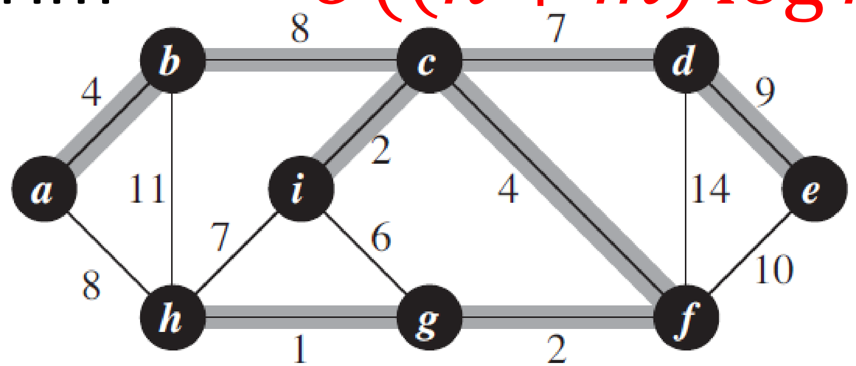
Introdução à Teoria dos Grafos

Prof. Alexandre Noma

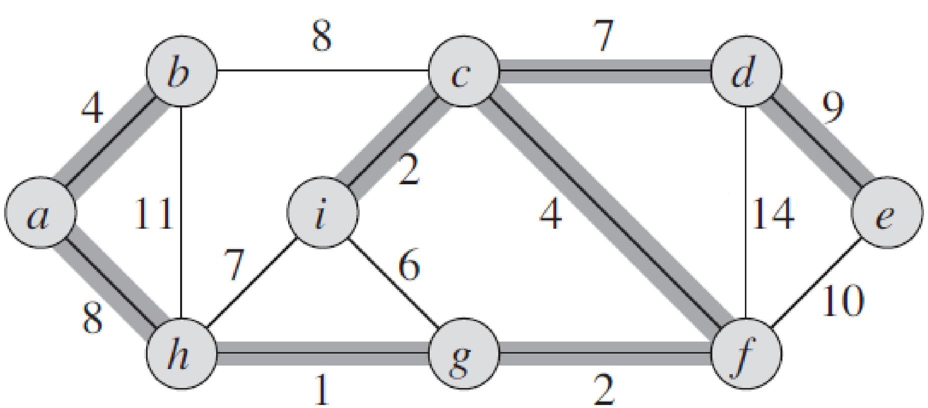
Aula passada: Prim vs Kruskal

- Prim $O((n+m)\log n)$



$$w(T) = 4+8+7+9+$$
$$+4+2+1+2 = 37$$

- Kruskal $\Omega(m \log m)$



$$w(T) = 1+2+2+4+$$
$$+4+7+8+9 = 37$$

Conjuntos disjuntos

- Coleção $S = \{S_1, S_1, \ldots, S_k\}$ de conjuntos disjuntos.
- Cada conjunto é identificado por um representante.
- Ex. S_1 S_2 S_3 S_4 S_5
  - {**a**,b,c,d} {e,**g**} {**f**} {h,**i**} {**j**}

- Operações
  - MakeSet(x): Cria novo conjunto com único elemento x.
  - FindSet(x): Devolve o representante de S_x.
  - Union(x, y): Une dois conjuntos S_x e S_y.

Consumo de tempo

MST-Kruskal(G,w)	**Consumo de tempo**
`T = Ø`	???
`para cada vértice u em  G.V  faça`	???
`    MakeSet(u)`	???
`ordenar arestas G.E por peso (crescente)`	???
`para cada aresta uv em G.E (ordenada) faça`	???
`    se FindSet(u) != FindSet(v)`	???
`      entao Union(u,v)`	???
`            T = T ∪ {uv}`	???
`devolva T`	???

Total: $T(n,m) = ???$

Consumo de tempo

MST-Kruskal(G,w)	**Consumo de tempo**
T = Ø	O(1)
para cada vértice u em G.V faça	O(n)
MakeSet(u)	???
ordenar arestas G.E por peso (crescente)	O(m log m)
para cada aresta uv em G.E (ordenada) faça	O(m)
se **FindSet**(u) != **FindSet**(v)	???
entao **Union**(u,v)	???
T = T U {uv}	O(m)*O(1)
devolva T	O(1)

Total: **T(n,m) = ???**

Conjuntos disjuntos

- Operações
  - MakeSet(x): consome O(1) unidades de tempo.
  - FindSet(x): Devolve o representante de S_x.
  - Union(x, y): Une dois conjuntos

- m operações FindSet(x) e Union(x,y) consomem $O(m \log m)$ unidades de tempo

Livro: Introduction to algorithms
(CLRS: Cormen, Leiserson, Rivest, Stein)

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on how we implement the disjoint-set data structure. We assume that we use the disjoint-set-forest implementation of Section 21.3 with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. Initializing the set A in line 1 takes $O(1)$ time, and the time to sort the edges in line 4 is $O(E \lg E)$. (We will account for the cost of the $|V|$ MAKE-SET operations in the **for** loop of lines 2–3 in a moment.) The **for** loop of lines 5–8 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. Along with the $|V|$ MAKE-SET operations, these take a total of $O((V + E)\,\alpha(V))$ time, where α is the very slowly growing function defined in Section 21.4. Because we assume that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\,\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$.

