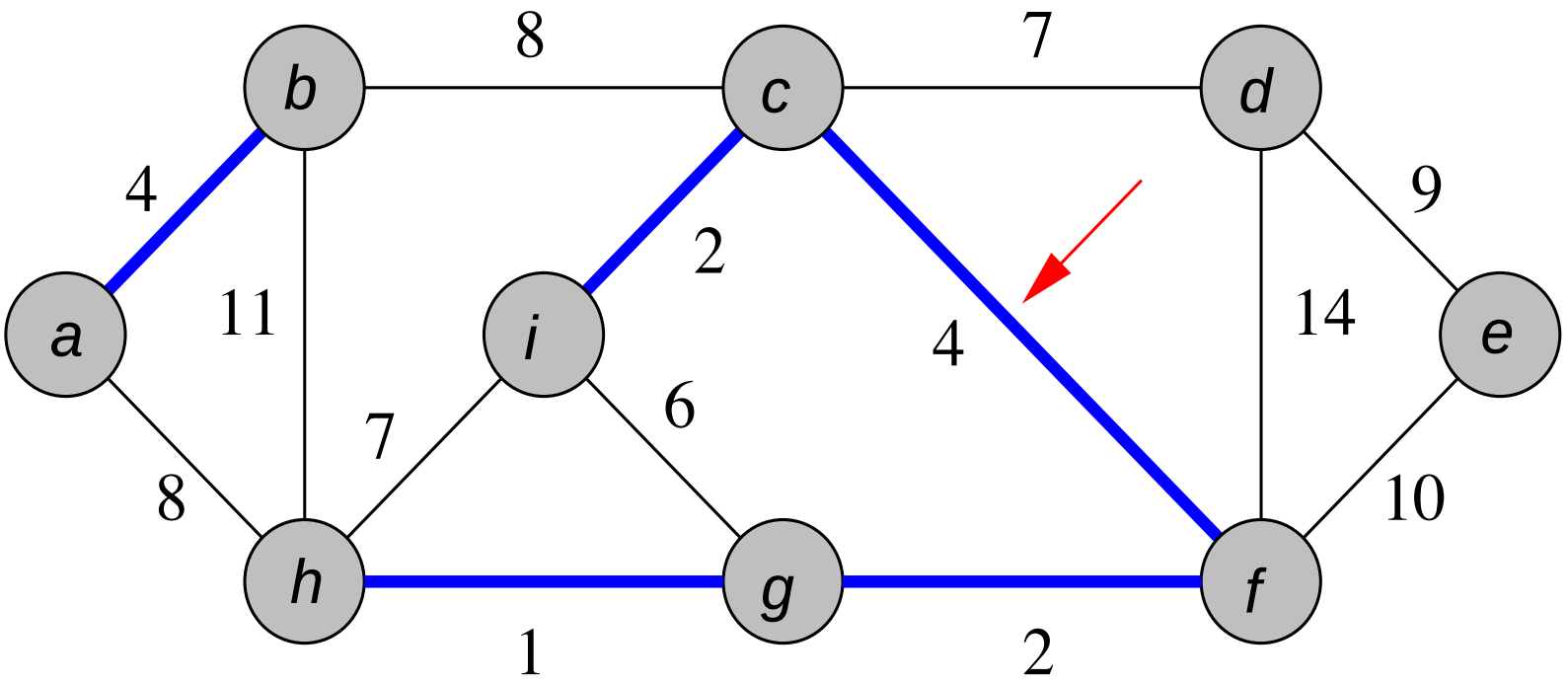
O algoritmo de Kruskal



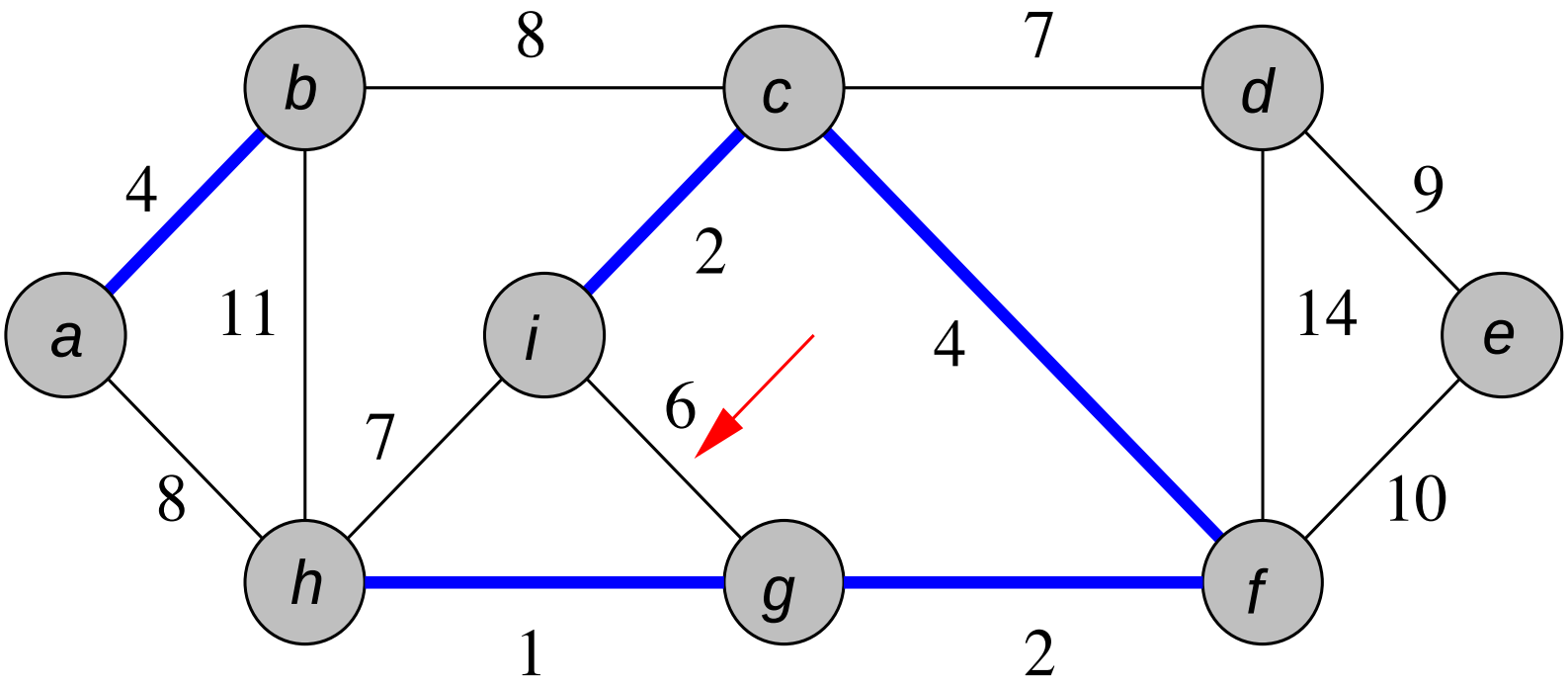
O algoritmo de Kruskal



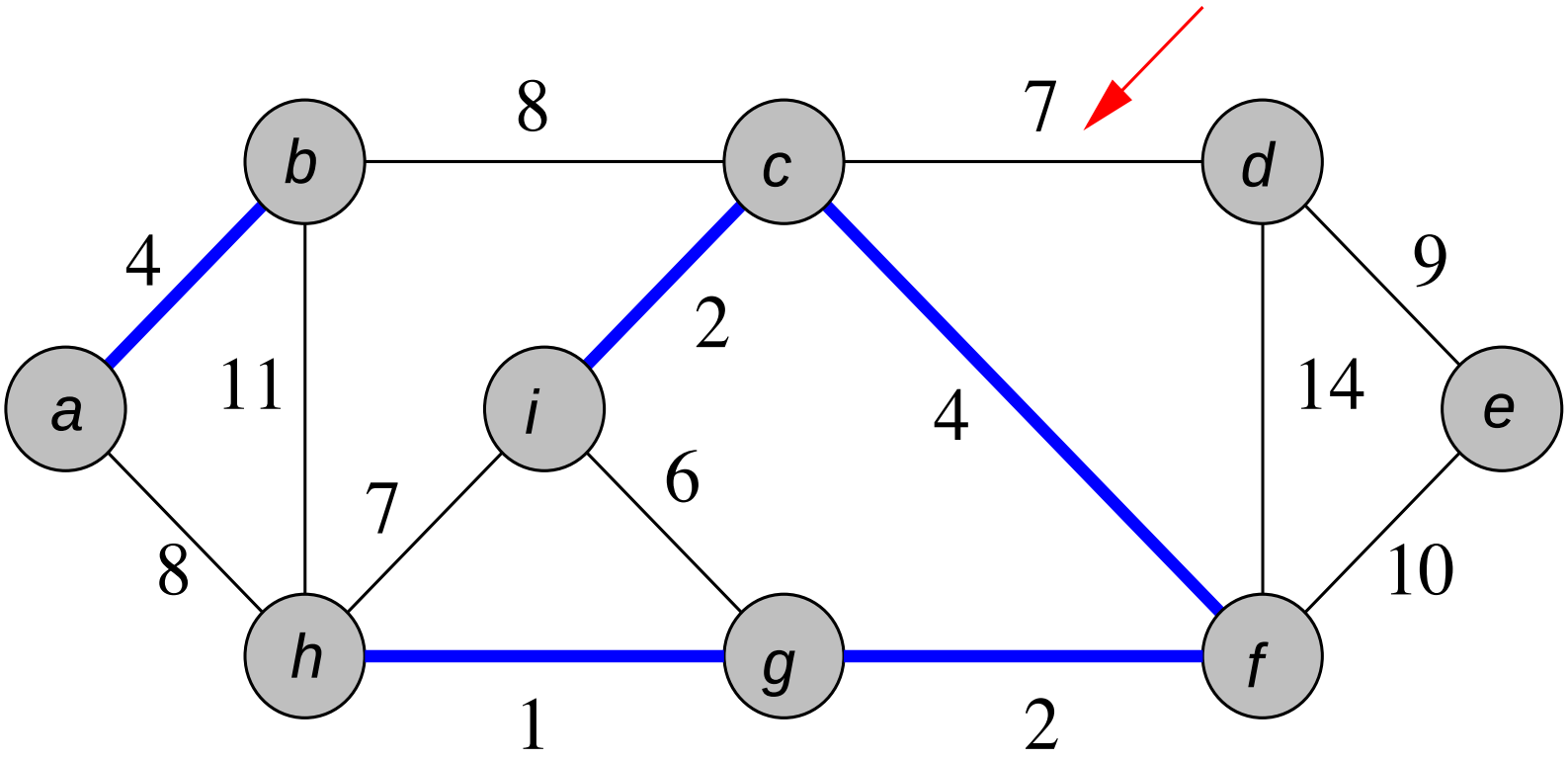
O algoritmo de Kruskal



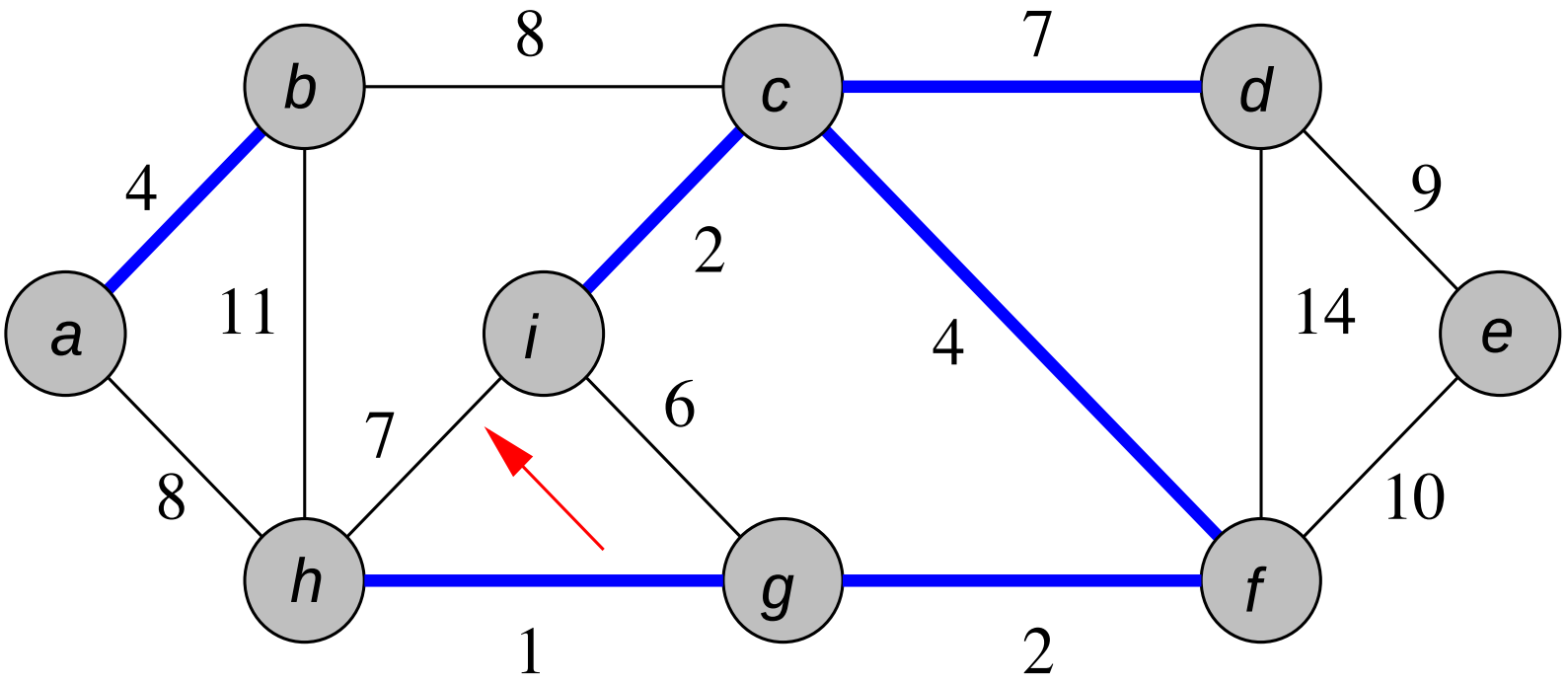
O algoritmo de Kruskal



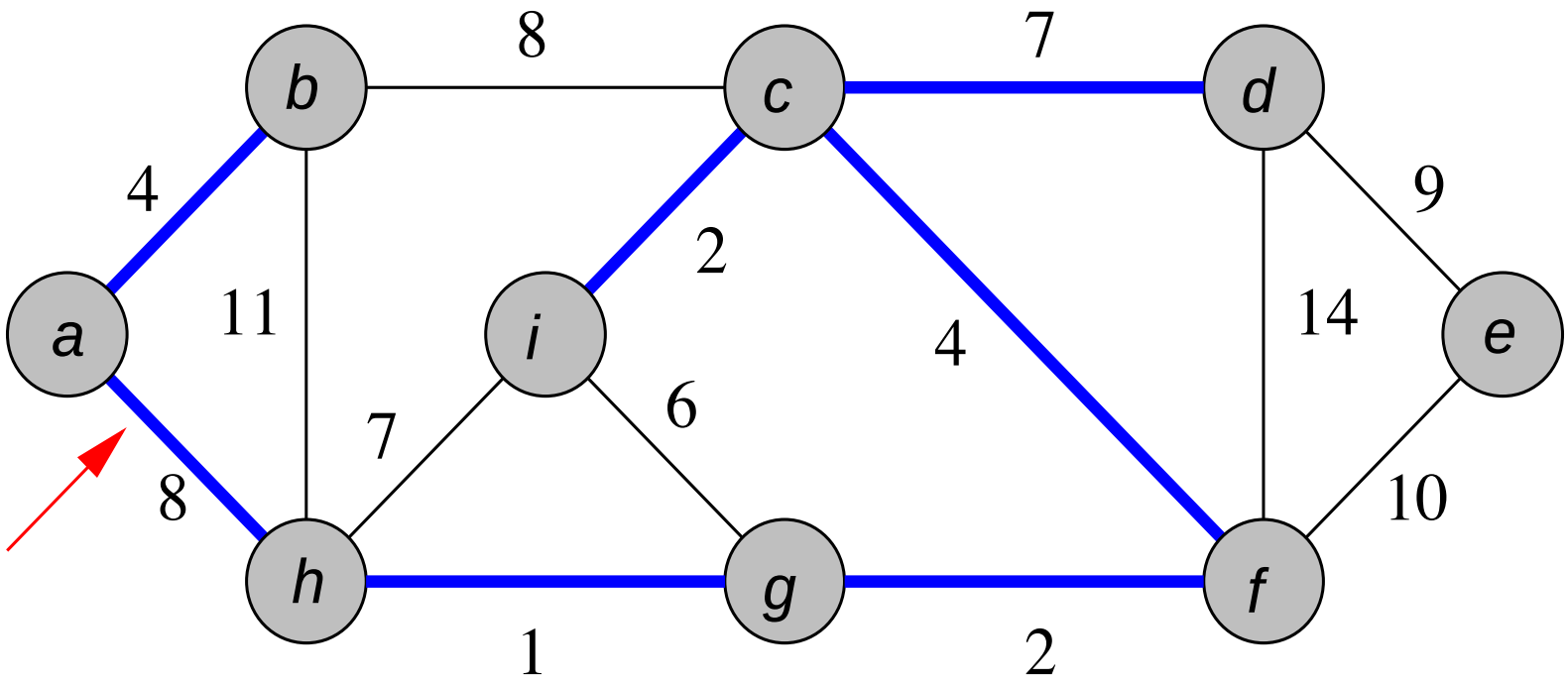
O algoritmo de Kruskal



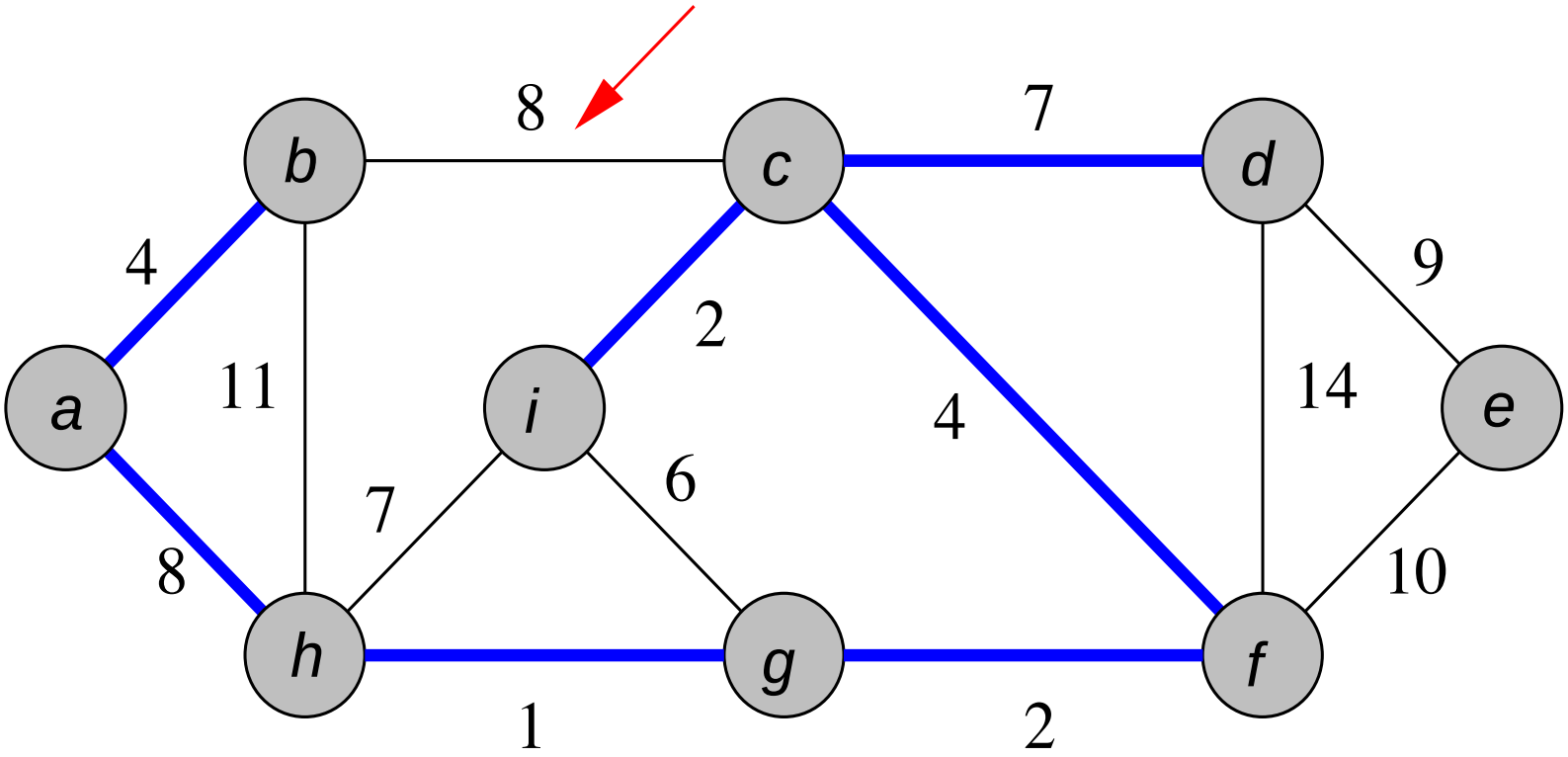
O algoritmo de Kruskal



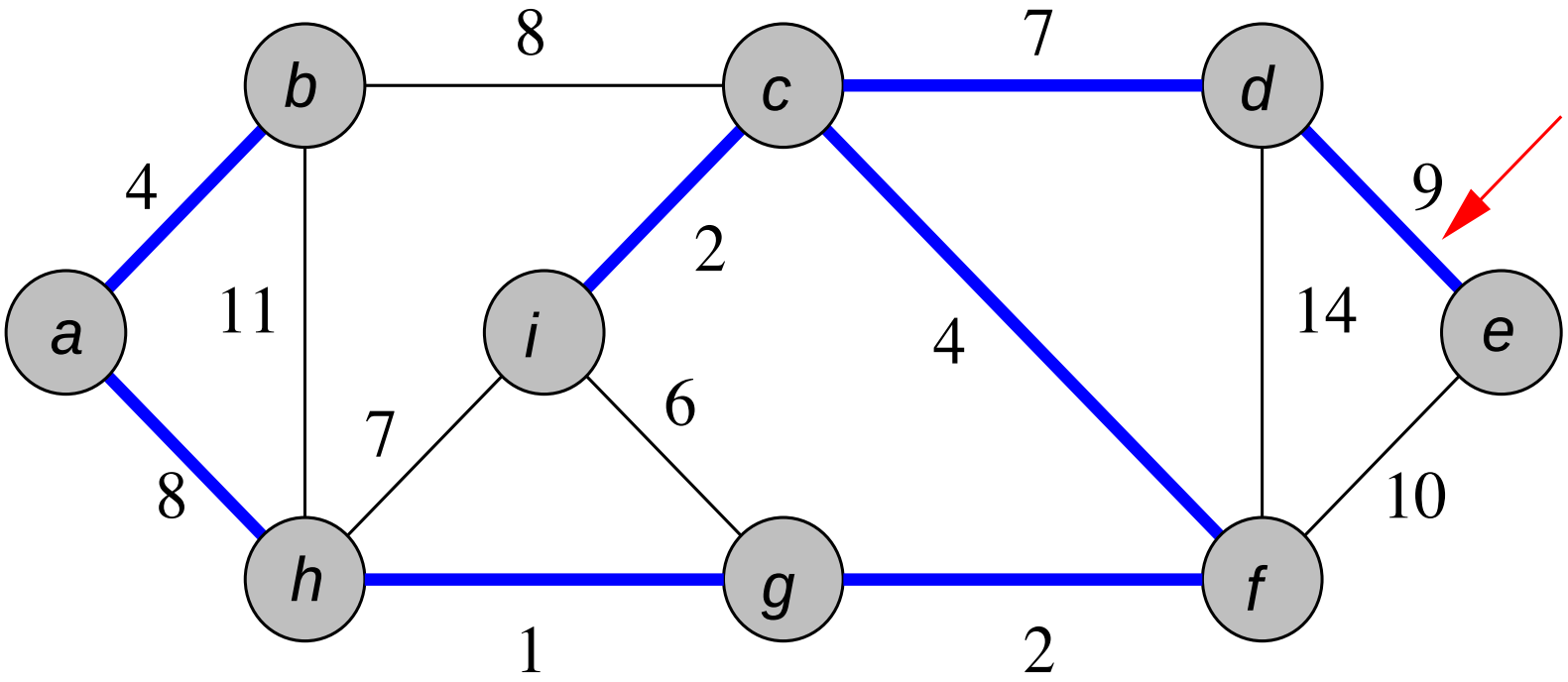
O algoritmo de Kruskal



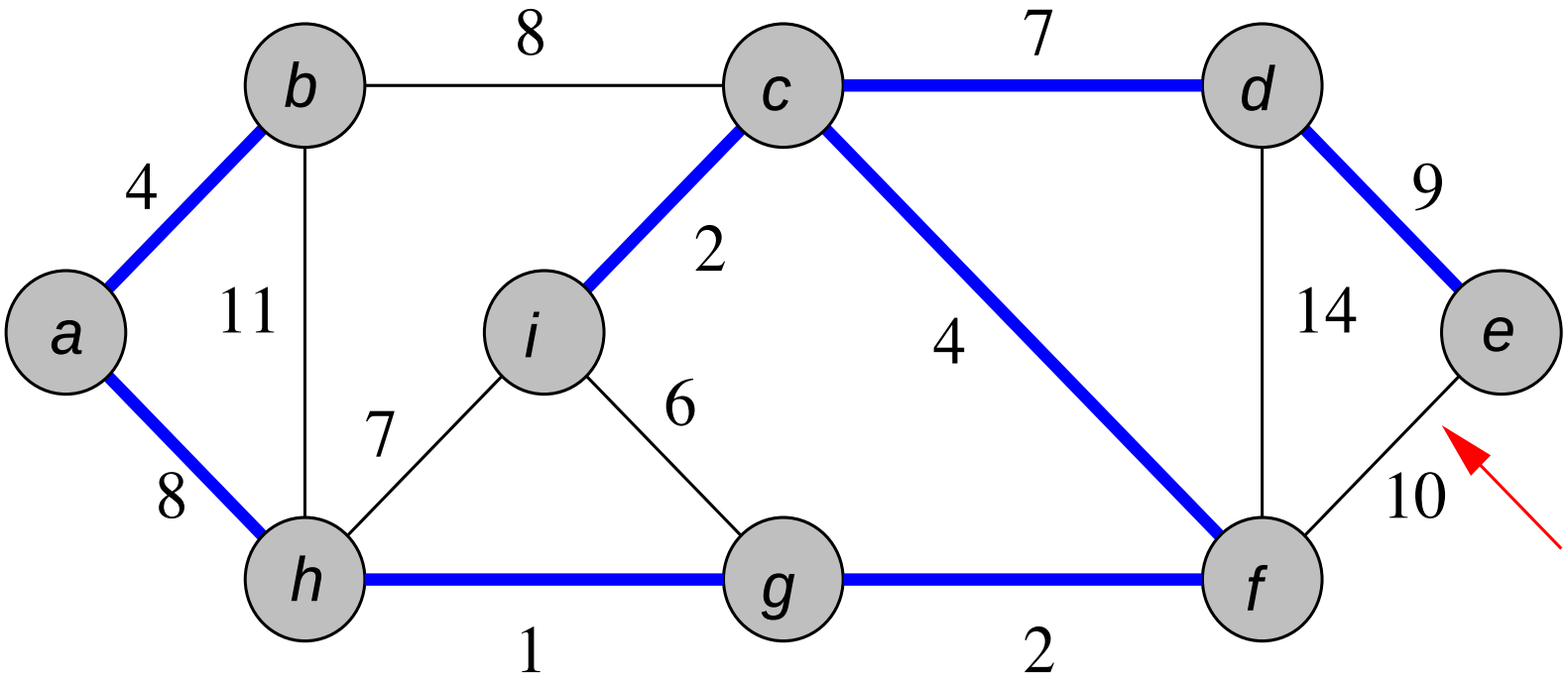
O algoritmo de Kruskal



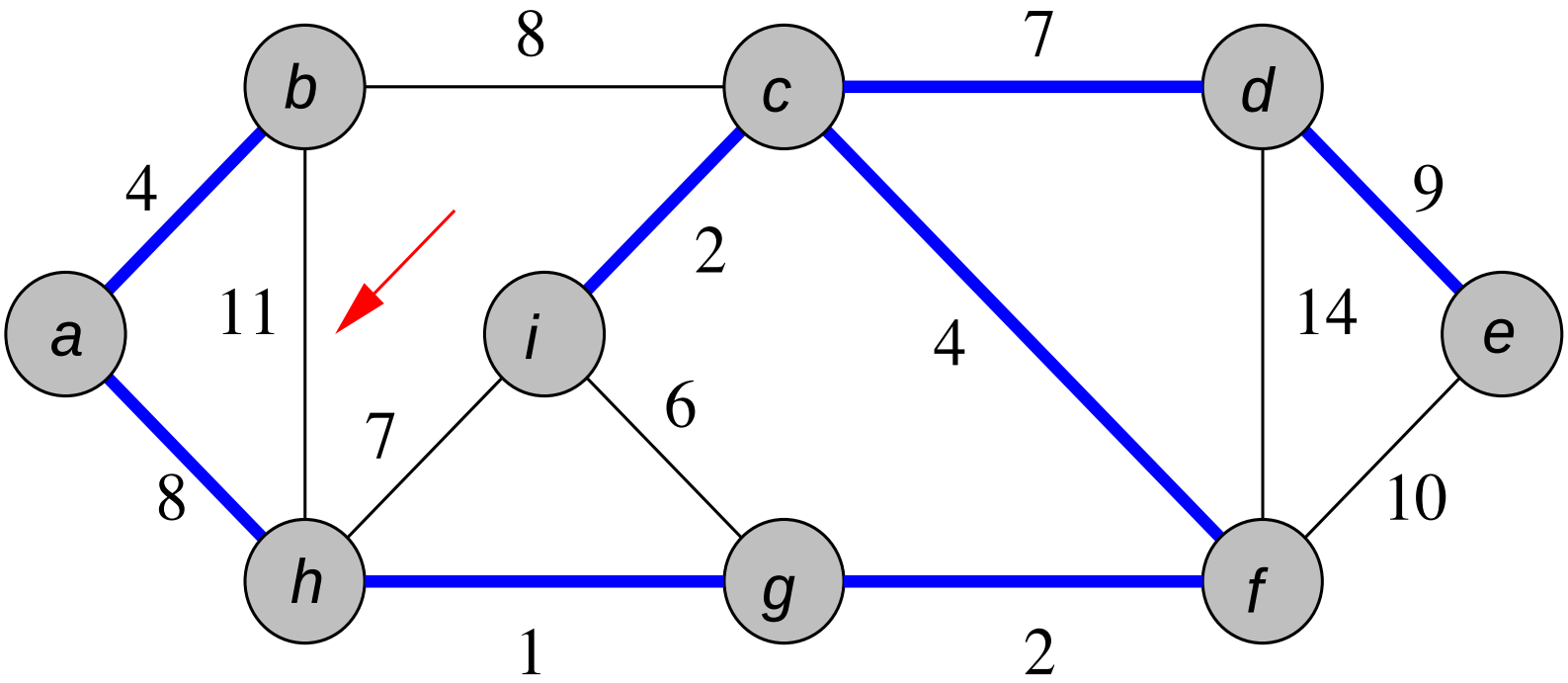
O algoritmo de Kruskal



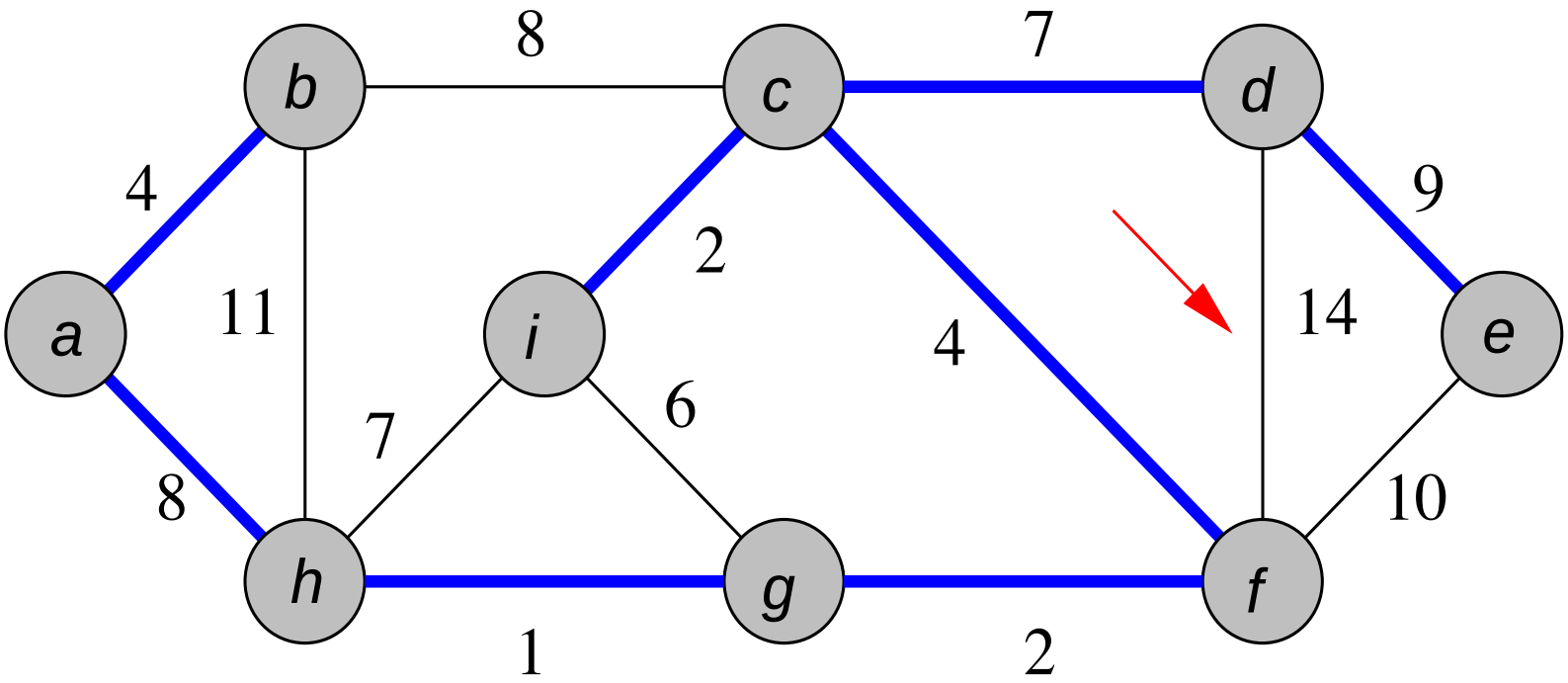