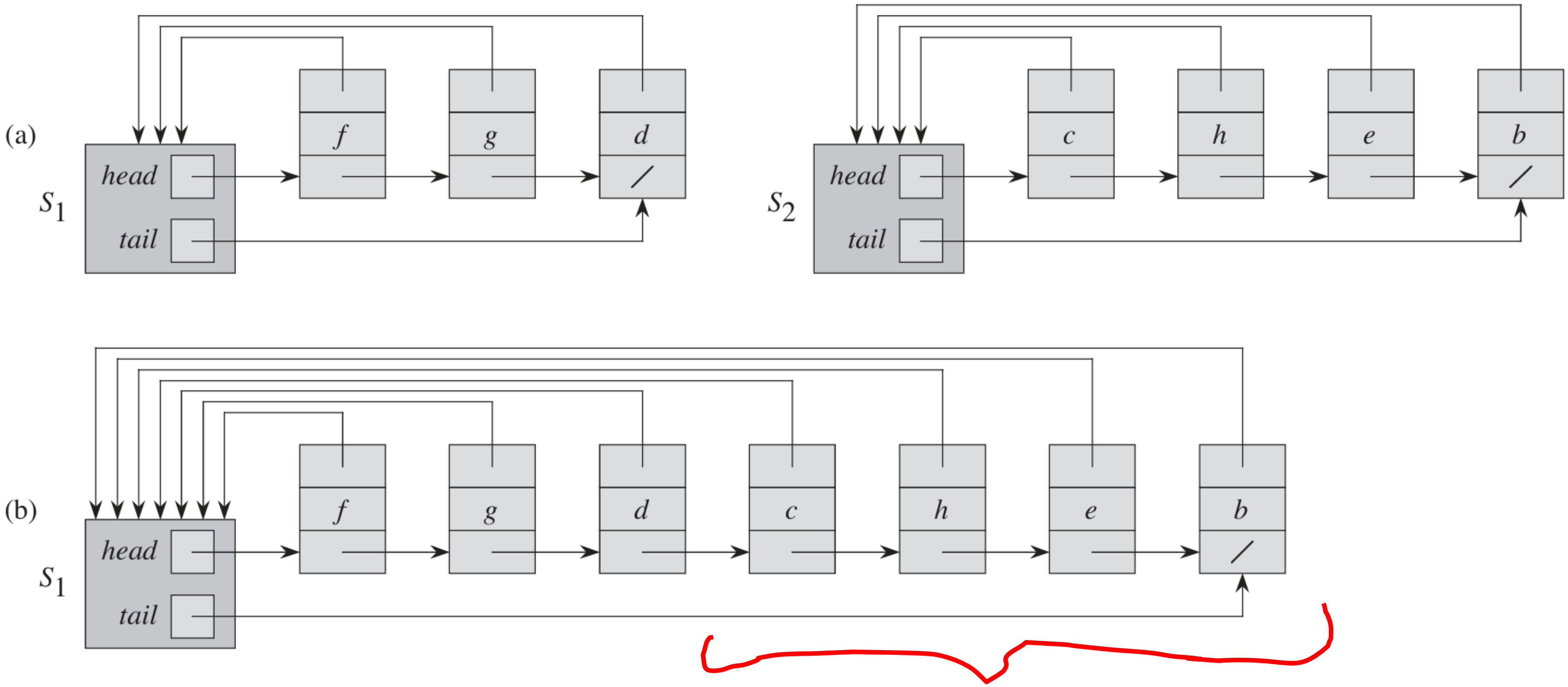


Figure 21.2 **(a)** Linked-list representations of two sets. Set S_1 contains members d, f, and g, with representative f, and set S_2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers *head* and *tail* to the first and last objects, respectively. **(b)** The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f. The set object for e's list, S_2, is destroyed.

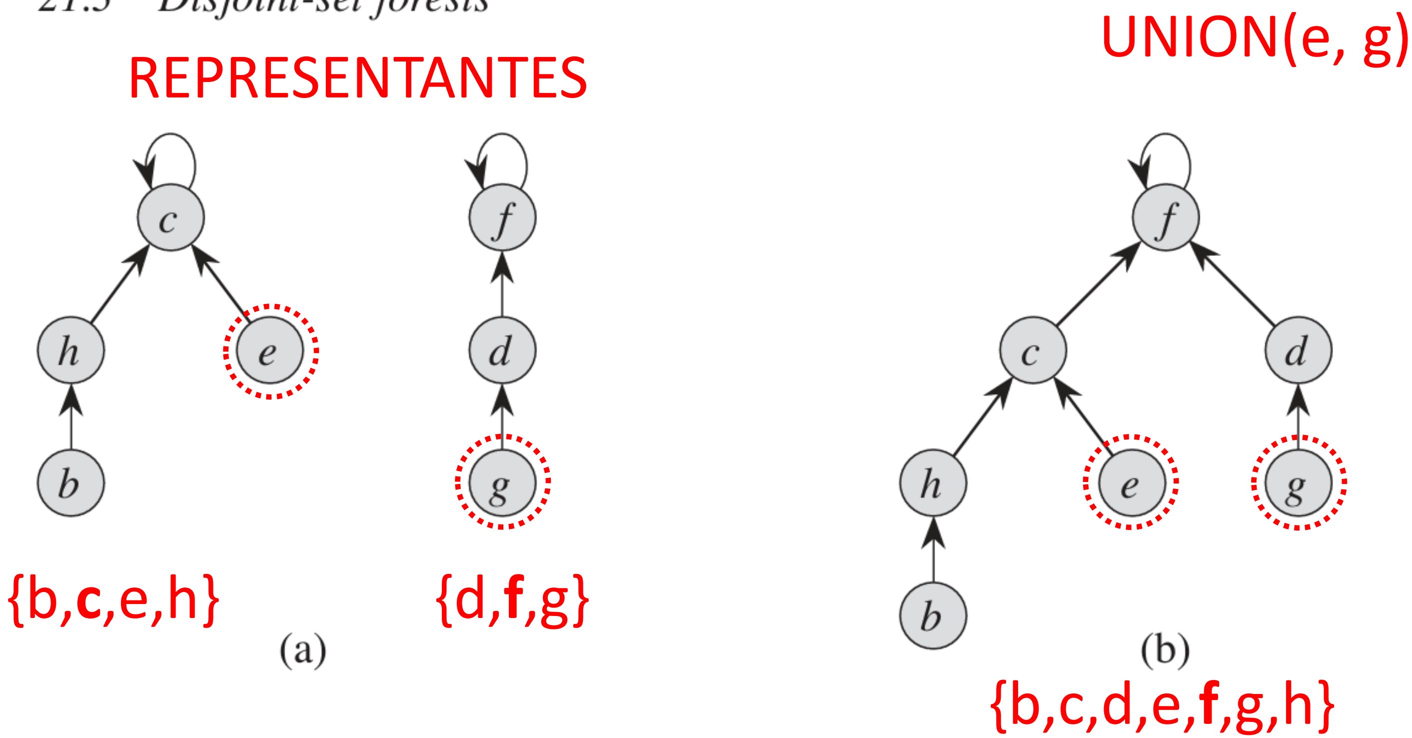


Figure 21.4 A disjoint-set forest. **(a)** Two trees representing the two sets of Figure 21.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. **(b)** The result of UNION(e, g).

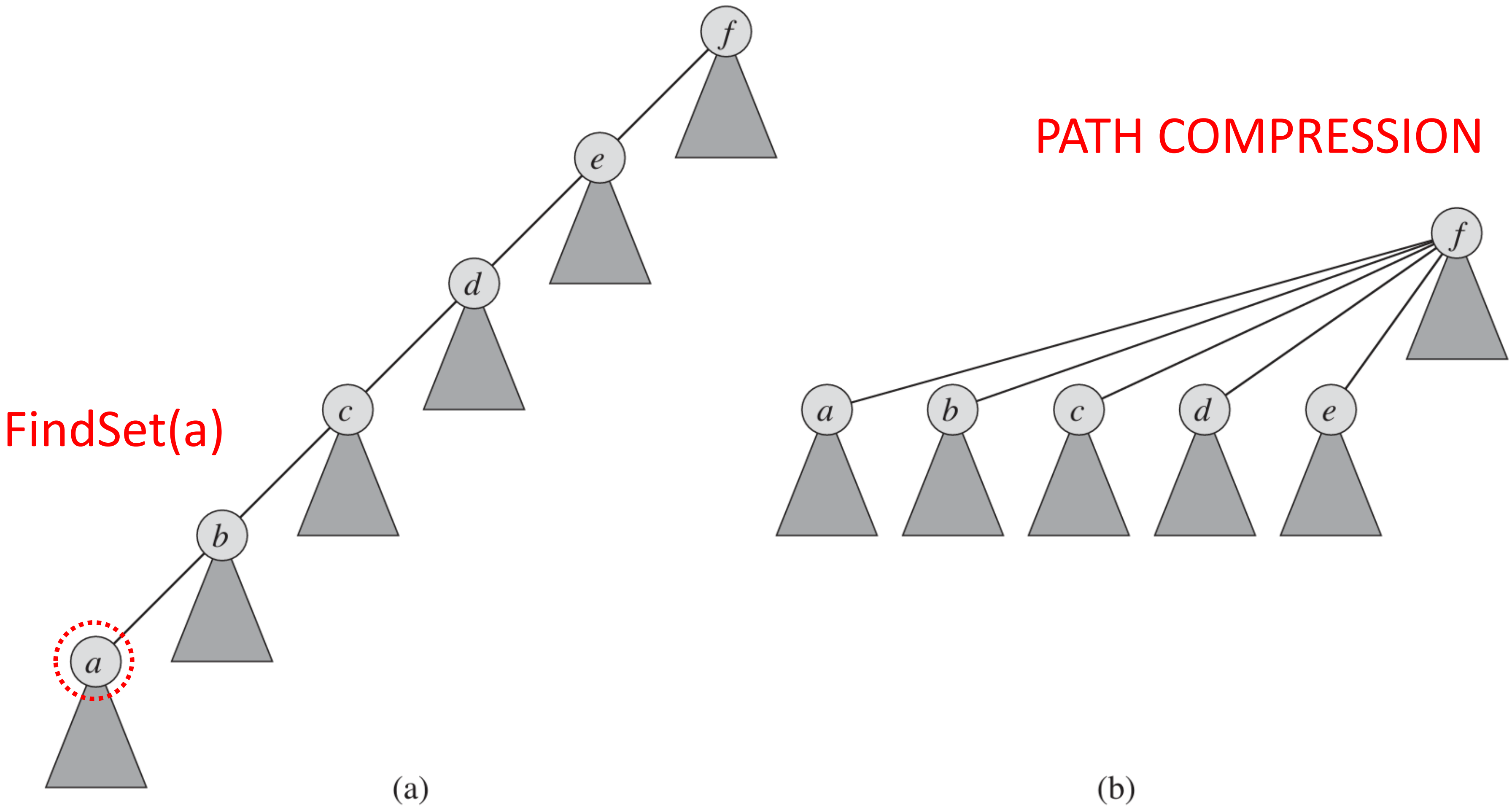


Figure 21.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

Conjuntos disjuntos

- Operações
  - MakeSet(x): consome O(1) unidades de tempo.
  - FindSet(x): Devolve o representante de S_x.
  - Union(x, y): Une dois conjuntos

- m operações FindSet(x) e Union(x,y) consomem $O(m \log m)$ unidades de tempo

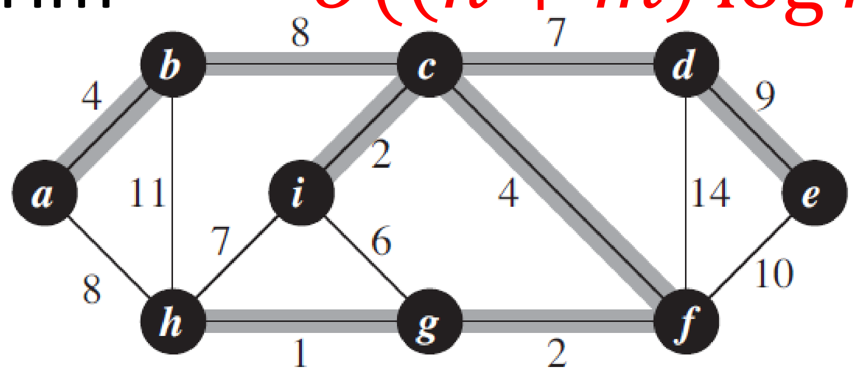
Consumo de tempo

MST-Kruskal(G,w)	Consumo de tempo
T = Ø	O(1)
para cada vértice u em G.V faça	O(n)
MakeSet(u)	O(n)*O(1)
ordenar arestas G.E por peso (crescente)	O(m log m)
para cada aresta uv em G.E (ordenada) faça	O(m)
se FindSet(u) != FindSet(v)	$O(m \log m)$
entao Union(u,v)	
T = T ∪ {uv}	O(m)*O(1)
devolva T	O(1)

Total: T(n,m) = O(m log m)