O algoritmo de Kruskal



O algoritmo de Kruskal



O algoritmo de Kruskal



O algoritmo de Kruskal

Eis uma versão 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

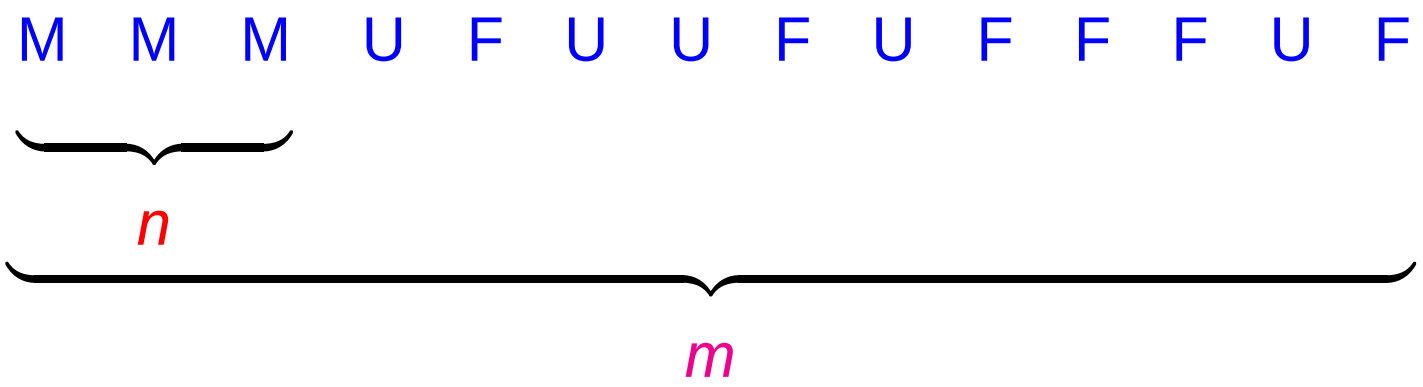
"Complexidade" de AGM-KRUSKAL

- 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

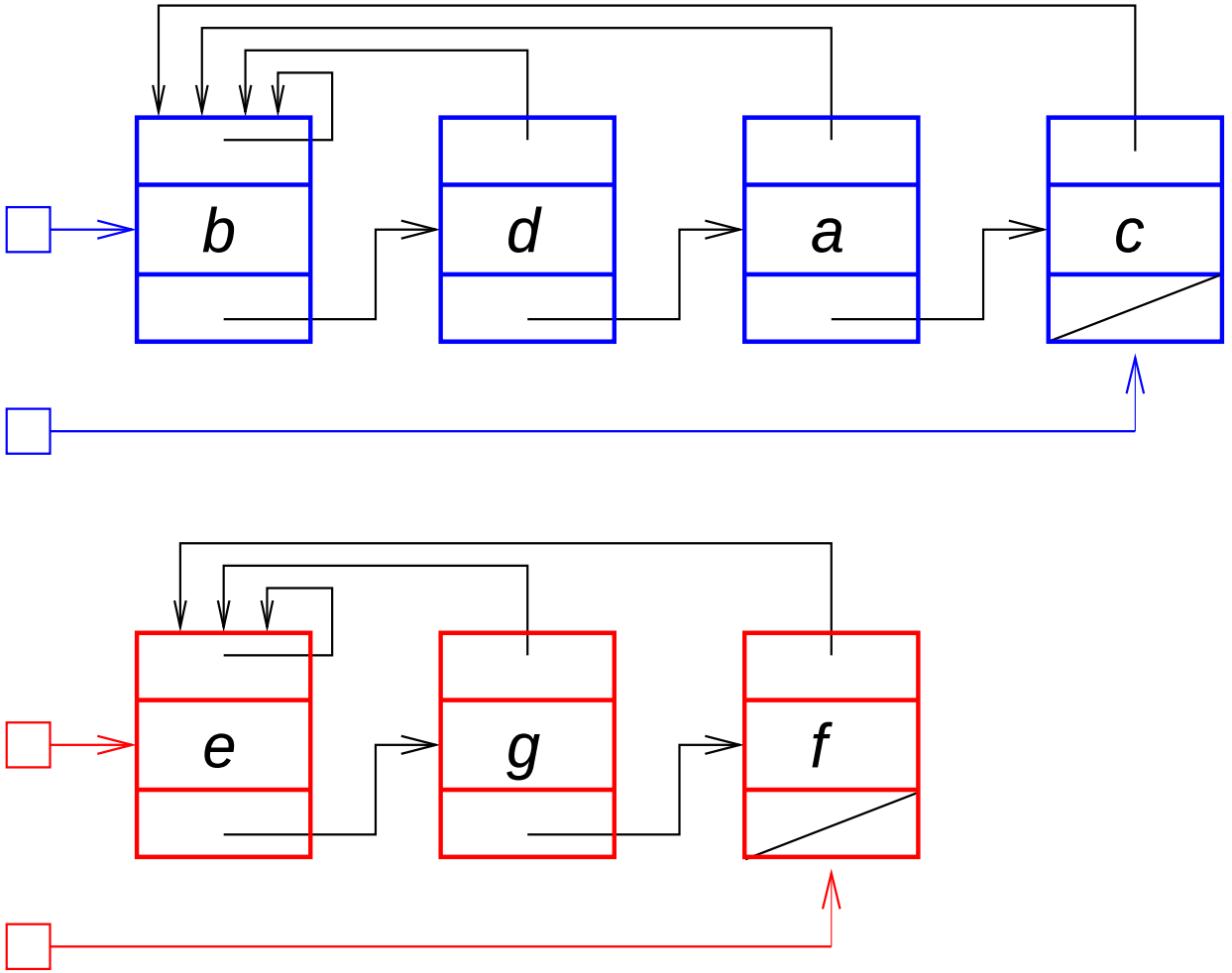
Seqüência de operações MAKE-SET, UNION e FIND-SET



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

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

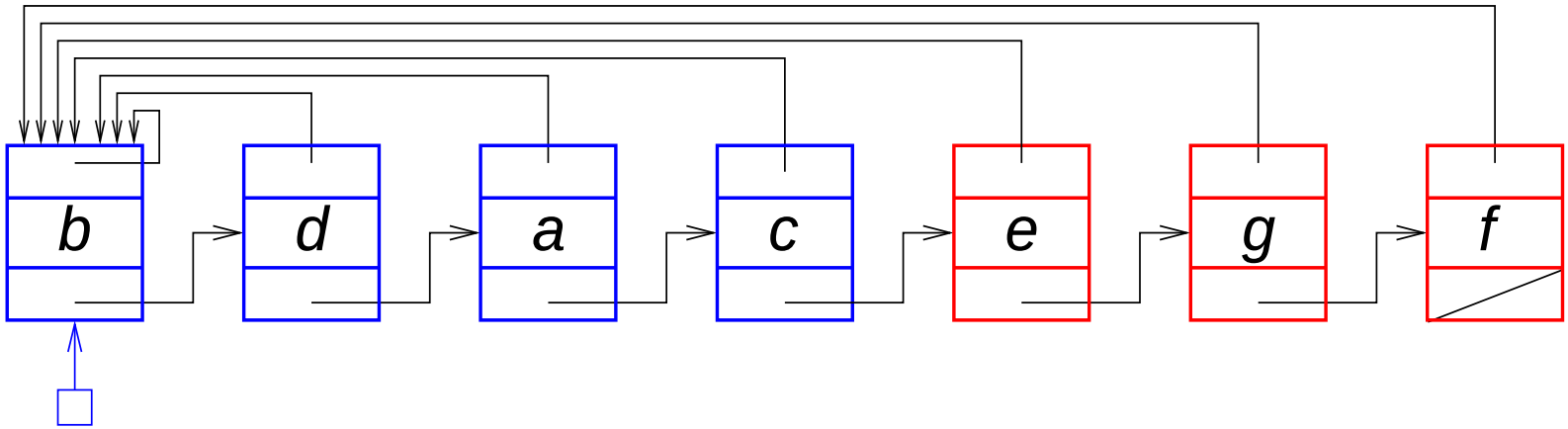
Representação por listas ligadas



- 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.

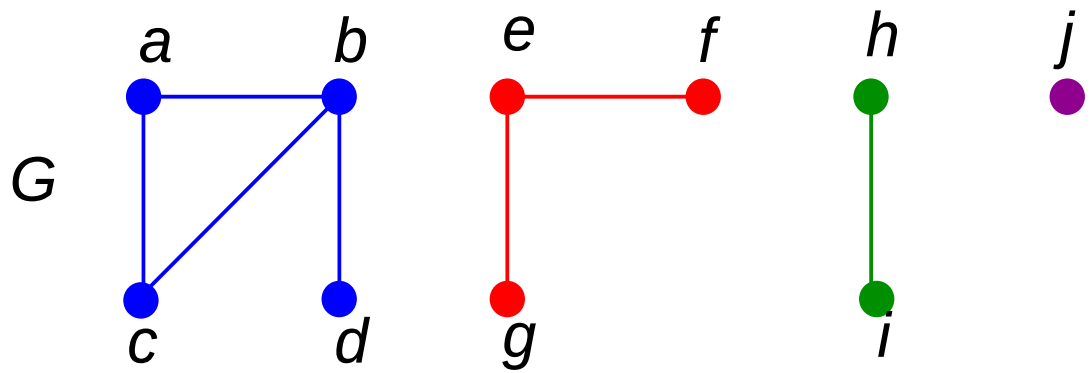
Após ser atualizado $\lceil \lg k \rceil$ vezes, a lista tem tamanho pelo menos k. Como k tem que ser menor que n, cada apontador é atualizado no máximo $O(\lg n)$ vezes.

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.

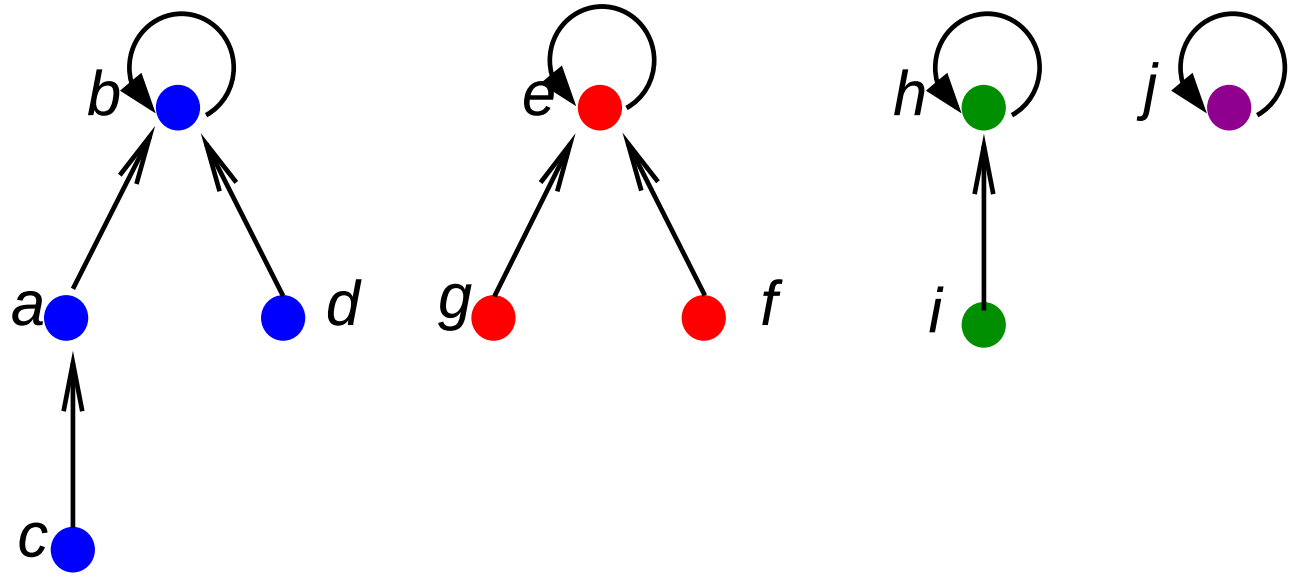
Representação por *disjoint-set forests*



Grafo com vários componentes.

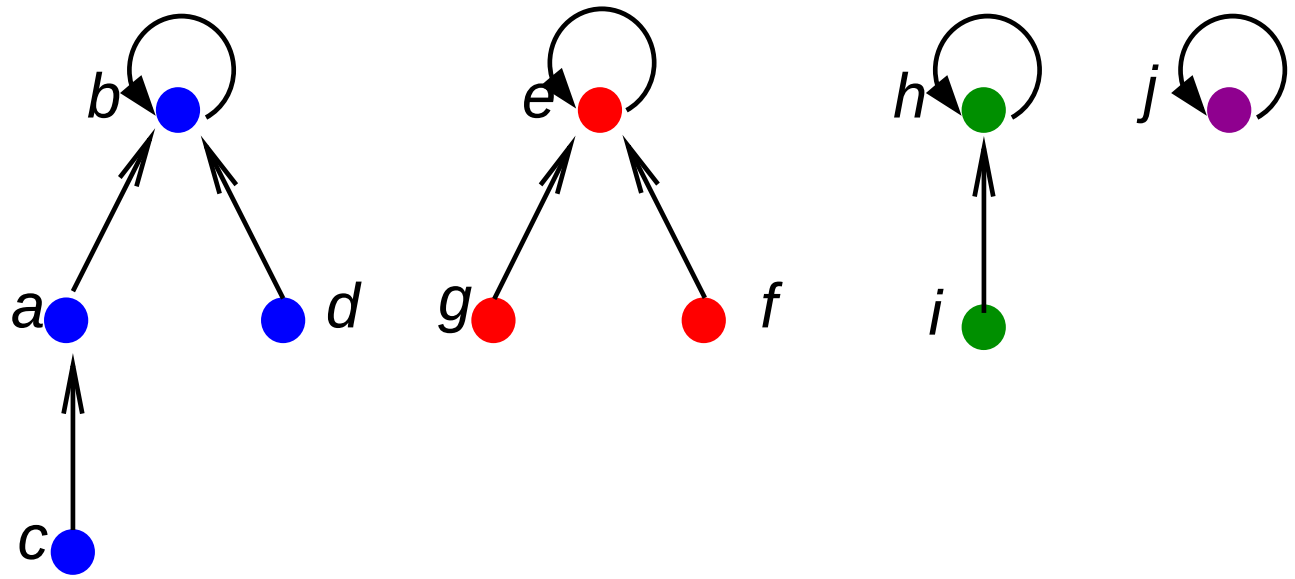
Como é a representação dos componentes na estrutura de dados *disjoint-set forests*?

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.

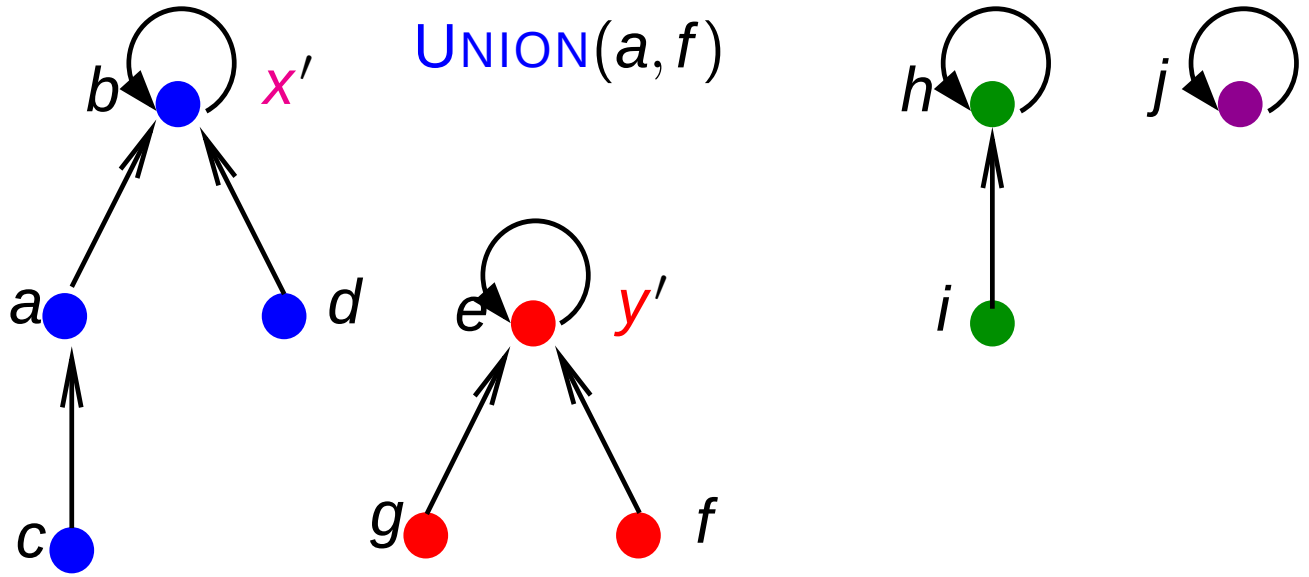
Representação por *disjoint-set forests*



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

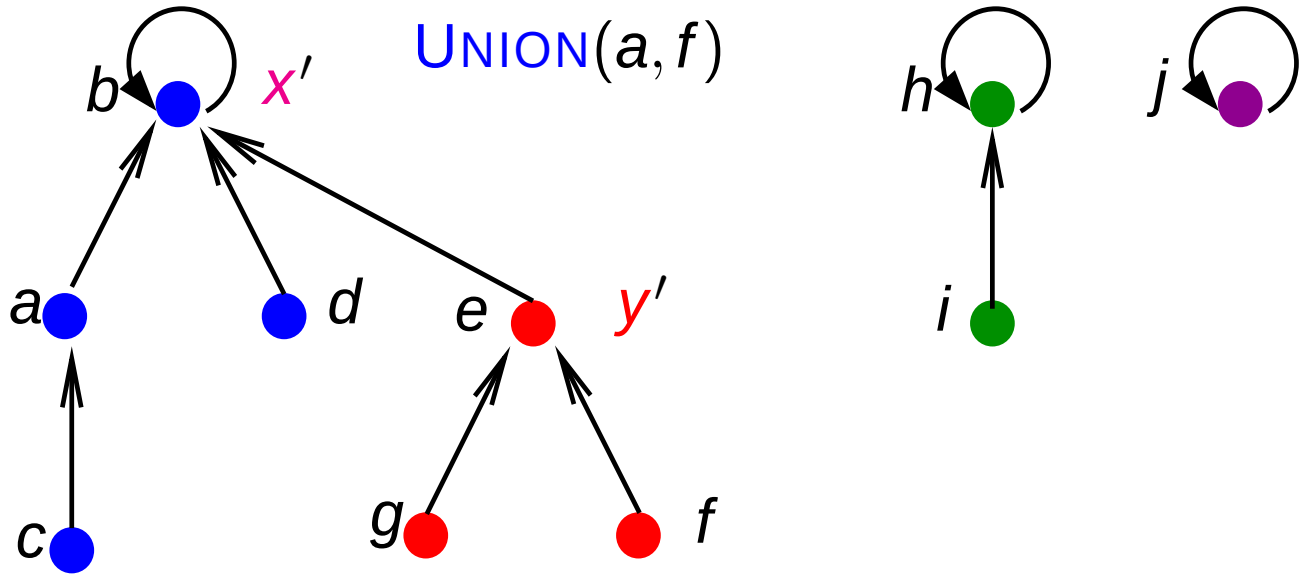
FIND-SET(x)
1 **se** $x = \text{pai}[x]$
2 $\quad$ **então devolva** x
3 $\quad$ **senão devolva** FIND-SET($\text{pai}[x]$)

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

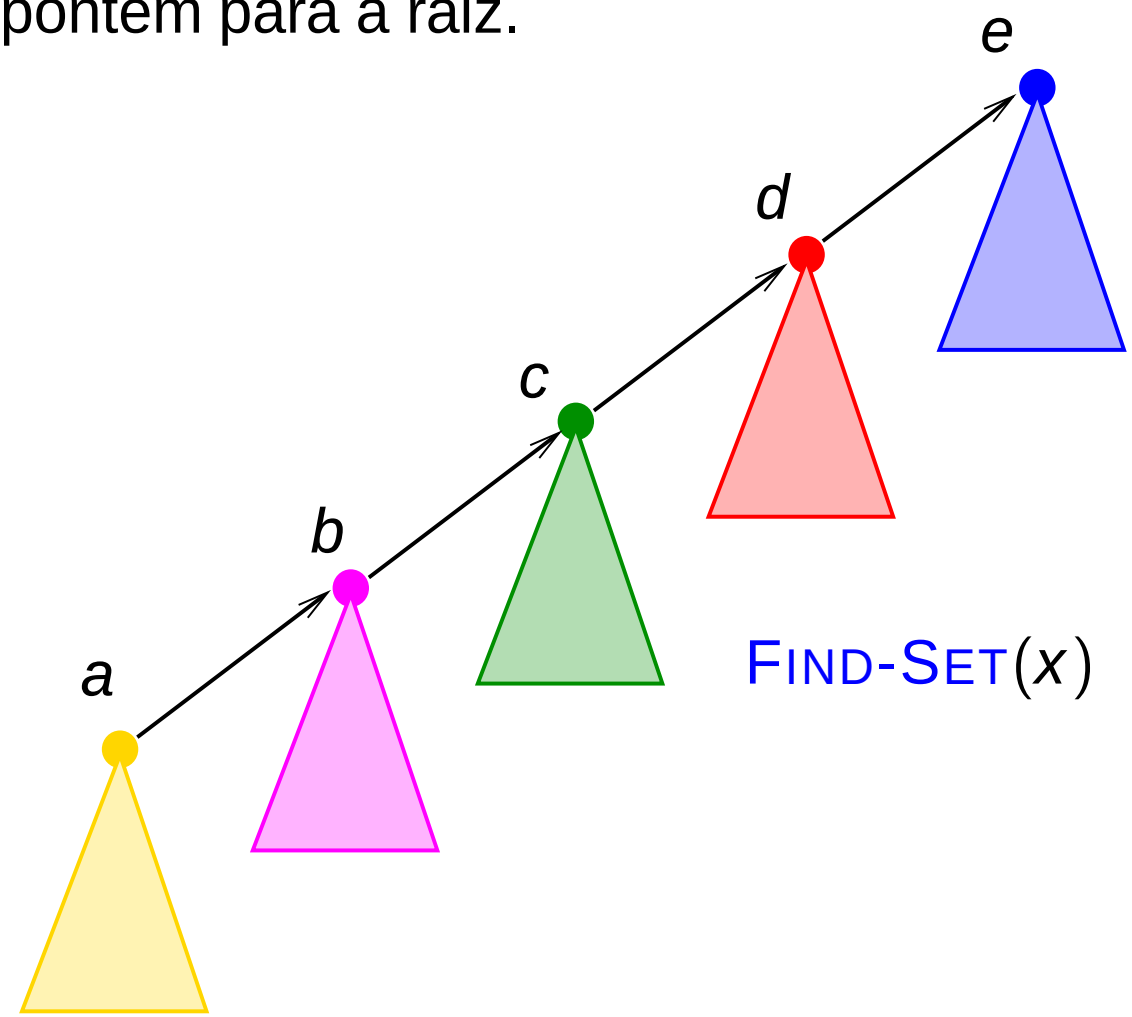
```
MAKE-SET(x)
1   pai[x] ← x
2   rank[x] ← 0

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
```

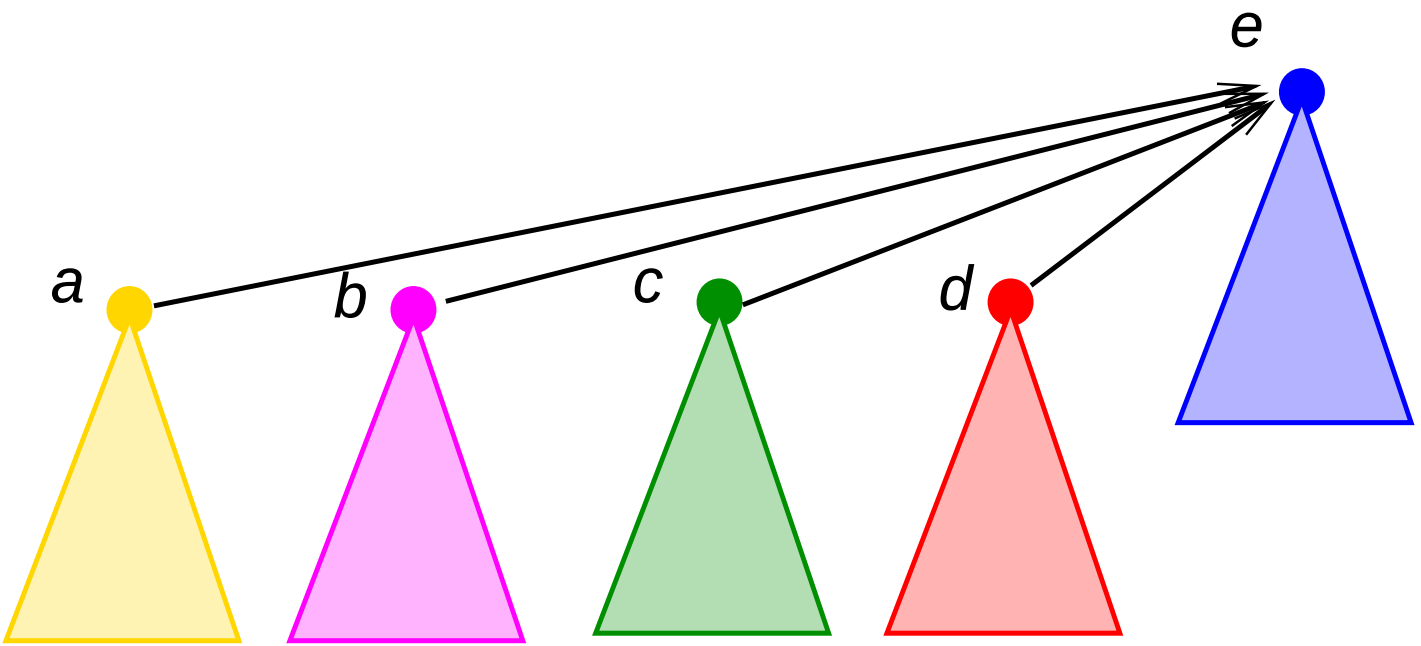
Path compression

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.



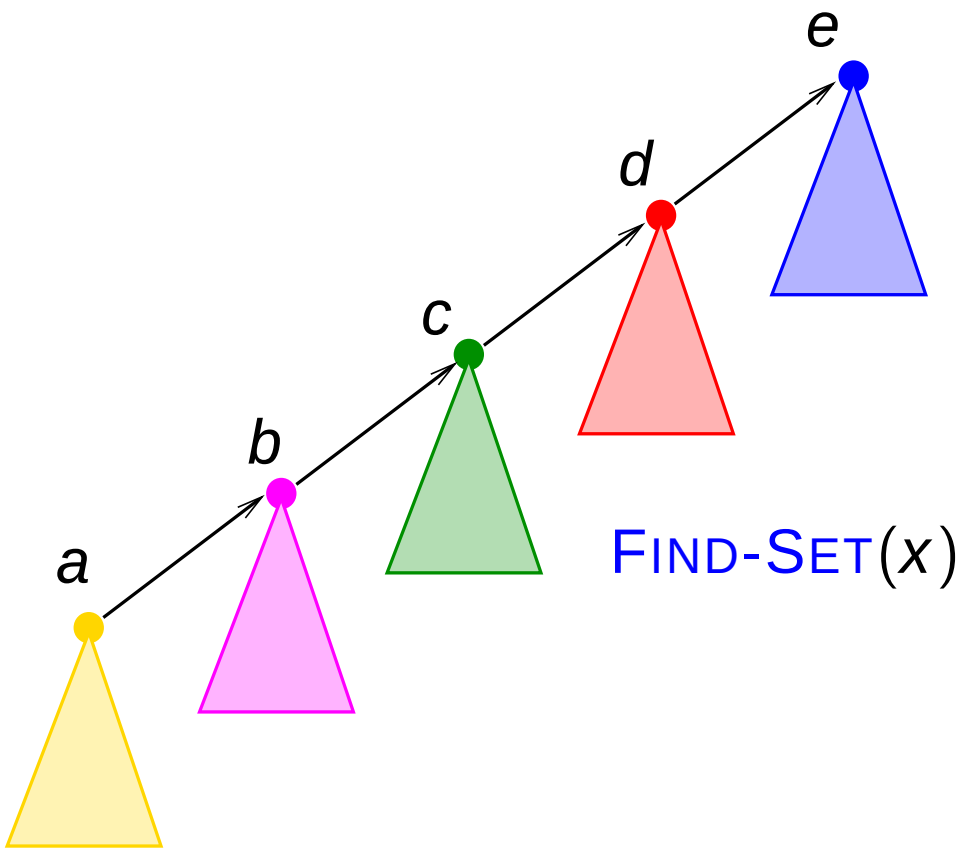
Path compression

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
- $2|E| + |V| - 1 = O(E)$ 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 $\alpha(V) \leq 4$ o passo que consome mais tempo no algoritmo de Kruskal e a ordenação.