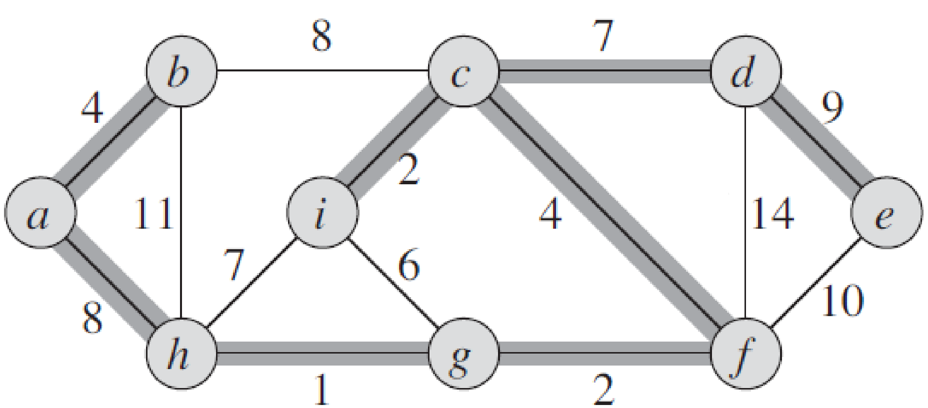
Aula passada: Prim vs Kruskal

- Prim $O((n+m)\log n)$



$w(T) = 4+8+7+9+$
$+4+2+1+2 = 37$

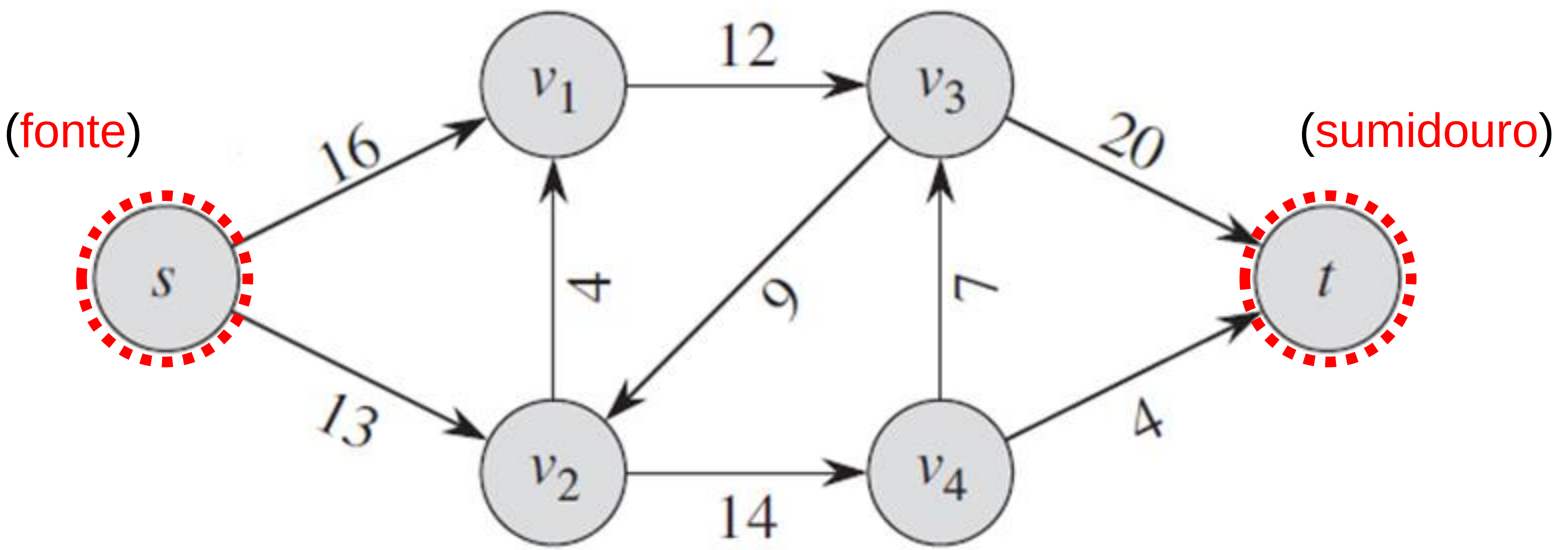
- Kruskal $O(m \log m)$



$w(T) = 1+2+2+4+$
$+4+7+8+9 = 37$

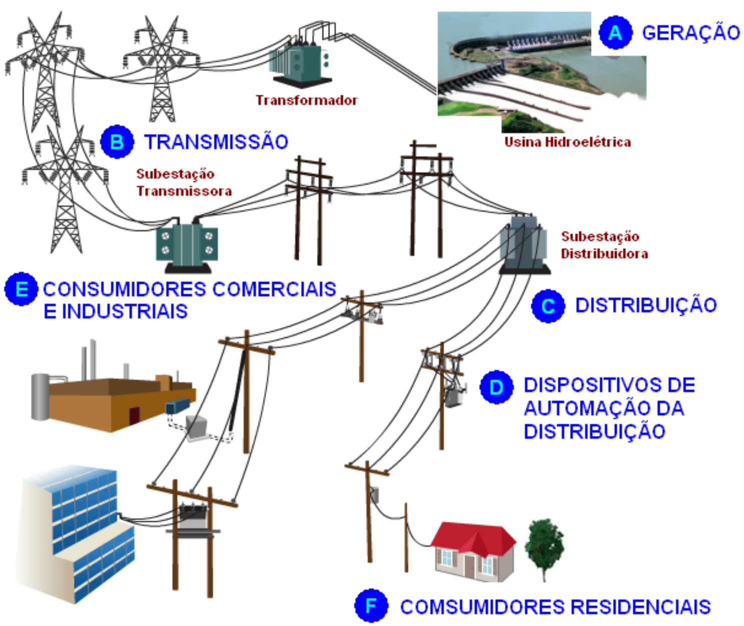
Hoje

- Problema: Qual é o fluxo máximo de **s** a **t**?