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

Algoritmos gulosos – seleção de atividades

Seleção de Atividades

- $S = \{a_1, \ldots, a_n\}$: conjunto de n atividades que podem ser executadas em um mesmo local. Exemplo: palestras em um auditório.
- Para todo $i = 1, \ldots, n$, a atividade a_i começa no instante s_i e termina no instante f_i, com $0 \leq s_i < f_i < \infty$.
  Ou seja, supõe-se que a atividade a_i será executada no intervalo de tempo (semi-aberto) $[s_i, f_i)$.

Definição

As atividades a_i e a_j são ditas **compatíveis** se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ são disjuntos.

Problema de Seleção de Atividades

Encontre em S um subconjunto de atividades mutuamente compatíveis que tenha tamanho **máximo**.

Seleção de Atividades

- Exemplo:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	4	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- Pares de atividades incompatíveis: (a_1, a_2), (a_1, a_3)
  Pares de atividades compatíveis: (a_1, a_4), (a_4, a_8)
- Conjunto **maximal** de atividades compatíveis: (a_3, a_9, a_{11}).
- Conjunto **máximo** de atividades compatíveis: (a_1, a_4, a_8, a_{11}).

As atividades estão ordenadas em ordem crescente de instantes de término! Isso será importante mais adiante.

Seleção de Atividades

Suponha que $f_1 \leq f_2 \leq \ldots \leq f_n$, ou seja, as atividades estão ordenadas em ordem crescente de instantes de término.

Definição

Denote $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$, ou seja, S_{ij} é o conjunto de atividades que começam depois do término de a_i e terminam antes do início de a_j.

- Atividades artificiais: a_0 com $f_0 = 0$ e a_{n+1} com $s_{n+1} = \infty$
- Tem-se que $S = S_{0,n+1}$ e, com isso, S_{ij} está bem definido para qualquer par (i, j) tal que $0 \leq i, j \leq n + 1$.
- Note que $S_{ij} = \emptyset$ para todo $i \geq j$.
  Por quê?

Seleção de Atividades

- **Subestrutura ótima**: considere o *subproblema* da seleção de atividades definido sobre S_{ij}. Suponha que a_k pertence a uma solução ótima de S_{ij}.

  Como $f_i \leq s_k < f_k \leq s_j$, uma solução ótima para S_{ij} que contenha a_k será composta pelas atividades de uma solução ótima de S_{ik}, pelas atividades de uma solução ótima de S_{kj} e por a_k.
  Por quê?

Seleção de Atividades

- **Definição**: para todo $0 \leq i, j \leq n+1$, seja $c[i,j]$ o valor ótimo do problema de seleção de atividades para a instância S_{ij}.
- Deste modo, o valor ótimo do problema de seleção de atividades para instância $S = S_{0,n+1}$ é $c[0, n+1]$.
- **Fórmula de recorrência**:

$$c[i,j] = \begin{cases} 0 & \text{se } S_{ij} = \emptyset \\ \max\limits_{i<k<j : a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{se } S_{ij} \neq \emptyset \end{cases}$$

Agora é fácil escrever o algoritmo de programação dinâmica. (PD será vista adiante)

Seleção de Atividades

Podemos “converter” o algoritmo de programação dinâmica em um algoritmo guloso se notarmos que o primeiro resolve subproblemas desnecessariamente.

Teorema: (escolha gulosa)

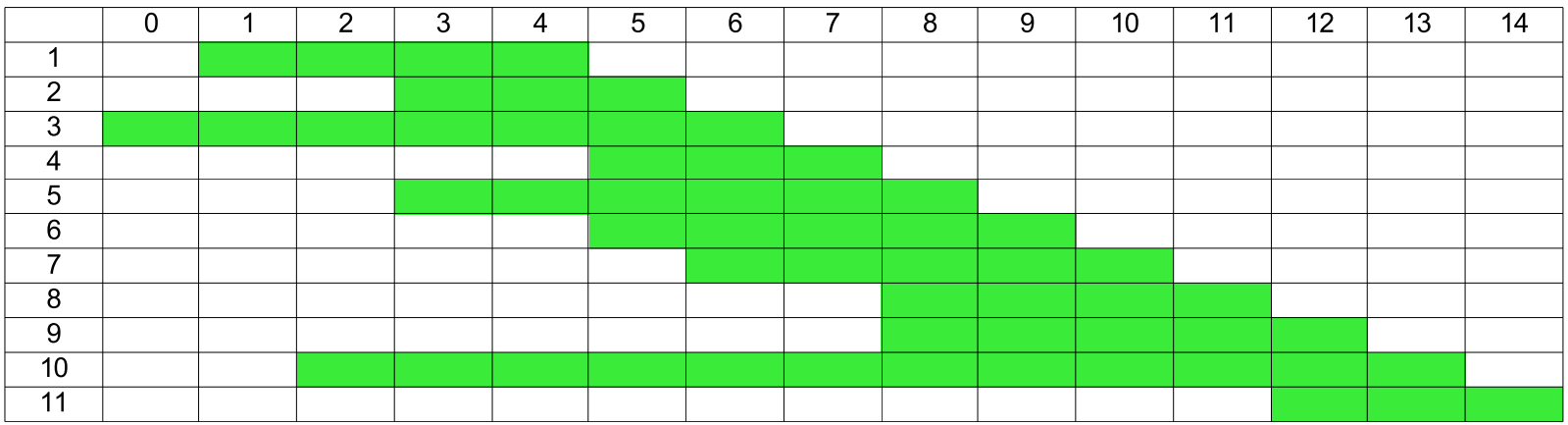
Considere o subproblema definido para uma instância não-vazia S_{ij}, e seja a_m a atividade de S_{ij} com o menor tempo de término, i.e.:

$$f_m = \min\{f_k : a_k \in S_{ij}\}.$$

Então **(a)** existe uma solução ótima para S_{ij} que contém a_m e **(b)** S_{im} é vazio e o subproblema definido para esta instância é trivial, portanto, a escolha de a_m deixa apenas um dos subproblemas com solução possivelmente não-trivial, já que S_{mj} pode não ser vazio.

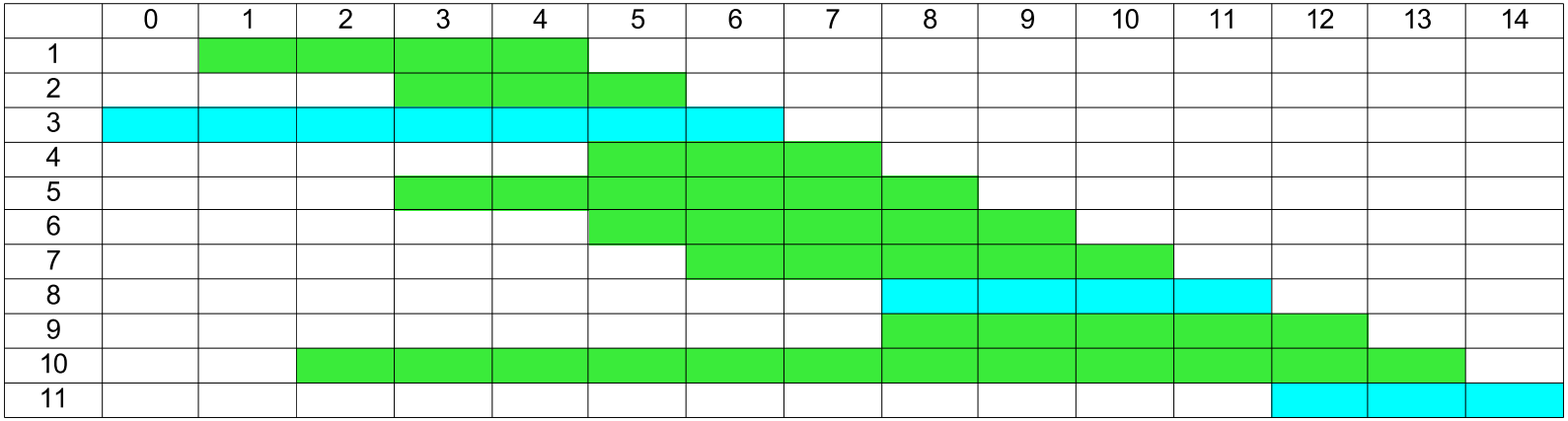
Seleção de Atividades

- Critérios para seleção de atividades
  - Exemplo: 11 atividades, em 14 unidades de tempo



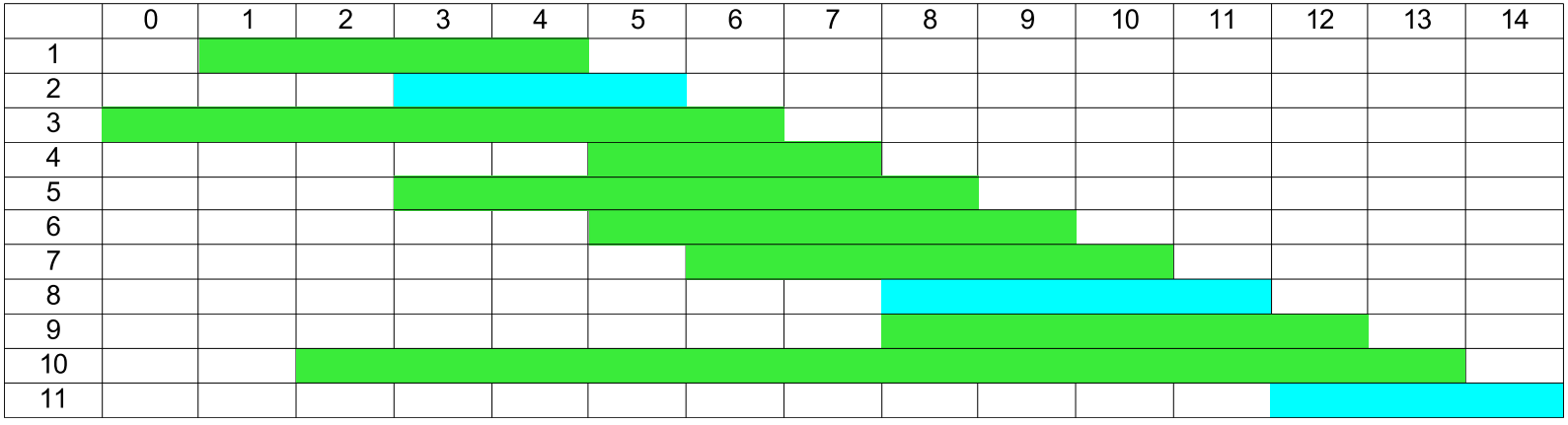
Seleção de Atividades

- Critérios para seleção de atividades
  - Exemplo: 11 atividades, em 14 unidades de tempo
  - **Tentativa 1** – Escolher primeiro as atividades que começam primeiro
    - Não foi o ideal, pois escolheu apenas três atividades: 3, 8 e 11
      (quando poderia ter escolhido quatro: 1, 4, 8 e 11)



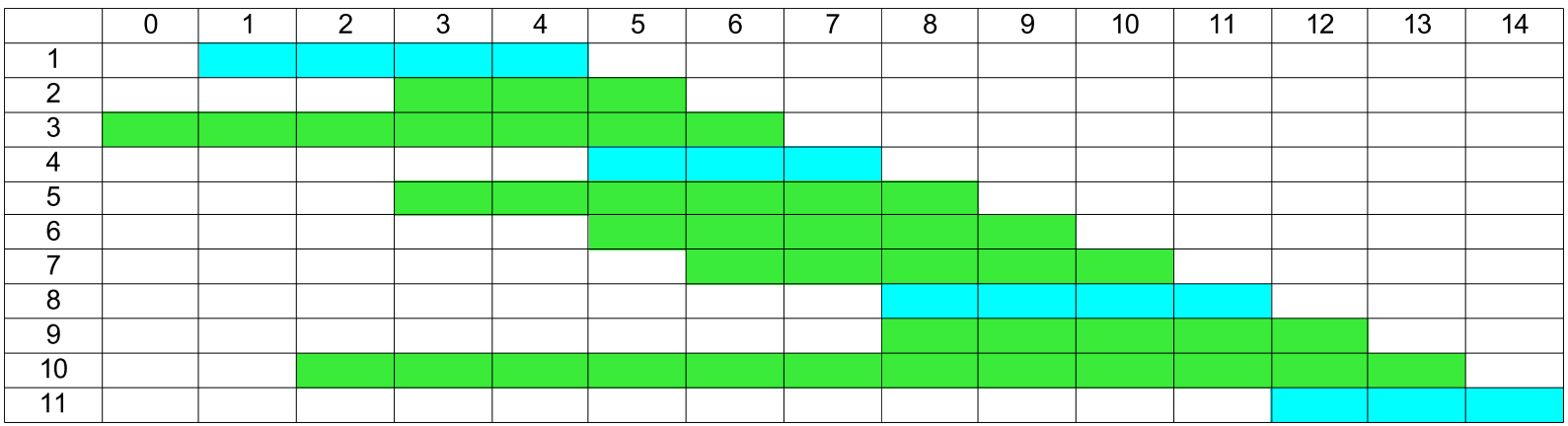
Seleção de Atividades

- Critérios para seleção de atividades
  - Exemplo: 11 atividades, em 14 unidades de tempo
  - **Tentativa 2** – Escolher primeiro as atividades que demoram menos tempo
    - Não foi o ideal, pois escolheu apenas três atividades: 2, 8 e 11
      (quando poderia ter escolhido quatro: 1, 4, 8 e 11)



Seleção de Atividades

- Critérios para seleção de atividades
  - Exemplo: 11 atividades, em 14 unidades de tempo
  - **Tentativa 3** – Escolher primeiro as atividades que terminam primeiro → melhor resultado!
    - Agora escolheu a solução ótima: 1, 4, 8 e 11



Seleção de Atividades

Algoritmo :

- Suponha que estamos tentando resolver S_{ij}.
- Determine a atividade a_m com menor tempo de término em S_{ij}.
- Resolva o subproblema S_{mj} e junte a_m à solução obtida na recursão. Devolva este conjunto de atividades.

Seleção de Atividades

SelecAtivGulRec(s, f, i, j)

▷ **Entrada:** vetores s e f com instantes de início e término das atividades $a_i, a_{i+1}, \ldots, a_j$, sendo $f_i \leq \ldots \leq f_j$.

▷ **Saída:** conjunto de tamanho máximo de índices de atividades mutuamente compatíveis.

1. $m \leftarrow i + 1$;
   ▷ Busca atividade com menor tempo de término que está em S_{ij}
2. **enquanto** $m < j$ **e** $s_m < f_i$ **faça** $m \leftarrow m + 1$;
3. **se** $m \geq j$ **então devolva** $\emptyset$;
4. **senão**
5. **se** $f_m > s_j$ **então devolva** $\emptyset$; ▷ $a_m \notin S_{ij}$
6. **senão devolva** $\{a_m\} \cup$ SelecAtivGulRec(s, f, m, j).

Seleção de Atividades

- A chamada inicial será $SelecAtivGulRec(s, f, 0, n + 1)$.
- **Complexidade**: $\Theta(n)$.
  Ao longo de todas as chamadas recursivas,cada atividade é examinada exatamente uma vez no laço da linha 2. Em particular, a atividade a_k é examinada na última chamada com $i < k$.
- Como o algoritmo anterior é um caso simples de recursão caudal, é trivial escrever uma versão iterativa do mesmo.

Seleção de Atividades

SelecAtivGulIter(s, f, n)

▷ **Entrada:** vetores s e f com instantes de início e término das n atividades com os instantes de término em ordem crescente.
▷ **Saída:** um conjunto A de tamanho máximo contendo atividades mutuamente compatíveis.

1. $A \leftarrow \{a_1\}$;
2. $i \leftarrow 1$;
3. **para** $m \leftarrow 2$ **até** n **faça**
4. **se** $s_m \geq f_i$ **então**
5. $A \leftarrow A \cup \{a_m\}$;
6. $i \leftarrow m$;
7. **devolva** A.

Seleção de Atividades

- Observe que na linha 3, i é o índice da última atividade colocada em A. Como as atividades estão em ordenadas pelo instante de término, tem-se que:

$$f_i = \max\{f_k : a_k \in A\},$$

  ou seja, f_i é sempre o maior instante de término de uma atividade em A.
- Pode-se concluir que o algoritmo faz as mesmas escolhas de *SelecAtivGulRec* e portanto, está correto.
- **Complexidade**: $\Theta(n)$.

Algoritmos gulosos – códigos de Huffman

Códigos de Huffman

- Códigos de Huffman: técnica de compressão de dados.
- Reduções no tamanho dos arquivos dependem das características dos dados contidos nos mesmos. Valores típicos oscilam entre 20 e 90%.
- Exemplo: arquivo texto contendo 100.000 caracteres no alfabeto $\Sigma = \{a, b, c, d, e, f\}$. As *freqüências* de cada caracter no arquivo são indicadas na tabela abaixo.

	a	*b*	*c*	*d*	*e*	*f*
Freqüência (em milhares)	45	13	12	16	9	5
Código de tamanho fixo	000	001	010	011	100	101
Código de tamanho variável	0	101	100	111	1101	1100

- Codificação do arquivo: representar cada caracter por uma seqüência de *bits*
- Alternativas:
  1. seqüências de tamanho fixo.
  2. seqüências de tamanho variável.

Códigos de Huffman

- Qual o tamanho (em *bits*) do arquivo comprimido usando os códigos acima ?
- Códigos de tamanho fixo: $3 \times 100.000 = 300.000$
  Códigos de tamanho variável:

$$(\underbrace{45 \times 1}_{a} + \underbrace{13 \times 3}_{b} + \underbrace{12 \times 3}_{c} + \underbrace{16 \times 3}_{d} + \underbrace{9 \times 4}_{e} + \underbrace{5 \times 4}_{f}) \times 1.000 = 224.000$$

Ganho de $\approx 25\%$ em relação à solução anterior.

Problema da Codificação:

Dadas as freqüências de ocorrência dos caracteres de um arquivo, encontrar as seqüências de *bits* (códigos) para representá-los de modo que o arquivo comprimido tenha tamanho mínimo.

Códigos de Huffman

Definição:

Códigos livres de prefixo são aqueles onde, dados dois caracteres quaisquer i e j representados pela codificação, a seqüência de *bits* associada a i **não** é um *prefixo* da seqüência associada a j.

Importante:

Pode-se provar que sempre **existe** uma solução ótima do problema da codificação que é dado por um código *livre de prefixo*.

Códigos de Huffman – codificação

O processo de codificação, i.e, de geração do arquivo comprimido é sempre fácil pois reduz-se a concatenar os códigos dos caracteres presentes no arquivo original em seqüência.

Exemplo: usando a codificação de tamanho variável do exemplo anterior, o arquivo original dado por `abc` seria codificado por `0101100`.

Códigos de Huffman – decodificação

- A vantagem dos códigos livres de prefixo se torna evidente quando vamos decodificar o arquivo comprimido.
- Como nenhum código é prefixo de outro código, o código que se encontra no início do arquivo comprimido não apresenta ambigüidade. Pode-se simplesmente identificar este código inicial, traduzi-lo de volta ao caracter original e repetir o processo no restante do arquivo comprimido.
- Exemplo: usando a codificação de tamanho variável do exemplo anterior, o arquivo comprimido contendo os *bits* `001011101` divide-se de **forma unívoca** em `0 0 101 1101`, ou seja, corresponde ao arquivo original dado por *aabe*.

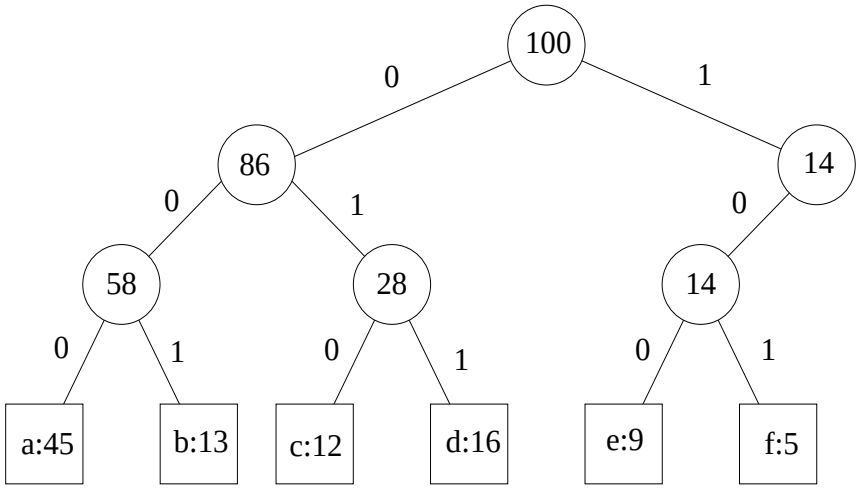
Códigos de Huffman

- Como representar de maneira conveniente uma codificação livre de prefixo de modo a facilitar o processo de decodificação?
- Solução: usar uma árvore binária.
  O filho esquerdo está associado ao *bit* ZERO enquanto o filho direito está associado ao *bit* UM. Nas folhas encontram-se os caracteres presentes no arquivo original.