Fluxo máximo

- Exemplos:
  - informação
  - corrente elétrica
  - líquido, mercadoria, etc



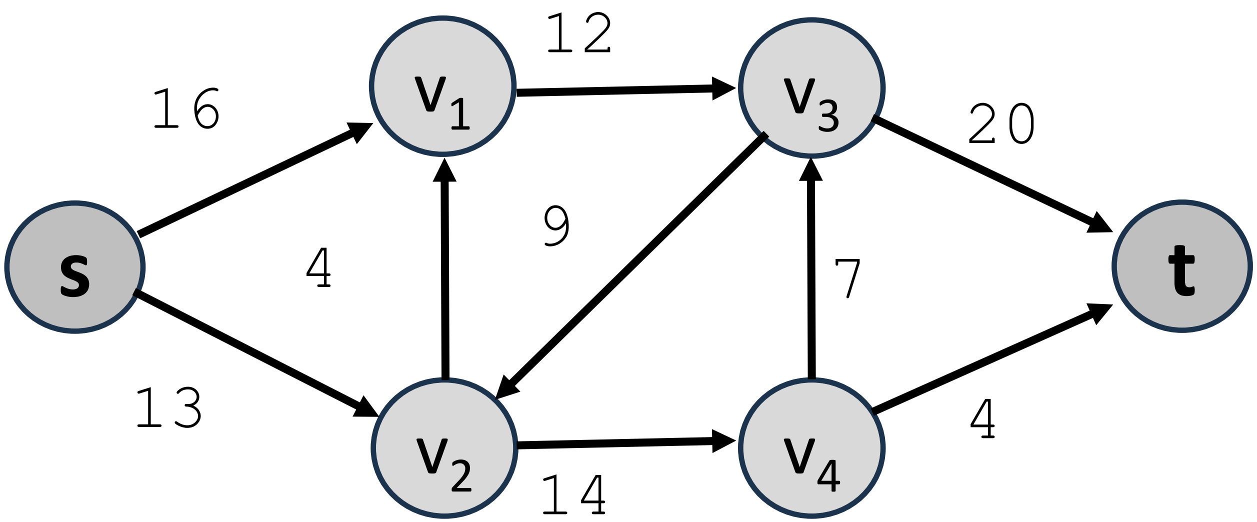
Fluxo máximo

- Ford-Fulkerson
  - Fluxo / Capacidade
  - Caminho aumentante
  - Rede residual

Exemplo

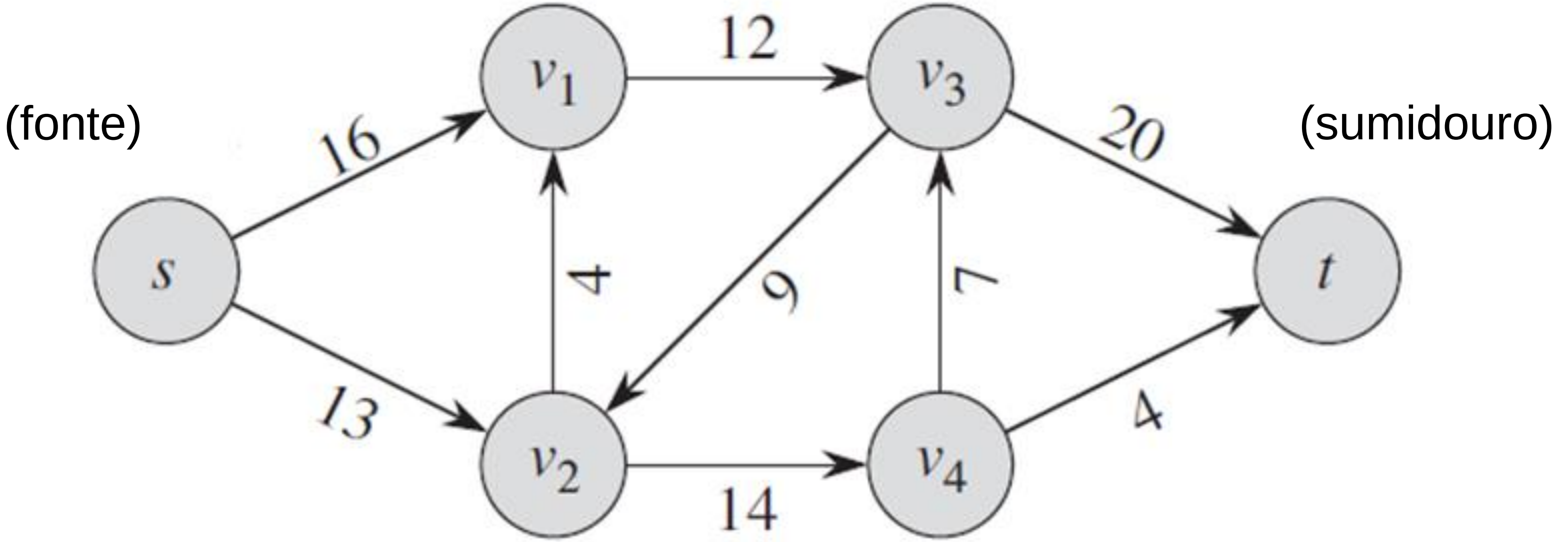
```
Ford-Fulkerson-Method(G, s, t):

 1. Inicialmente, fluxo  f = 0
 2. Enquanto existir um caminho aumentante P:
 3.       Incremente o fluxo f (usando P)
 4. Devolva f
```