Códigos de Huffman

Vejamos como ficam as árvores que representam os códigos do exemplo anterior.

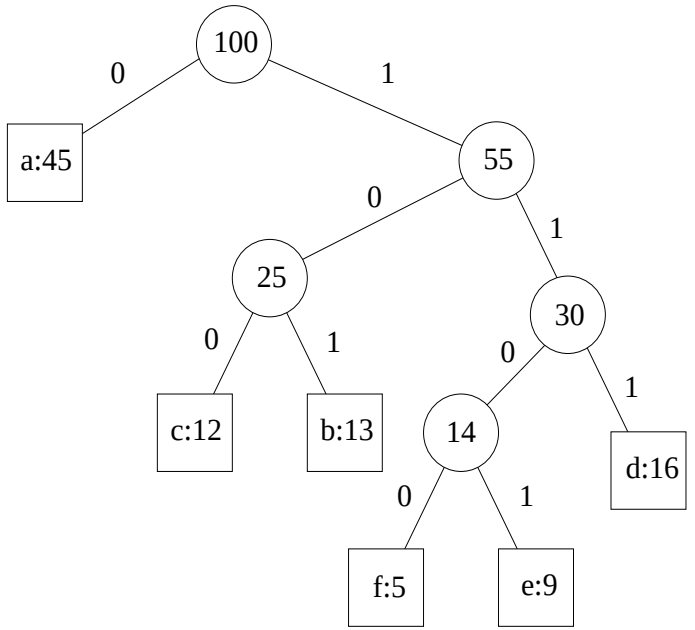
	a	*b*	*c*	*d*	*e*	*f*
Freqüência	45	13	12	16	9	5
Código fixo	000	001	010	011	100	101



Códigos de Huffman

Vejamos como ficam as árvores que representam os códigos do exemplo anterior.

	a	*b*	*c*	*d*	*e*	*f*
Freqüência	45	13	12	16	9	5
Código variável	0	101	100	111	1101	1100



Códigos de Huffman

- Pode-se mostrar (Exercício!) que uma codificação ótima sempre pode ser representada por uma árvore binária cheia, na qual cada vértice interno tem exatamente dois filhos.
- Então podemos restringir nossa atenção às árvores binárias cheias com $|C|$ folhas e $|C| - 1$ vértices internos (Exercício!), onde C é o conjunto de caracteres do alfabeto no qual está escrito o arquivo original.

Códigos de Huffman

Computando o tamanho do arquivo comprimido:

Se T é a árvore que representa a codificação, $d_T(c)$ é a profundidade da folha representado o caracter c e $f(c)$ é a sua freqüência, o tamanho do arquivo comprimido será dado por:

$$B(T) = \sum_{c \in C} f(c) d_T(c).$$

Dizemos que $B(T)$ é o **custo** da árvore T.
Isto é exatamente o tamanho do arquivo codificado.

Códigos de Huffman

- **Idéia do algoritmo de Huffman**: Começar com $|C|$ folhas e realizar sequencialmente $|C| - 1$ operações de "intercalação" de dois vértices da árvore.
  Cada uma destas intercalações dá origem a um novo vértice interno, que será o pai dos vértices que participaram da intercalação.
- A escolha do par de vértices que dará origem a intercalação em cada passo depende da soma das freqüências das folhas das subárvores com raízes nos vértices que ainda não participaram de intercalações.

Algoritmo de Huffman

Huffman(C)

▷ **Entrada:** Conjunto de caracteres C e as freqüências f dos caracteres em C.

▷ **Saída:** raiz de uma árvore binária representando uma codificação ótima livre de prefixos.

1. $n \leftarrow |C|$;
   ▷ Q é fila de prioridades dada pelas freqüências dos vértices ainda não intercalados
2. $Q \leftarrow C$;
3. **para** $i \leftarrow 1$ **até** $n - 1$ **faça**
4. **alocar novo registro** z; ▷ vértice de T
5. $z.esq \leftarrow x \leftarrow$ EXTRAI_MIN(Q);
6. $z.dir \leftarrow y \leftarrow$ EXTRAI_MIN(Q);
7. $z.f \leftarrow x.f + y.f$;
8. INSERE(Q, z);
9. **retorne** EXTRAI_MIN(Q).

Corretude do algoritmo de Huffman

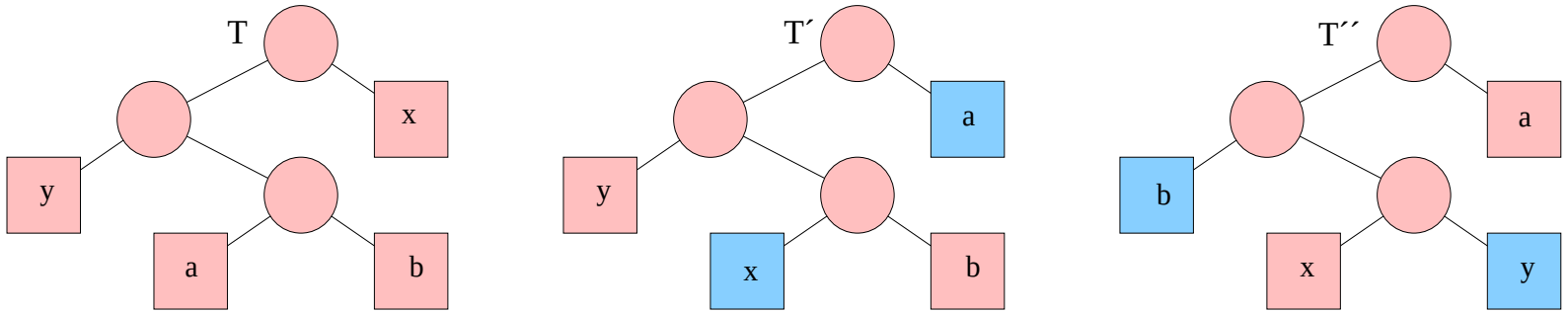
Lema 1: (escolha gulosa)

Seja C um alfabeto onde cada caracter $c \in C$ tem freqüência $f[c]$. Sejam x e y dois caracteres em C com as **menores** freqüências. Então, existe **um** código ótimo livre de prefixo para C no qual os códigos para x e y tem o mesmo comprimento e diferem apenas no último bit.

Prova do Lema 1:

- Seja T uma árvore **ótima**.
- Sejam a e b duas folhas "irmãs" (i.e. usadas em uma intercalação) mais profundas de T e x e y as folhas de T de menor freqüência.
- **Idéia:** a partir de T, obter uma outra árvore **ótima** T' com x e y sendo duas folhas "irmãs".

Corretude do algoritmo de Huffman



$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x] d_T(x) + f[a] d_T(a) - f[x] d_{T'}(x) - f[a] d_{T'}(a) \\
&= f[x] d_T(x) + f[a] d_T(a) - f[x] d_T(a) - f[a] d_T(x) \\
&= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0
\end{aligned}
$$

Assim, $B(T) \geq B(T')$.

Analogamente $B(T') \geq B(T'')$.

Como T é ótima, T'' é ótima e o resultado vale. $\square$

Corretude do algoritmo de Huffman

Lema 2: (subestrutura ótima)

Seja C um alfabeto com freqüência $f[c]$ definida para cada caracter $c \in C$. Sejam x e y dois caracteres de C com as menores freqüências. Seja C' o alfabeto obtido pela remoção de x e y e pela inclusão de um **novo** caracter z, ou seja, $C' = C \cup \{z\} - \{x, y\}$. As freqüências dos caracteres em $C' \cap C$ são as mesmas que em C e $f[z]$ é definida como sendo $f[z] = f[x] + f[y]$.
Seja T' uma árvore binária representado um código ótimo livre de prefixo para C'. Então a árvore binária T obtida de T' substituindo-se o vértice (folha) z pela por um vértice interno tendo x e y como fihos, representa uma código ótimo livre de prefixo para C.

Corretude do algoritmo de Huffman

Teorema:

O algoritmo de Huffman constrói um código ótimo (livre de prefixo).

Segue imediatamente dos Lemas 1 e 2.

Algoritmos gulosos – caminho mínimo

Problema do(s) Caminho(s) Mínimo(s)

Seja G um grafo orientado e suponha que para cada aresta (u, v) associamos um peso (custo, distância) $w(u, v)$.
Usaremos a notação (G, w).

- Problema do Caminho Mínimo entre Dois Vértices:
  Dados dois vértices s e t em (G, w), encontrar um caminho (de peso) mínimo de s a t.
- Aparentemente, este problema não é mais fácil do que o

  Problema dos Caminhos Mínimos com Mesma Origem:
  Dados (G, w) e $s \in V[G]$, encontrar para cada vértice v de G, um caminho mínimo de s a v.

Subsestrutura ótima de caminhos mínimos

Teorema. Seja (G, w) um grafo orientado e seja

$$P = (v_1, v_2, \ldots, v_k)$$

um caminho mínimo de v_1 a v_k.

Então para quaisquer i, j com $1 \leq i \leq j \leq k$

$$P_{ij} = (v_i, v_{i+1}, \ldots, v_j)$$

é um caminho mínimo de v_i a v_j.

Representação de caminhos mínimos

- Usamos uma idéia similar à usada em Busca em Largura nos algoritmos de caminhos mínimos que veremos.
- Para cada vértice $v \in V[G]$ associamos um predecessor $\pi[v]$.
- Ao final do algoritmo obtemos uma Árvore de Caminhos Mínimos com raiz s.
- Um caminho de s a v nesta árvore é um caminho mínimo de s a v em (G, w).

Estimativa de distâncias

- Para cada $v \in V[G]$ queremos determinar $dist(s, v)$, o peso de um caminho mínimo de s a v em (G, w) (distância de s a v.)
- Os algoritmos de caminhos mínimos associam a cada $v \in V[G]$ um valor $d[v]$ que é uma estimativa da distância $dist(s, v)$.

Inicialização

INITIALIZE-SINGLE-SOURCE(G, s)

1 **para cada** vértice $v \in V[G]$ **faça**
2 $\quad d[v] \leftarrow \infty$
3 $\quad \pi[v] \leftarrow$ NIL
4 $d[s] \leftarrow 0$

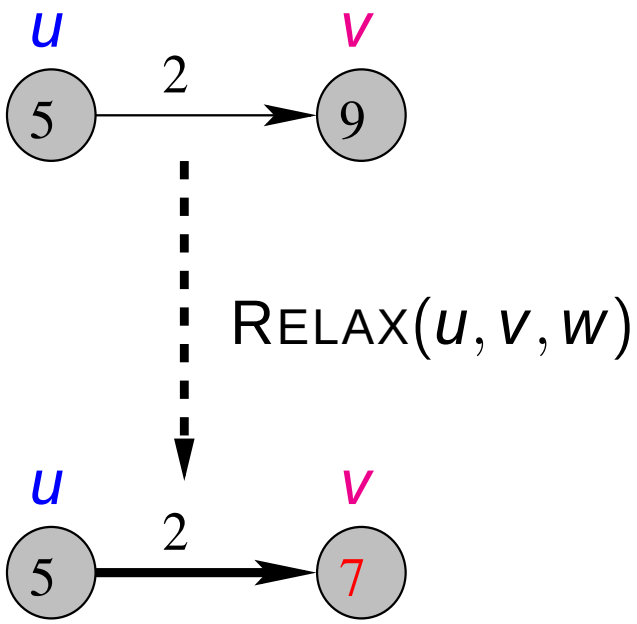
O valor $d[v]$ é uma estimativa superior para o peso de um caminho mínimo de s a v.

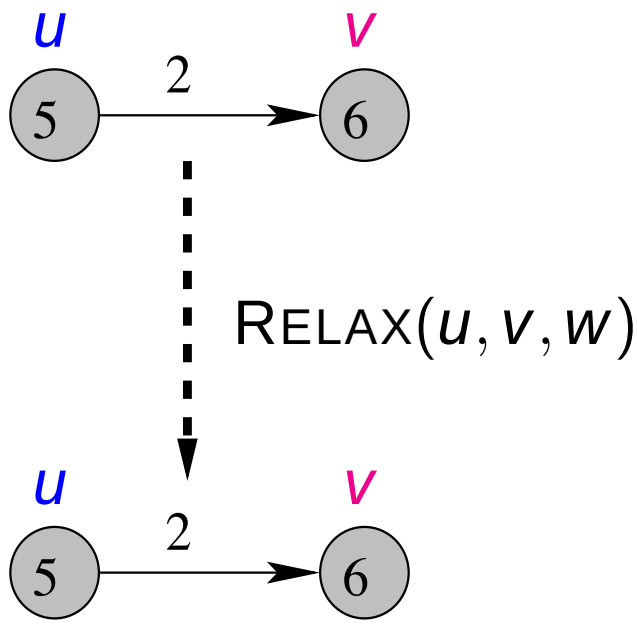
Ele indica que o algoritmo encontrou até aquele momento um caminho de s a v com peso $d[v]$.

O caminho pode ser recuperado por meio dos predecessores $\pi[\,]$.

Relaxação

Tenta melhorar a estimativa $d[v]$ examinando (u, v).



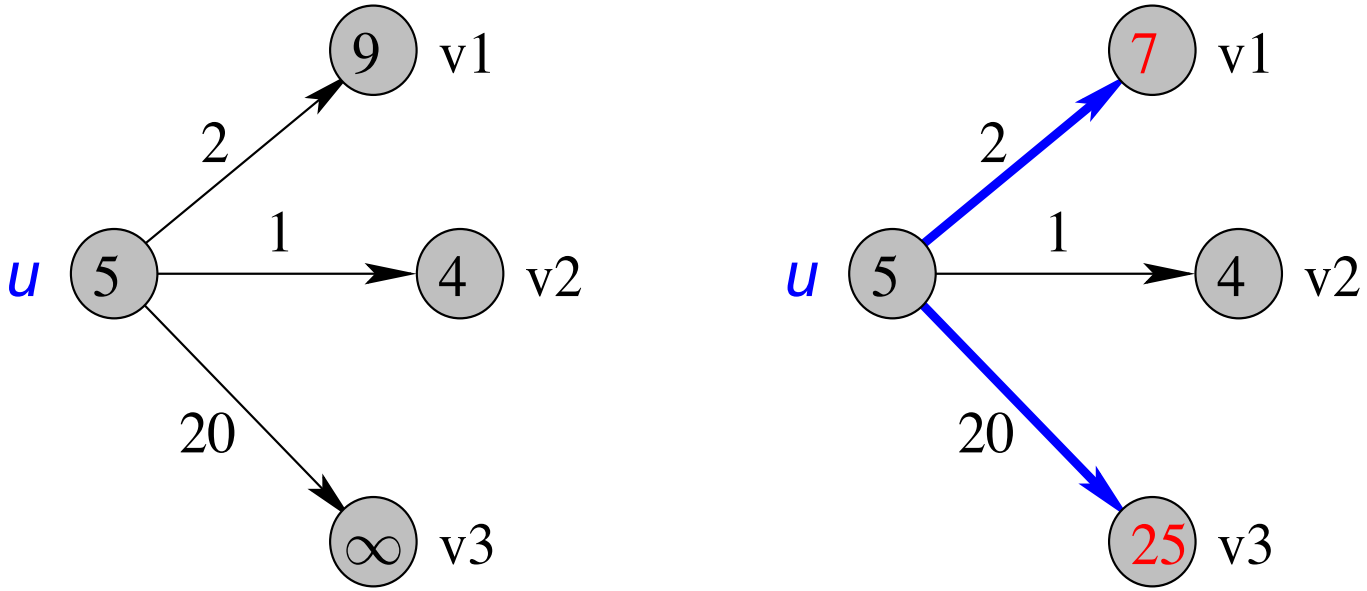


RELAX(u, v, w)

1 **se** $d[v] > d[u] + w(u, v)$
2 **então** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Relaxação dos vizinhos

Em cada iteração o algoritmo seleciona um vértice u e para cada vizinho v de u aplica RELAX(u, v, w).



RELAX(u, v, w)

1 **se** $d[v] > d[u] + w(u, v)$ **faça**
2 $\quad d[v] \leftarrow d[u] + w(u, v)$
3 $\quad \pi[v] \leftarrow u$

Caminhos Mínimos

Veremos três algoritmos baseados em relaxação para tipos de instâncias diferentes de Problemas de Caminhos Mínimos.

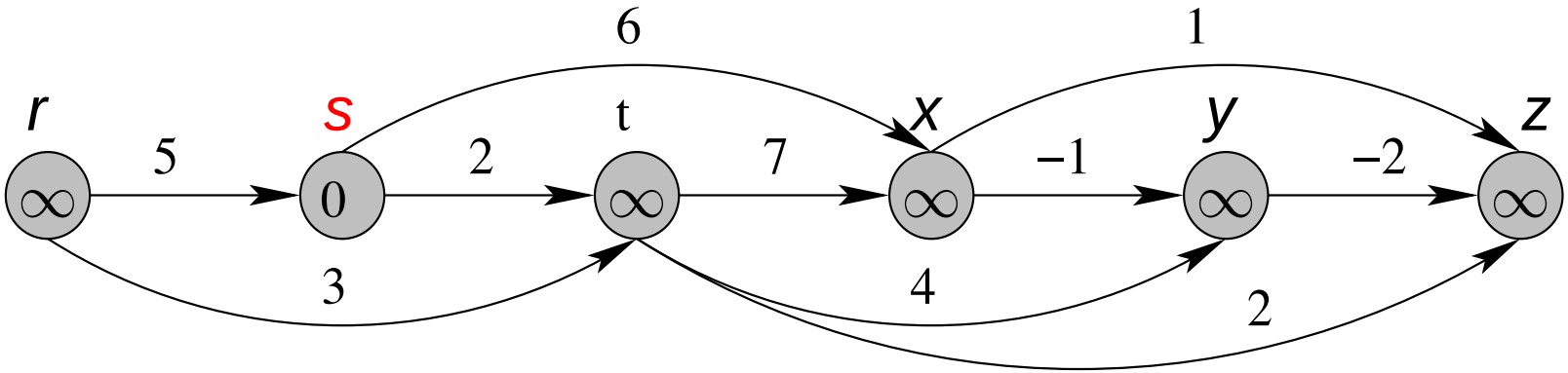
- G é acíclico: aplicação de ordenação topológica
- (G, w) não tem arestas de peso negativo: algoritmo de Dijkstra
- (G, w) tem arestas de peso negativo, mas não contém ciclos negativos: algoritmo de Bellman-Ford.

Caminhos mínimos em grafos acíclicos

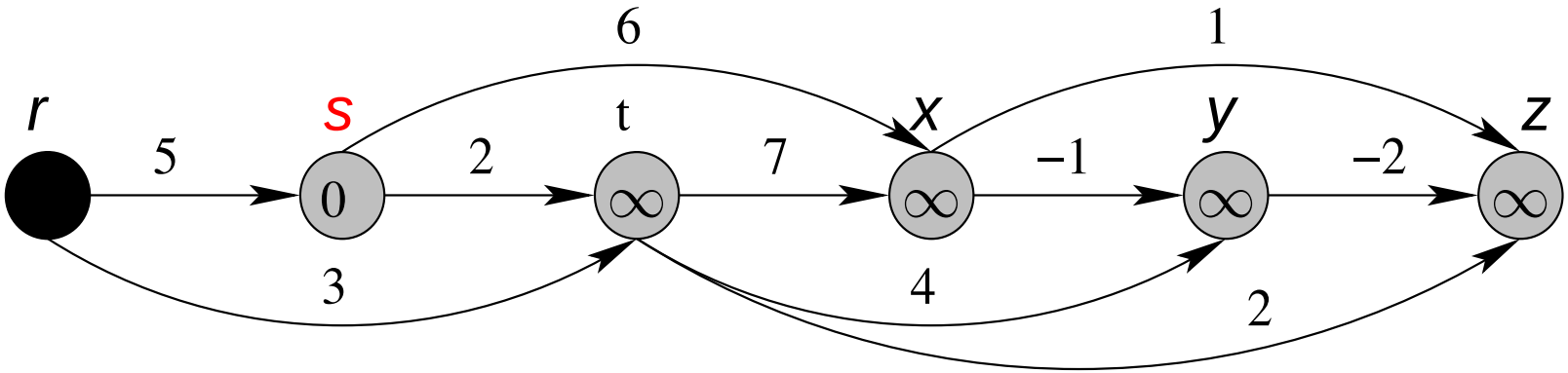
Entrada: grafo orientado acíclico $G = (V, E)$ com função peso w nas arestas e uma origem s.
Saída: vetor $d[v] = dist(s, v)$ para $v \in V$
e uma Árvore de Caminhos Mínimos definida por $\pi[\,]$.

DAG-SHORTEST-PATHS(G, w, s)
1 Ordene topologicamente os vértices de G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **para cada** vértice u na ordem topológica **faça**
4 **para cada** $v \in \mathrm{Adj}[u]$ **faça**
5 RELAX(u, v, w)

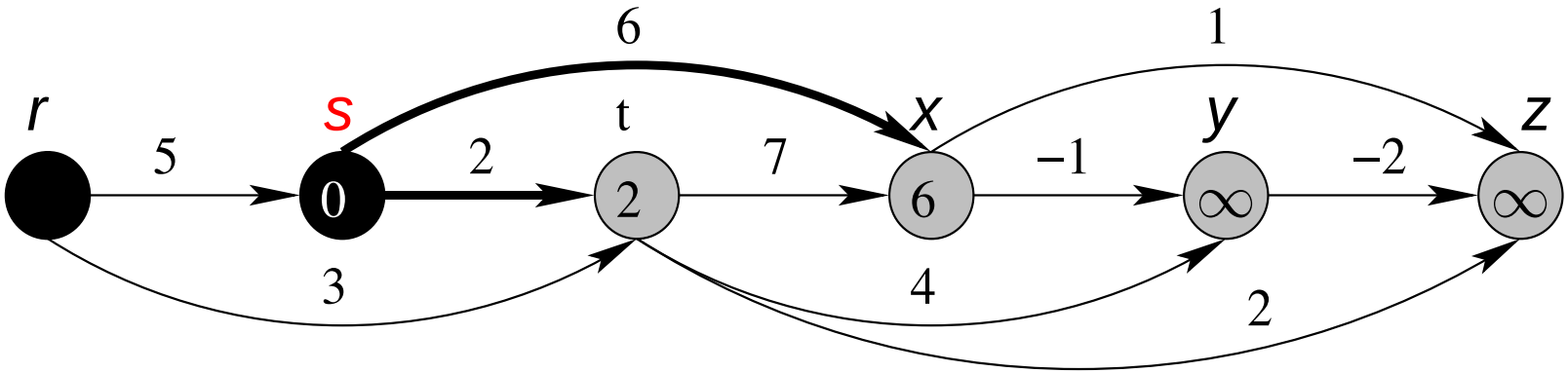
Exemplo



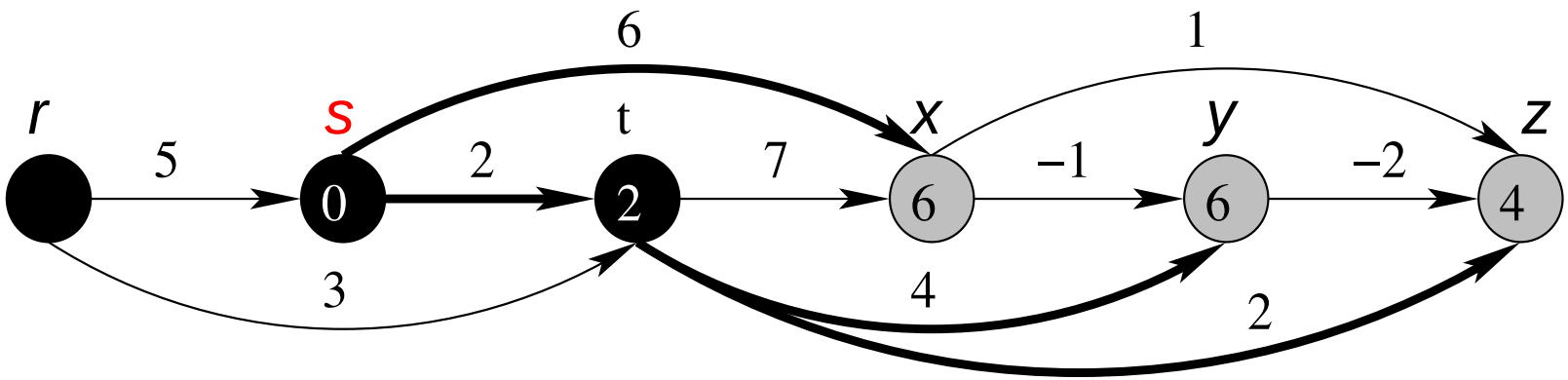
Exemplo



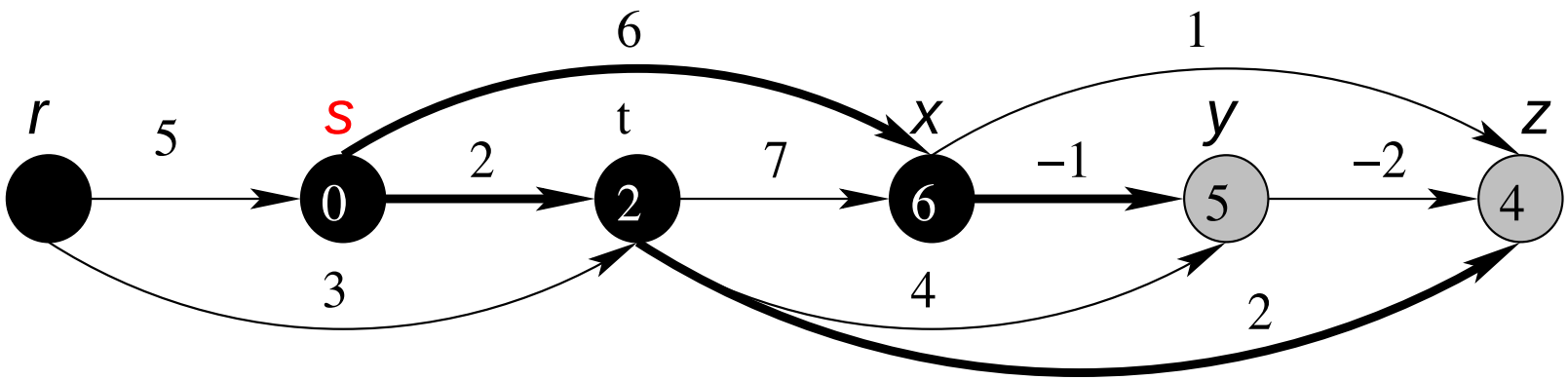
Exemplo



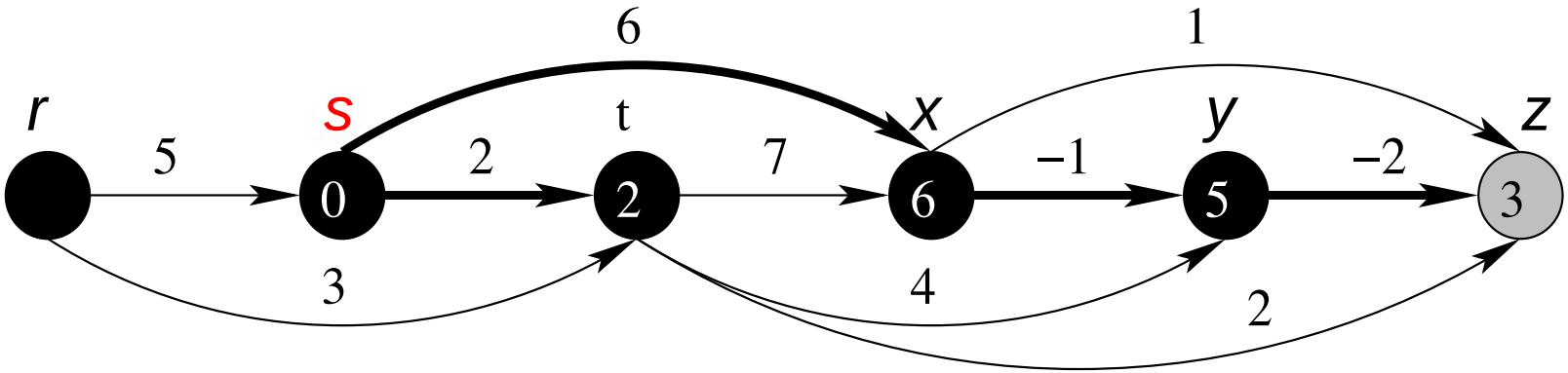
Exemplo



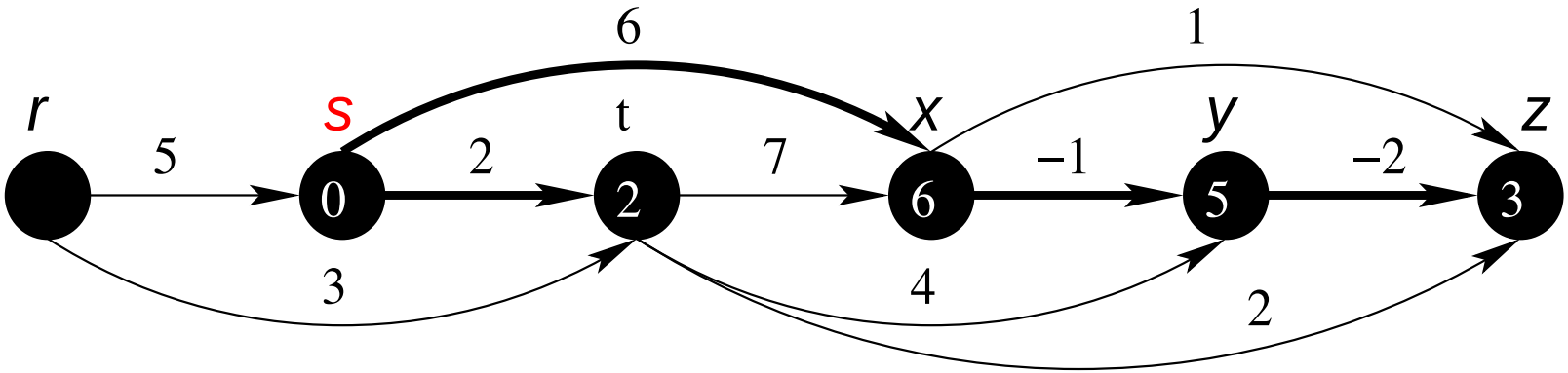
Exemplo



Exemplo



Exemplo



Complexidade

DAG-SHORTEST-PATHS(G, w, s)
1 Ordene topologicamente os vértices de G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **para cada** vértice u na ordem topológica **faça**
4 **para cada** $v \in \mathrm{Adj}[u]$ **faça**
5 RELAX(u, v, w)

Linha(s)	Tempo total
1	$O(V + E)$
2	$O(V)$
3-5	$O(V + E)$

Complexidade de DAG-SHORTEST-PATHS: $O(V + E)$

Algoritmo de Dijkstra

O algoritmo de Dijkstra recebe um grafo orientado (G, w) (sem arestas de peso negativo) e um vértice s de G

e devolve

- para cada $v \in V[G]$, o peso de um caminho mínimo de s a v
- e uma Árvore de Caminhos Mínimos com raiz s.
  Um caminho de s a v nesta árvore é um caminho mínimo de s a v em (G, w).

Algoritmo de Dijkstra

```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S ← ∅
3   Q ← V[G]
4   enquanto Q ≠ ∅ faça
5       u ← EXTRACT-MIN(Q)
6       S ← S ∪ {u}
7       para cada vértice v ∈ Adj[u] faça
8           RELAX(u, v, w)
```

O conjunto Q é implementado como uma fila de prioridade.

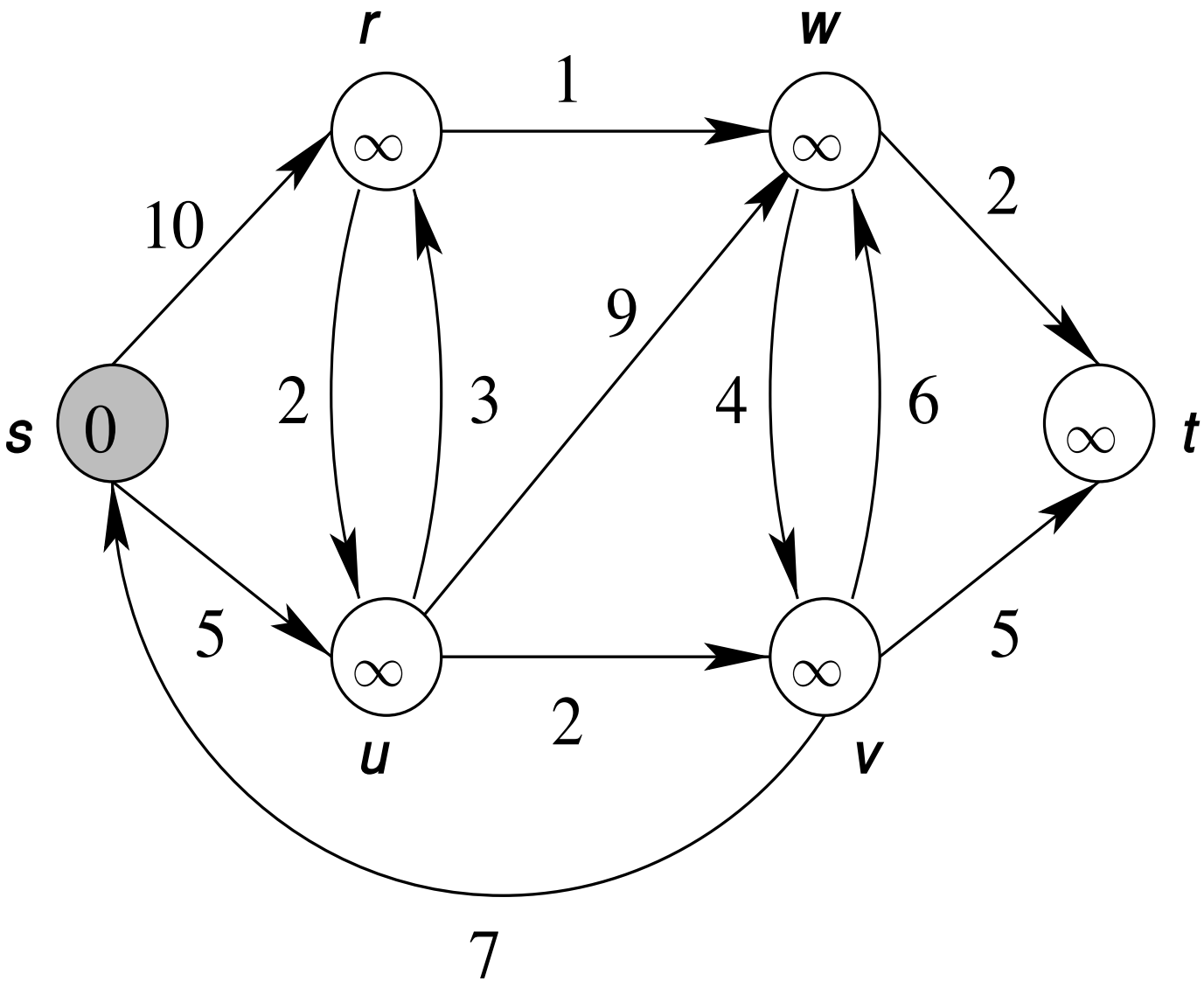
O conjunto S não é realmente necessário, mas simplifica a análise do algoritmo.

Intuição do algoritmo

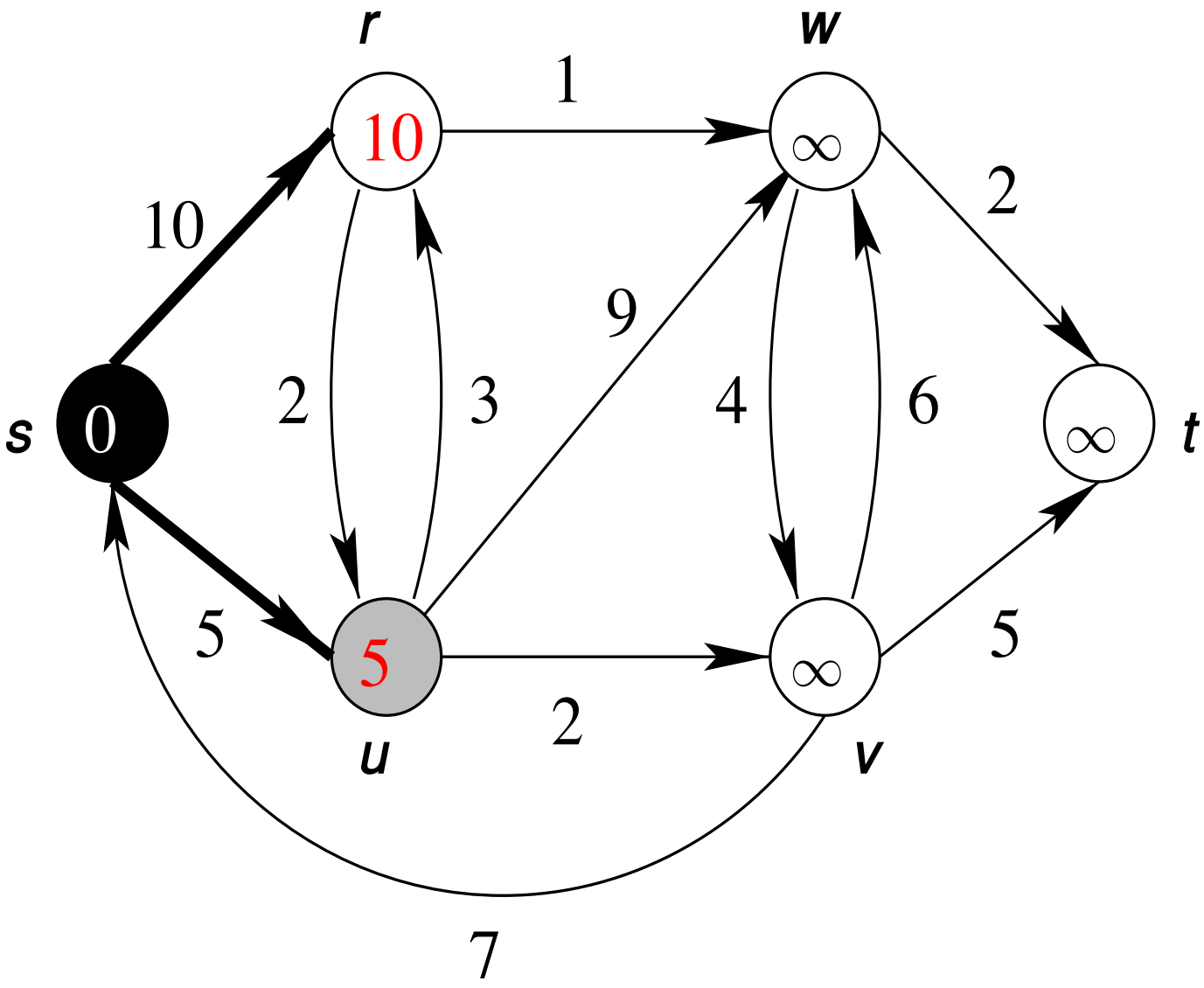
Em cada iteração, o algoritmo de DIJKSTRA

- escolhe um vértice u fora do conjunto S que esteja mais próximo a esse e acrescenta-o a S,
- atualiza as distâncias estimadas dos vizinhos de u e
- atualiza a Árvore dos Caminhos Mínimos.