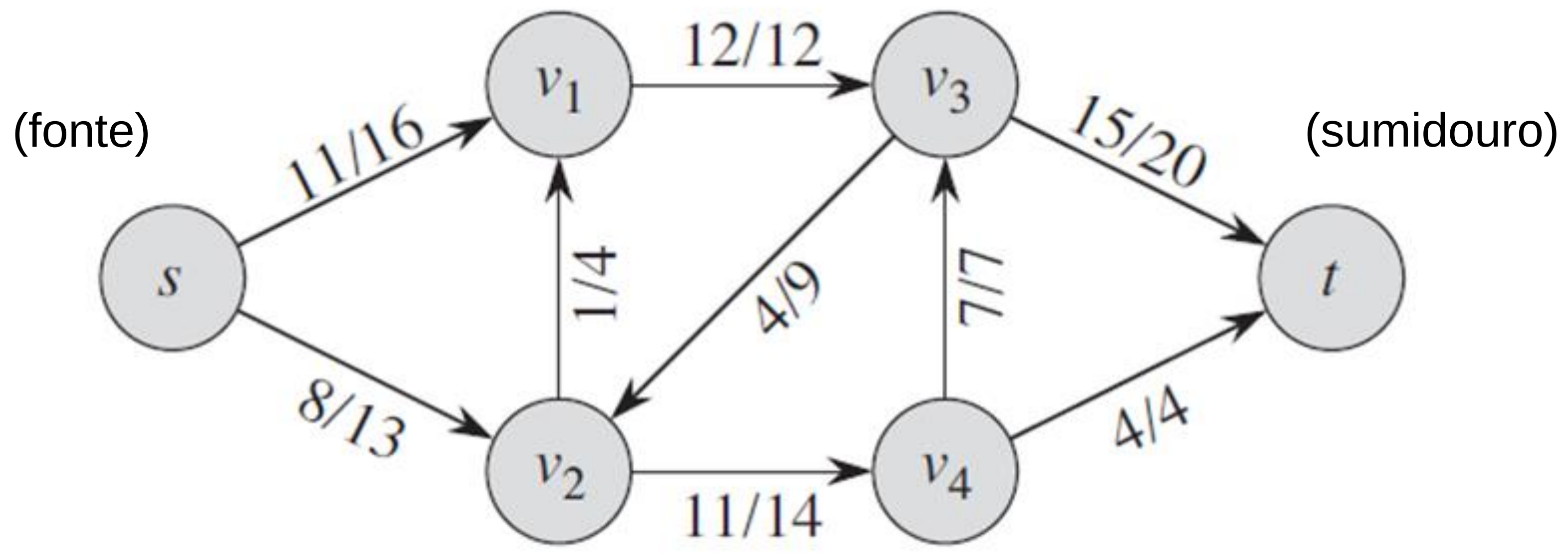
Fluxo máximo?

- Capacidade



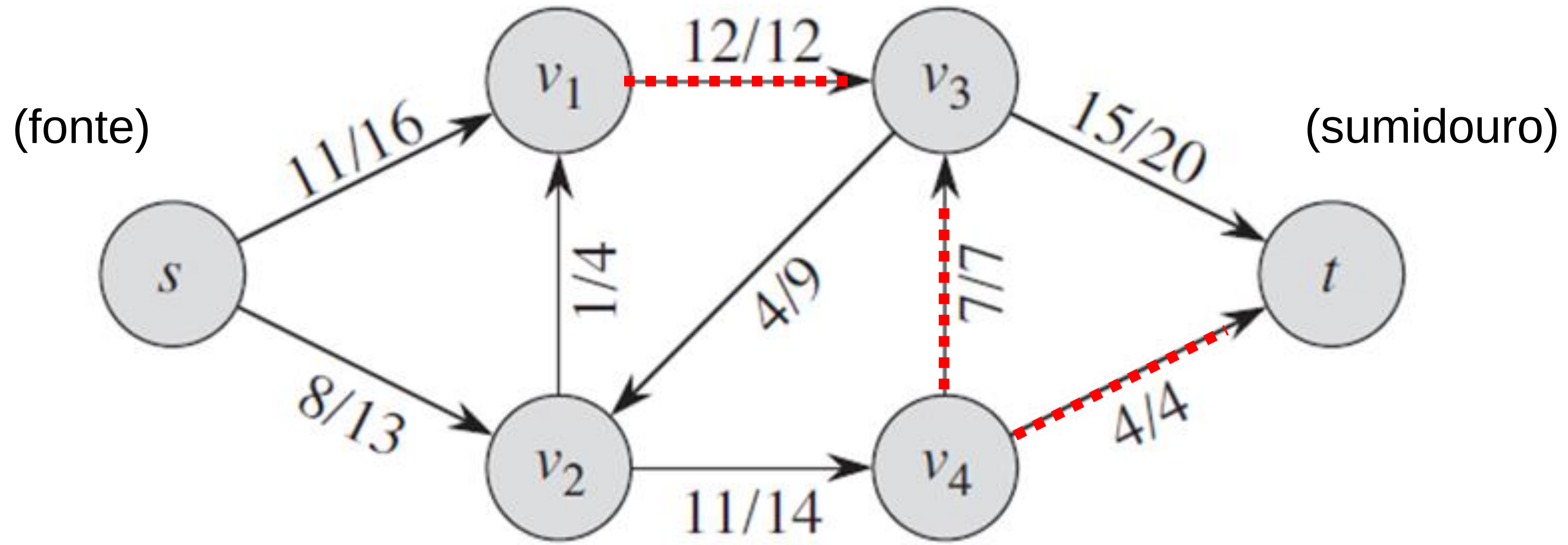
Fluxo máximo?

- Fluxo vs Capacidade



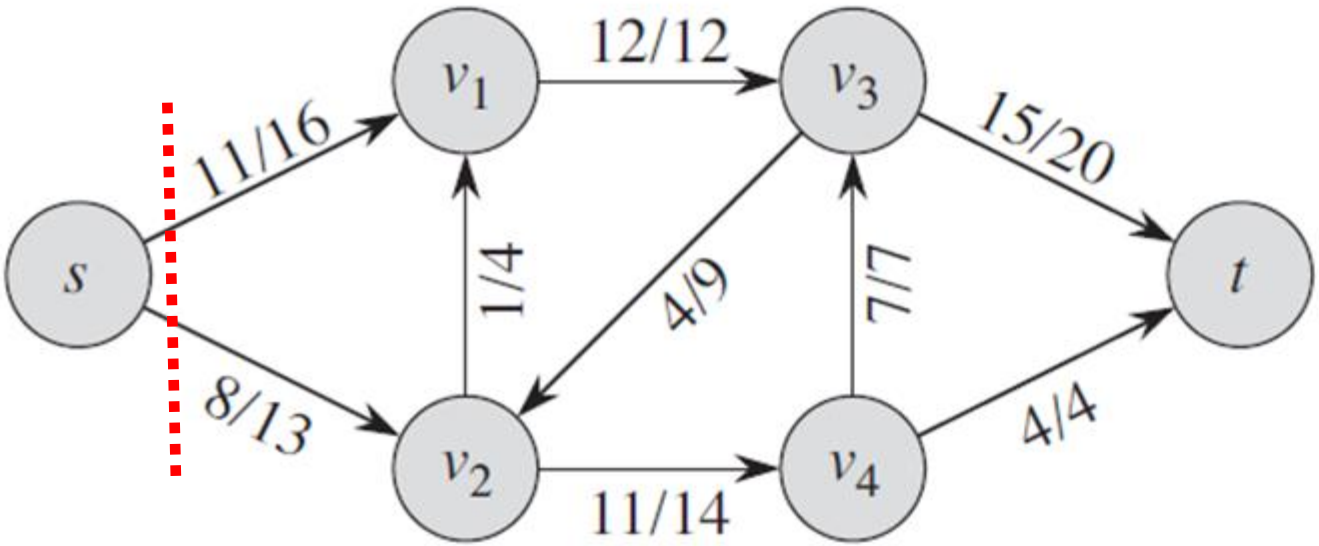
Fluxo máximo?