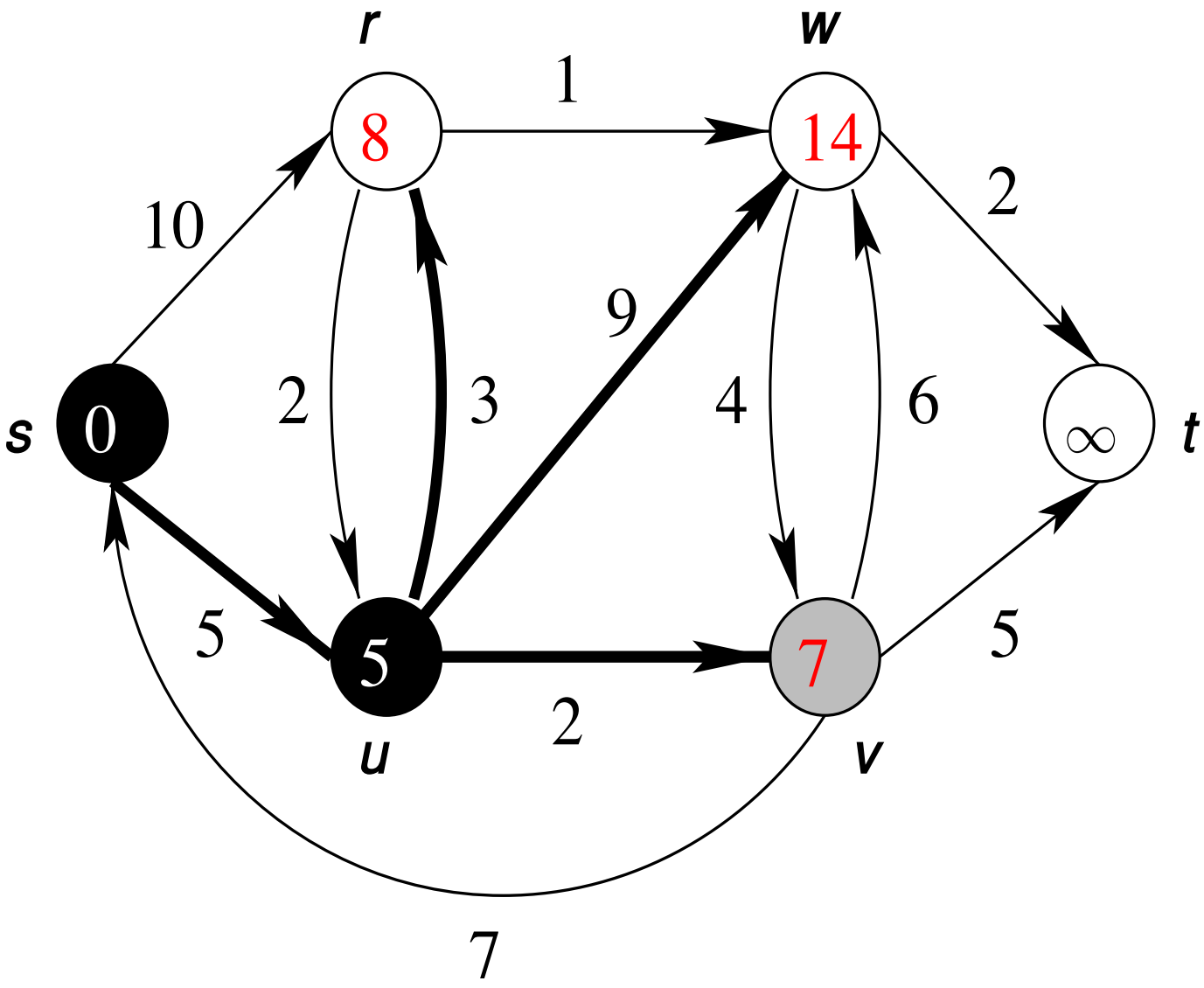
Exemplo (CLRS modificado)



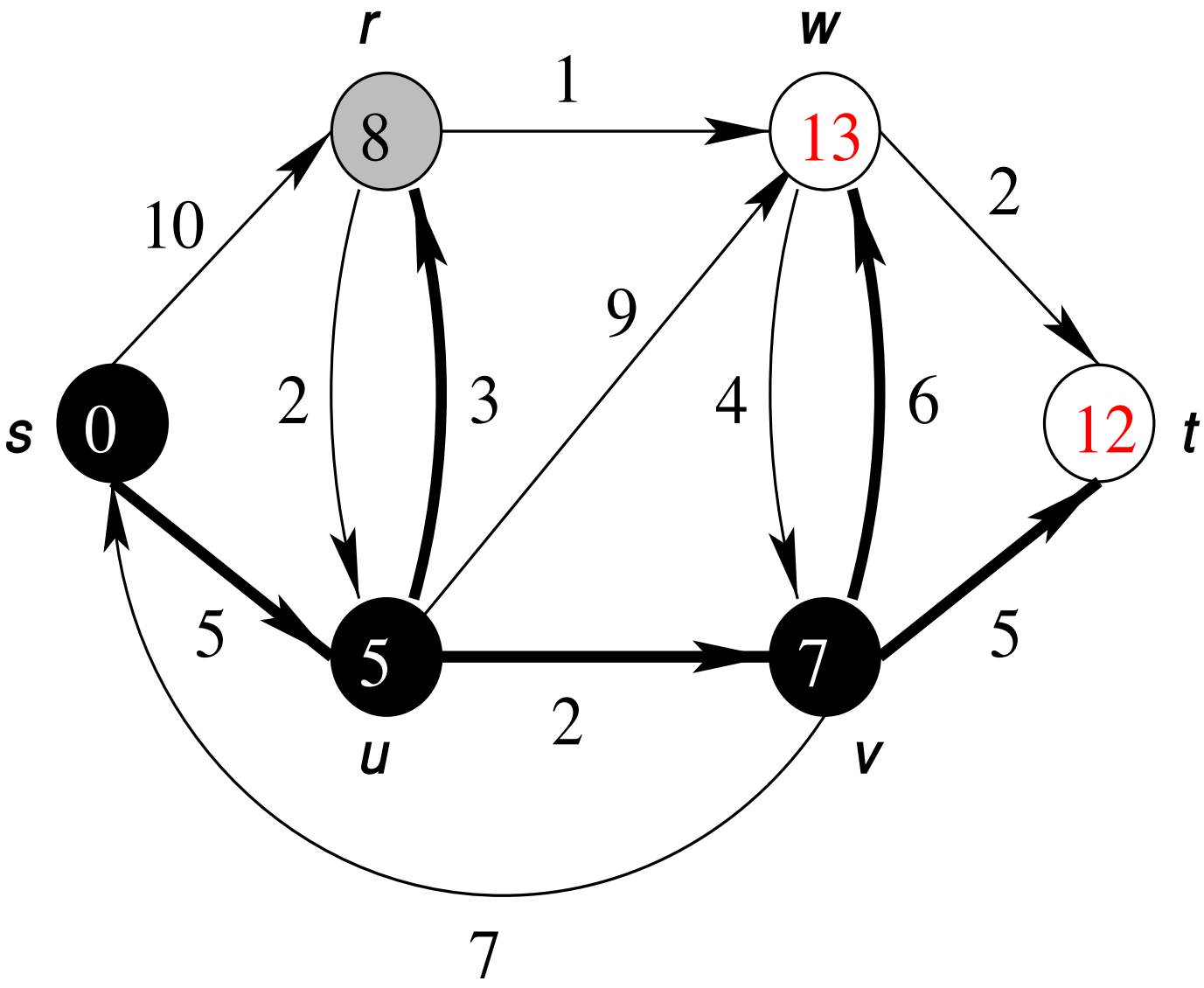
Exemplo (CLRS modificado)



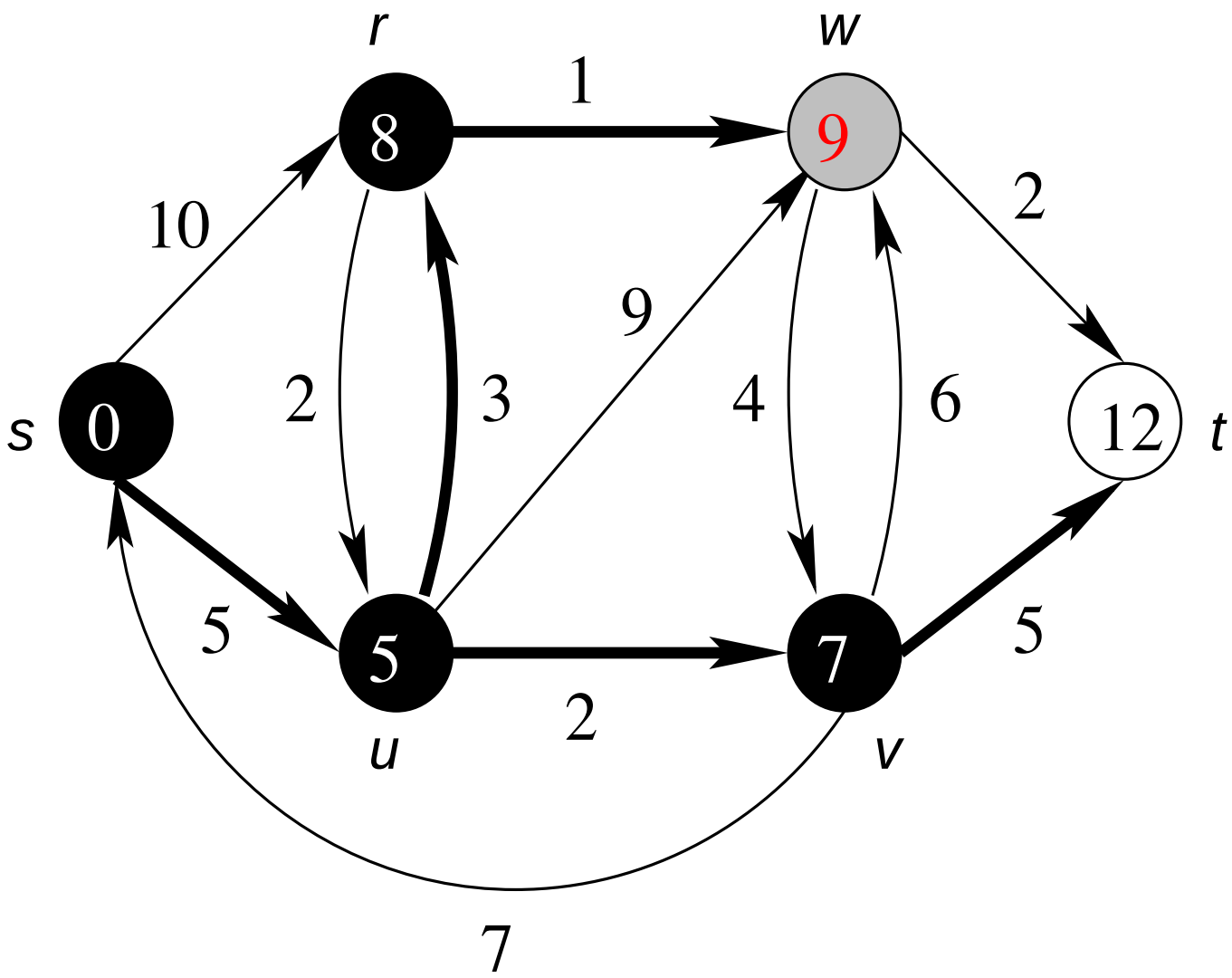
Exemplo (CLRS modificado)



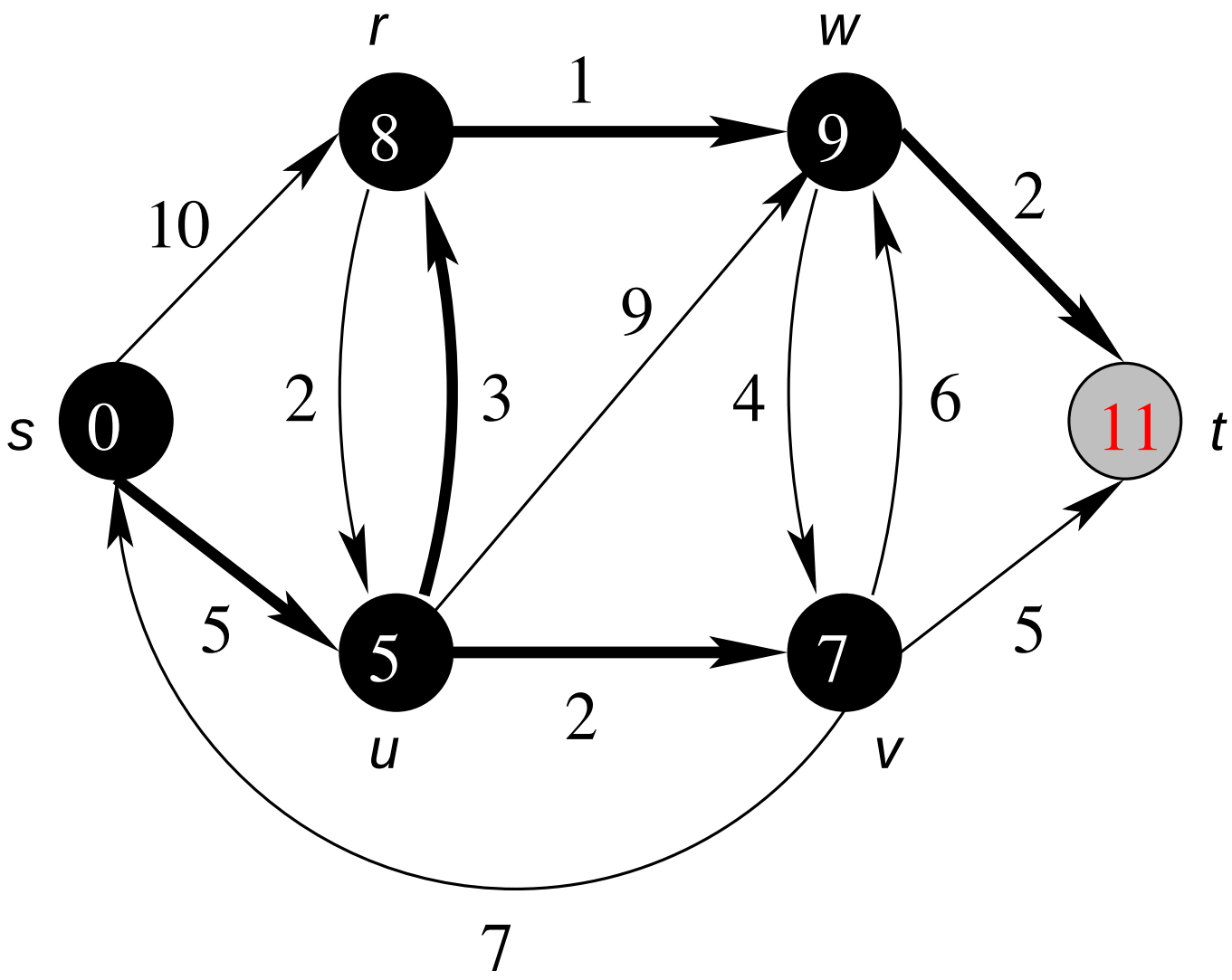
Exemplo (CLRS modificado)



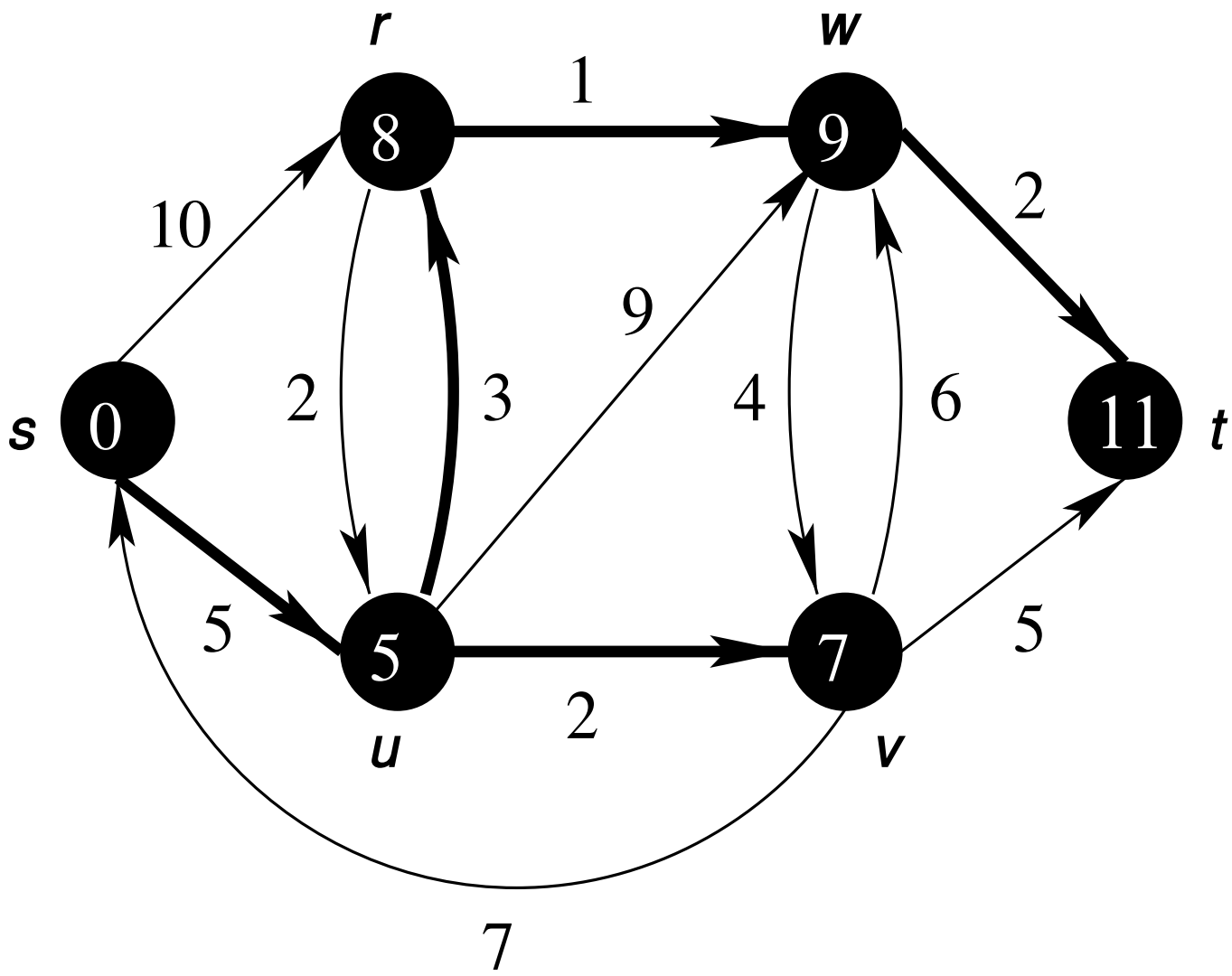
Exemplo (CLRS modificado)



Exemplo (CLRS modificado)



Exemplo (CLRS modificado)



Complexidade de tempo

```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S ← ∅
3   Q ← V[G]
4   enquanto Q ≠ ∅ faça
5       u ← EXTRACT-MIN(Q)
6       S ← S ∪ {u}
7       para cada vértice v ∈ Adj[u] faça
8           RELAX(u, v, w)
```

Depende de como a fila de prioridade Q é implementada.

Complexidade de tempo

A fila de prioridade Q é mantida com as seguintes operações:

- INSERT (implícito na linha 3)
- EXTRACT-MIN (linha 5) e
- DECREASE-KEY (implícito em RELAX na linha 8).

São executados (no máximo) $|V|$ operações EXTRACT-MIN.

Cada vértice $u \in V[G]$ é inserido em S (no máximo) uma vez e cada aresta (u, v) com $v \in \mathrm{Adj}[u]$ é examinada (no máximo) uma vez nas linhas 7-8 durante todo o algoritmo. Assim, são executados no máximo $|E|$ operações DECREASE-KEY.

Complexidade de tempo

No total temos $|V|$ chamadas a EXTRACT-MIN e $|E|$ chamadas a DECREASE-KEY.

- Implementando Q como um vetor (coloque $d[v]$ na posição v do vetor), INSERT e DECREASE-KEY gastam tempo $\Theta(1)$ e EXTRACT-MIN gasta tempo $O(V)$, resultando em um total de $O(V^2 + E) = O(V^2)$.
- Implementando a fila de prioridade Q como um min-heap, INSERT,EXTRACT-MIN e DECREASE-KEY gastam tempo $O(\lg V)$, resultando em um total de $O(V + E) \lg V$.
- Usando heaps de Fibonacci (EXTRACT-MIN é $O(\lg V)$ e DECREASE-KEY é $O(1)$) a complexidade reduz para $O(V \lg V + E)$.

Arestas/ciclos de peso negativo

- O algoritmo de Dijkstra resolve o Problema dos Caminhos Mínimos quando (G, w) não possui arestas de peso negativo.
- Quando (G, w) possui arestas negativas, o algoritmo de Dijkstra não funciona (Exercício).
- Uma das dificuldades com arestas negativas é a possível existência de ciclos de peso negativo ou simplesmente ciclos negativos.

Ciclos negativos — uma dificuldade

- Se um ciclo negativo C é atingível a partir da fonte s, em princípio o problema não tem solução pois o “caminho” pode passar ao longo do ciclo infinitas vezes obtendo caminhos cada vez menores.
- Naturalmente, podemos impor a restrição de que os caminhos tem que ser simples, sem repetição de vértices. Entretanto, esta versão do problema é NP-difícil.
- Assim, vamos nos restringir ao Problema de Caminhos Mínimos sem ciclos negativos.

O algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford recebe um grafo orientado (G, w) (possivelmente com arestas de peso negativo) e um vértice origem s de G

Ele devolve um valor booleano

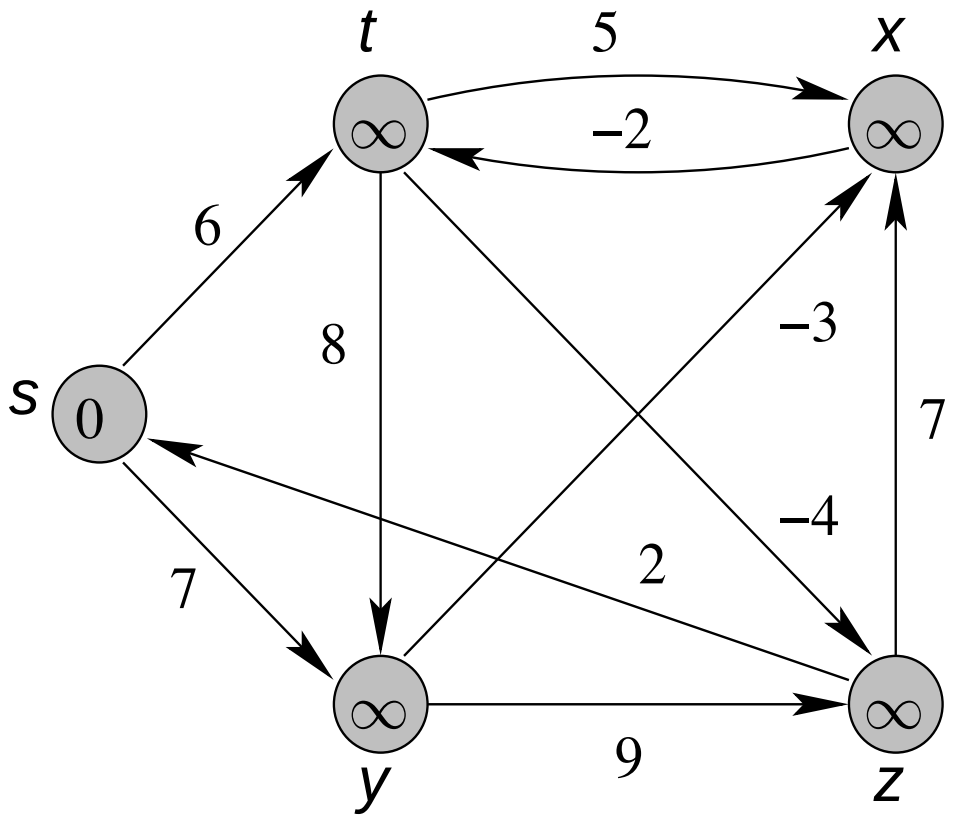
- FALSE se existe um ciclo negativo atingível a partir de s, ou
- TRUE e neste caso devolve também uma Árvore de Caminhos Mínimos com raiz s.

O algoritmo de Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   para i ← 1 até |V[G]| − 1 faça
3       para cada aresta (u, v) ∈ E[G] faça
4           RELAX(u, v, w)
5   para cada aresta (u, v) ∈ E[G] faça
6       se d[v] > d[u] + w(u, v)
7           então devolva FALSE
8   devolva TRUE
```

Complexidade de tempo: $O(VE)$

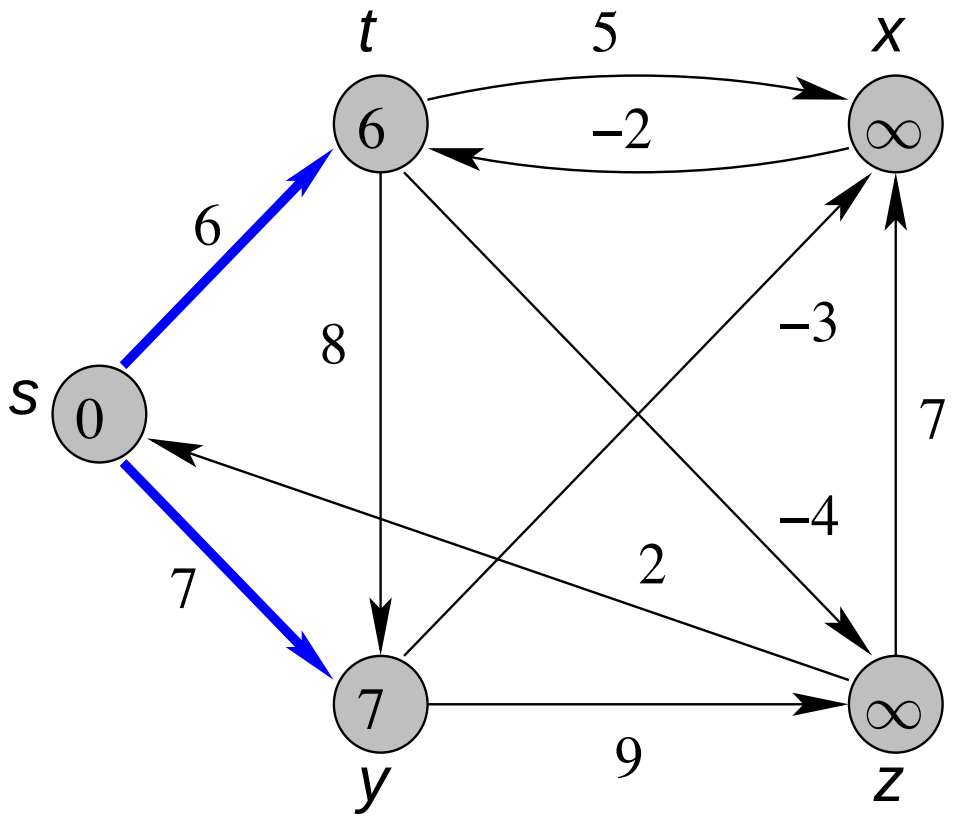
Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

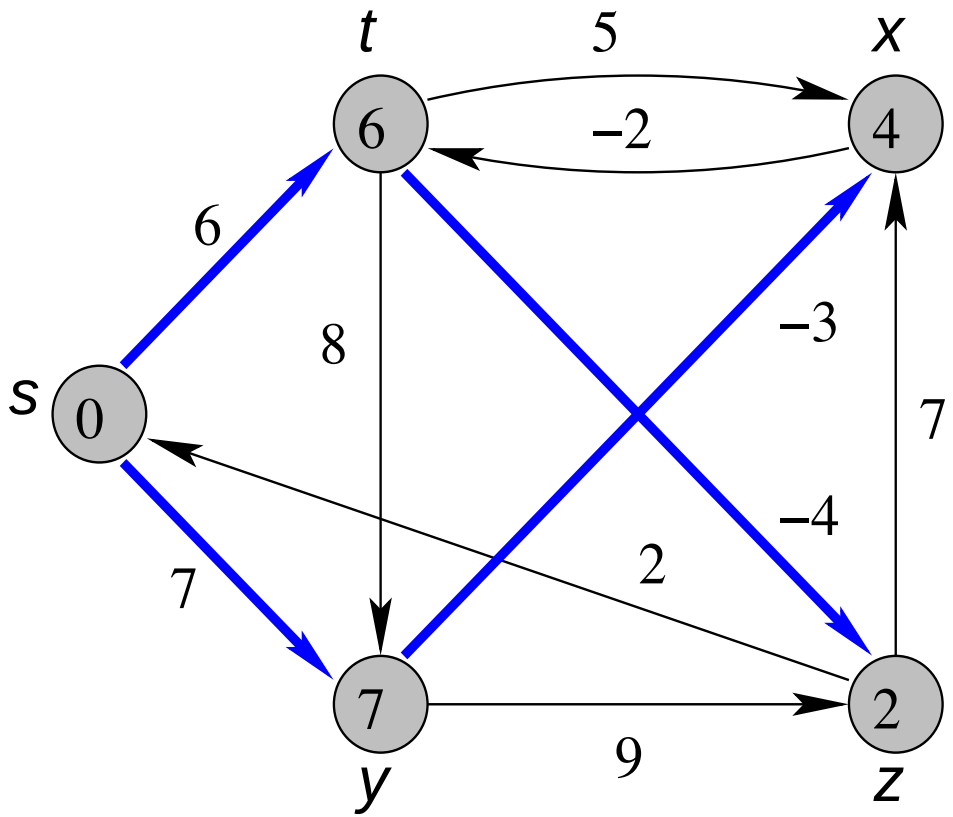
Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

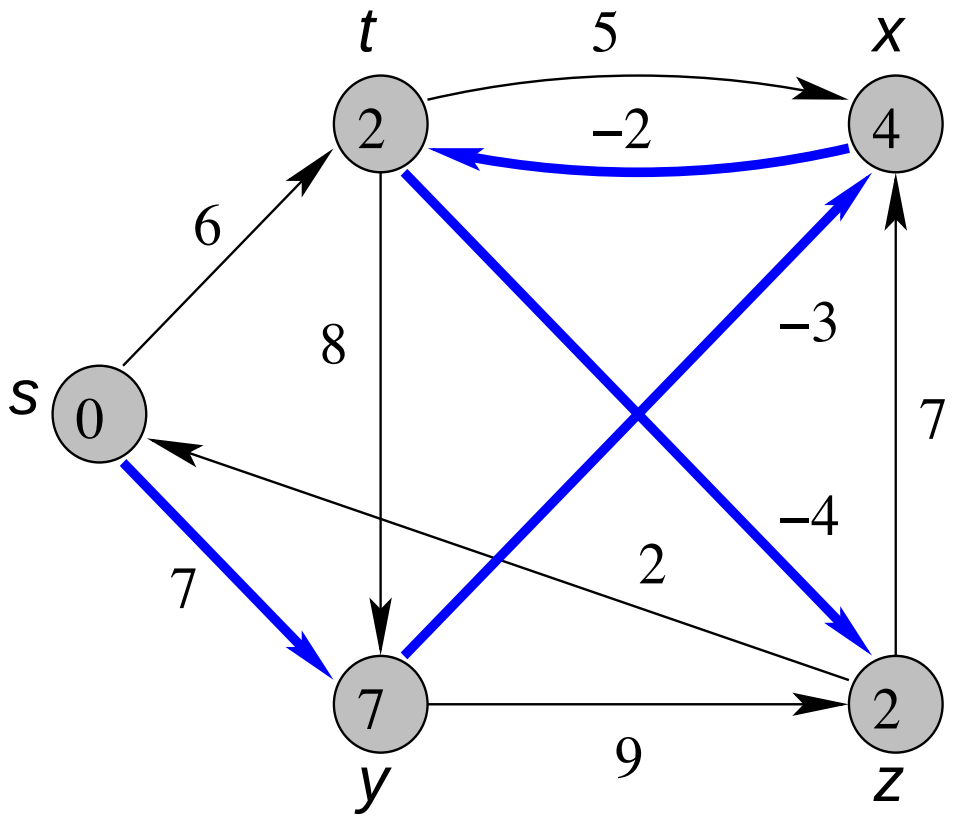
Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

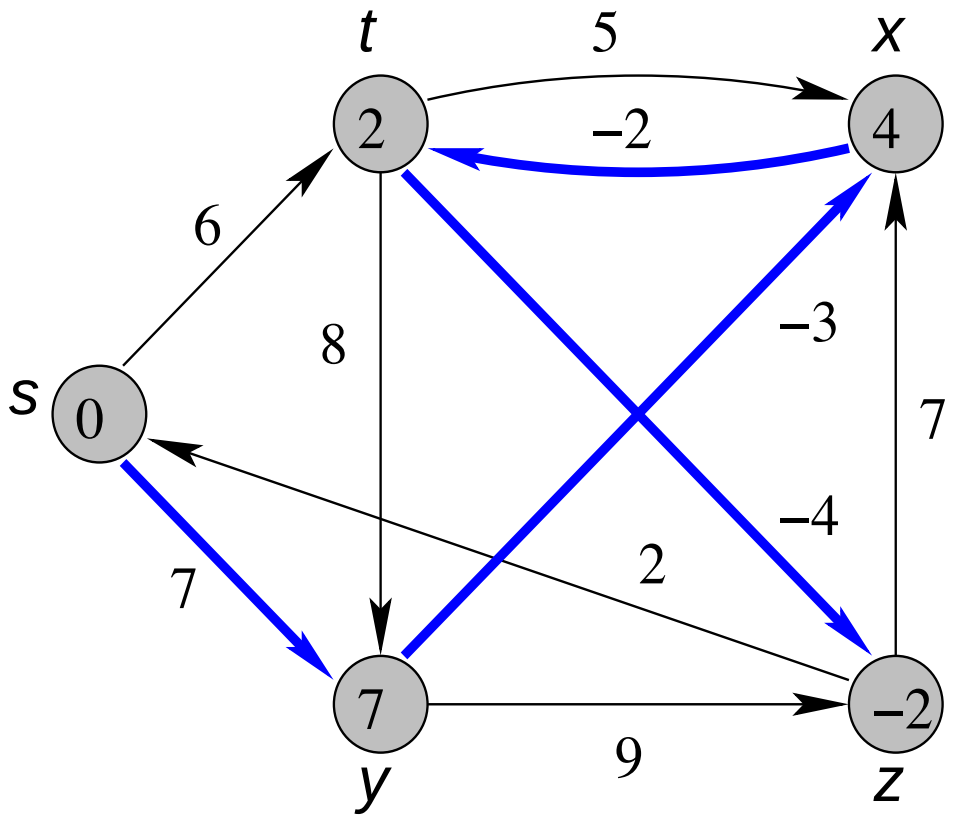
Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

Exemplo (CLRS)



Ordem:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

Caminhos mínimos entre todos os pares

O problema agora é dado um grafo (G, w) encontrar para todo para u, v de vértices um caminho mínimo de u a v.

Obviamente podemos executar $|V|$ vezes um algoritmo de Caminhos Mínimos com Mesma Origem.

- Se (G, w) não possui arestas negativas podemos usar o algoritmo de Dijkstra implementando a fila de prioridade como

  um vetor: $|V|.O(V^2 + E) = O(V^3 + VE)$ ou
  min-heap binário: $|V|.O(E \lg V) = O(VE \lg V)$ ou
  heap de Fibonacci: $|V|.O(V \lg V + E) = O(V^2 \lg V + VE)$.
- Se (G, w) possui arestas negativas podemos usar o algoritmo de Bellman-Ford: $|V|.O(VE) = O(V^2E)$.

O algoritmo de Floyd-Warshall

Veremos agora um método direto para resolver o problema que é assintoticamente melhor se G é denso.

O algoritmo de Floyd-Warshall baseia-se em programação dinâmica e resolve o problema em tempo $O(V^3)$.

O grafo (G, w) pode ter arestas negativas, mas suporemos que não contém ciclos negativos.

Vamos adotar a convenção de que (i, j) não é uma aresta de G então $w(i, j) = \infty$.

Estrutura de um caminho mínimo

Seja $P = (v_1, v_2, \ldots, v_k)$ um caminho (simples).

Um vértice intermediário de P é qualquer vértice de P distinto de v_1 e v_k, ou seja, em $\{v_2, \ldots, v_{k-1}\}$.

Para simplificar, suponha que $V = \{1, 2, \ldots, n\}$.

Estrutura de um caminho mínimo

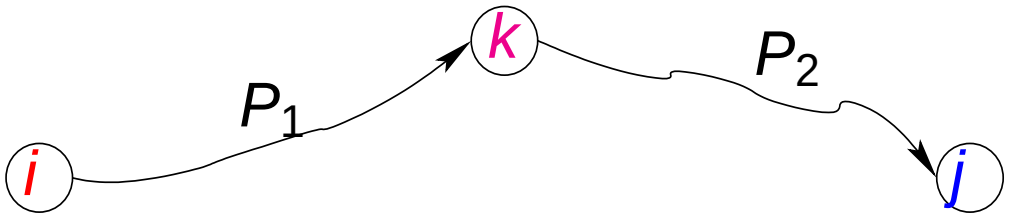
Sejam i e j dois vértices de G. Considere todos os caminhos cujos vértices intermediários pertencem a $\{1, \ldots, k\}$. Seja P um caminho mínimo entre todos eles.

O algoritmo de Floyd-Warshall explora a relação entre P e um caminho mínimo de i a j com vértices intermediários em $\{1, \ldots, k-1\}$.

Se k não é um vértice intermediário de P então P é um caminho mínimo de i a j com vértices intermediários em $\{1, \ldots, k-1\}$.

Estrutura de um caminho mínimo

- Se k é um vértice intermediário de P então P pode ser dividido em dois caminhos P_1 (com início em i e fim em k) e P_2 (com início em k e fim em j).



- P_1 é um caminho mínimo de i a k com vértices intermediários em $\{1, \ldots, k-1\}$
- P_2 é um caminho mínimo de k a j com vértices intermediários em $\{1, \ldots, k-1\}$.

Recorrência para caminhos mínimos

Seja $d_{ij}^{(k)}$ o peso de um caminho mínimo de i a j com vértices intermediários em $\{1, 2, \ldots, k\}$.

Quando $k = 0$ então $d_{ij}^{(0)} = w(i, j)$.

Temos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} w(i,j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Assim, queremos calcular a matrix $D^{(n)} = (d_{ij}^{(n)})$ com $d_{ij}^{(n)} = dist(i, j)$.

Algoritmo de Floyd-Warshall

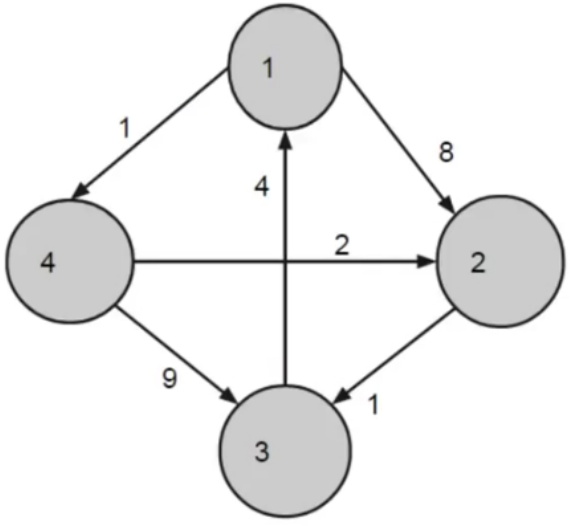
A entrada do algoritmo é a matriz $W = (w(i,j))$ com $n = |V|$ linhas e colunas.
A saída é a matriz $D^{(n)}$.

FLOYD-WARSHALL(W)

1 $D^{(0)} \leftarrow W$
2 **para** $k \leftarrow 1$ **até** n **faça**
3 **para** $i \leftarrow 1$ **até** n **faça**
4 **para** $j \leftarrow 1$ **até** n **faça**
5 $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
6 **devolva** $D^{(n)}$

Complexidade: $O(V^3)$

Algoritmo de Floyd-Warshall – exemplo

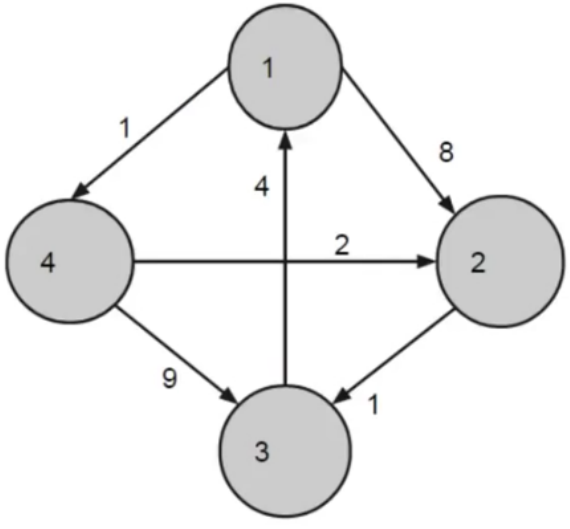


K = 0

	1	2	3	4
1	0	8	inf	1
2	inf	0	1	inf
3	4	inf	0	inf
4	inf	2	9	0

K = 1

	1	2	3	4
1	0	8	inf	1
2	inf	0	1	inf
3	4	inf	0	inf
4	inf	2	9	0



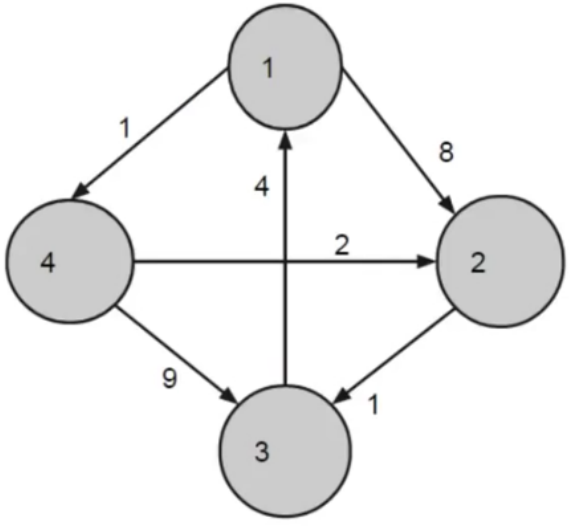
K = 2

	1	2	3	4
1	0	8	inf	1
2	inf	0	1	inf
3	4	12	0	5
4	inf	2	9	0

K = 3

	1	2	3	4
1	0	8	9	1
2	inf	0	1	inf
3	4	12	0	5
4	inf	2	3	0

Algoritmo de Floyd-Warshall – exemplo



K = 4

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

	1	2	3	4
1	0	3	4	1
2	5	0	1	6
3	4	7	0	5
4	7	2	3	0

Como encontrar os caminhos?

O algoritmo precisa devolver também uma matriz $\Pi = (\pi_{ij})$ tal que $\pi_{ij} = \text{NIL}$ se $i = j$ ou se não existe caminho de i a j, e caso contrário, π_{ij} é o predecessor de de j em algum caminho mínimo a partir de i.

Podemos computar os predecessores ao mesmo tempo que o algoritmo calcula as matrizes $D^{(k)}$. Determinamos uma seqüência de matrizes $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$ e $\pi_{ij}^{(k)}$ é o predecessor de j em um caminho mínimo a partir de i com vértices intermediários em $\{1, 2, \ldots, k\}$.

Quando $k = 0$ temos

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ ou } w(i,j) = \infty, \\ i & \text{se } i \neq j \text{ e } w(i,j) < \infty. \end{cases}$$

Como encontrar os caminhos?

Para $k \geq 1$ procedemos da seguinte forma. Considere um caminho mínimo P de i a j.

Se k não aparece em P então tomamos como predecessor de j o predecessor de j em um caminho mínimo de i a j com vértices intermediários em $\{1, 2, \ldots, k-1\}$.

Caso contrário, tomamos como predecessor de j o predecessor de j em um caminho mínimo de k a j com vértices intermediários em $\{1, 2, \ldots, k-1\}$.
Formalmente,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Exercício. Incorpore esta parte no algoritmo!

Passos do projeto de algoritmos gulosos: resumo

1. Formule o problema como um **problema de otimização** no qual uma escolha é feita, restando-nos então resolver um único subproblema a resolver.
2. Provar que existe sempre uma solução ótima do problema que atende à **escolha gulosa**, ou seja, a escolha feita pelo algoritmo guloso é segura.
3. Demonstrar que, uma vez feita a escolha gulosa, o que resta a resolver é um subproblema tal que se combinarmos a resposta ótima deste subproblema com o(s) elemento(s) da escolha gulosa, chega-se à solução ótima do problema original.
   Esta é a parte que requer mais engenhosidade!
   Normalmente a prova começa com uma solução ótima *genérica* e a modificamos até que ela inclua o(s) elemento(s) identificados pela escolha gulosa.