- Fluxo vs Capacidade: arcos saturados



Conservação do fluxo

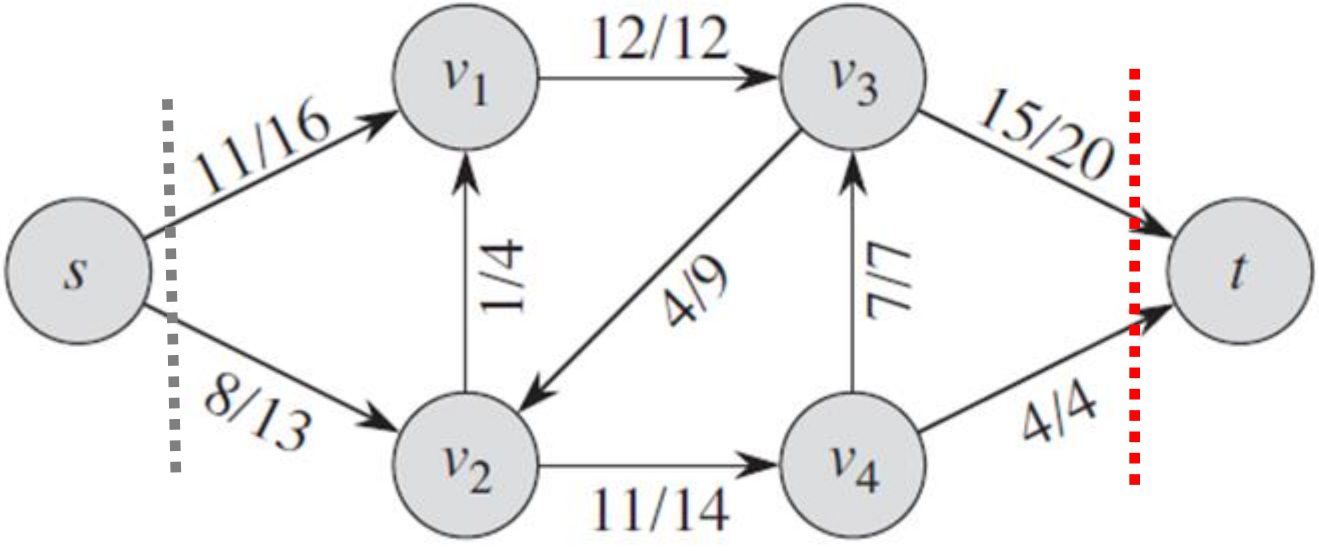
- Corte (S,T)



f = 11+8 = 19

Conservação do fluxo

- Corte (S,T)

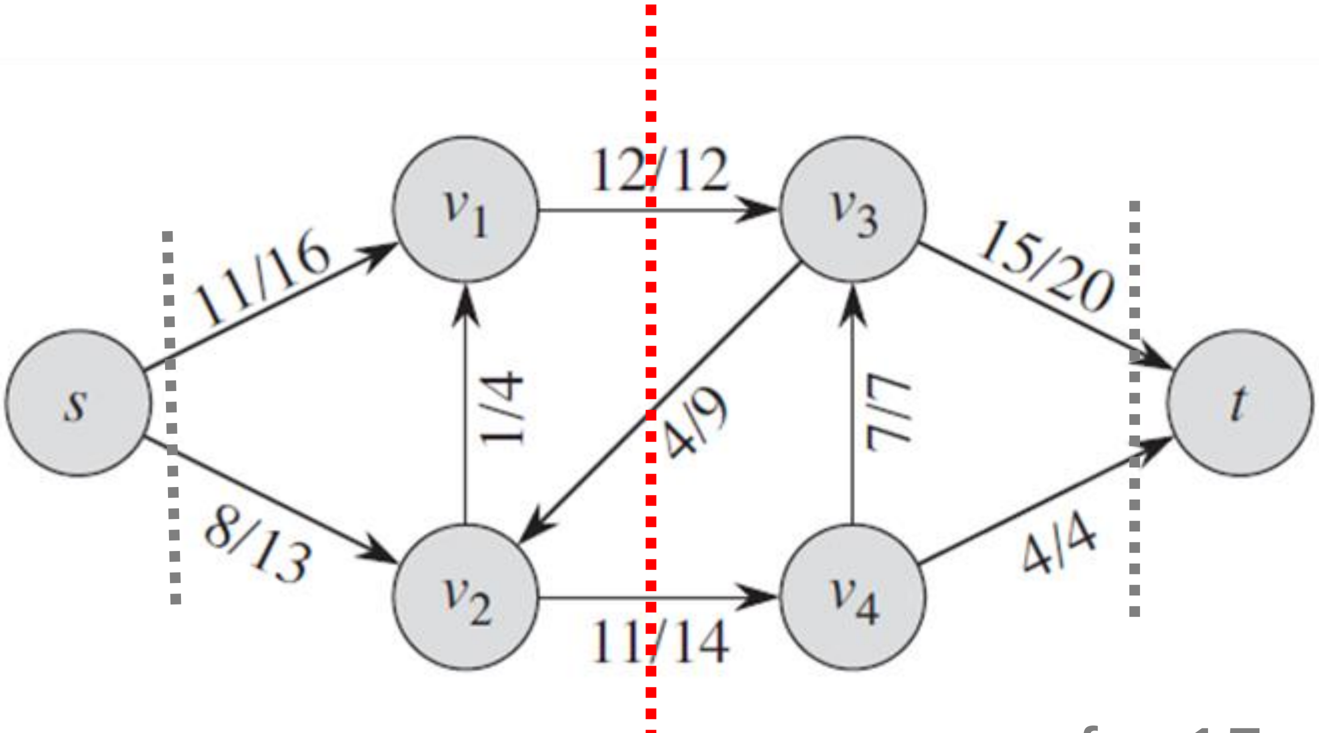


f = 11+8 = 19

f = 15+4 = 19

Conservação do fluxo

- Corte (S,T)

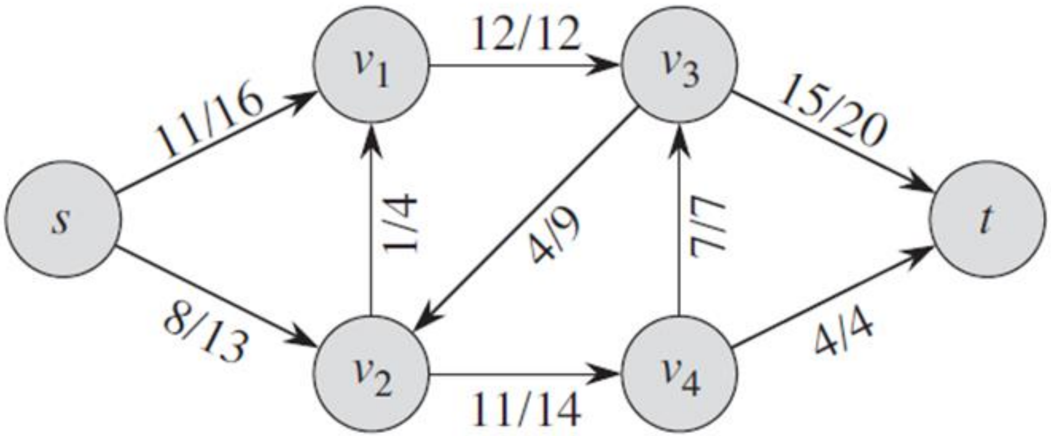


f = 11+8 = 19

f = 12+11-**4** = 19

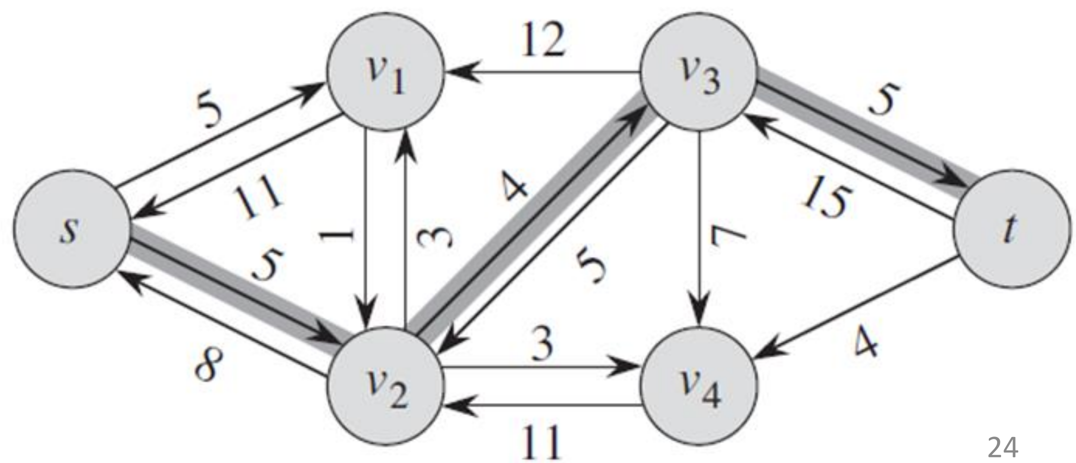
f = 15+4 = 19

Caminho aumentante



original

residual



Exercícios Programas

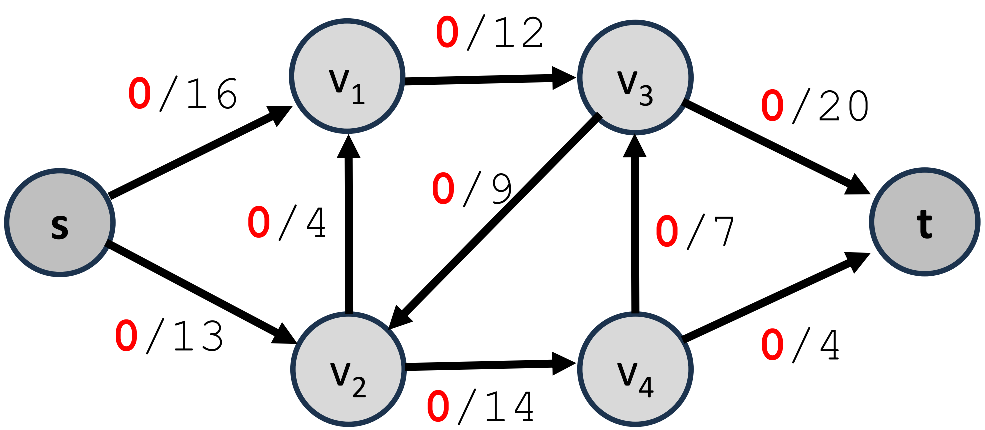
- 15-caminhoAumentante.py
- 16-aumentaFluxo.py

Algoritmo

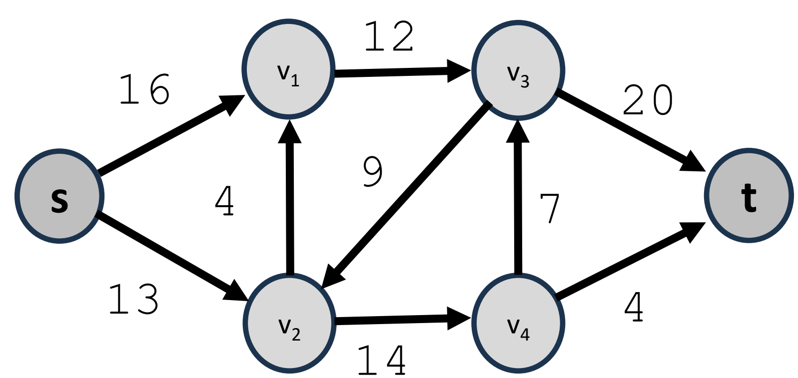
- Ford-Fulkerson(G, s, t)
  - Entrada: um grafo **G** ponderado,
    um vértice inicial **s**
    e um vértice final **t**
  - Saída: fluxo máximo de **s** até **t**

- Atributo (arco)
  - uv.**f**: fluxo

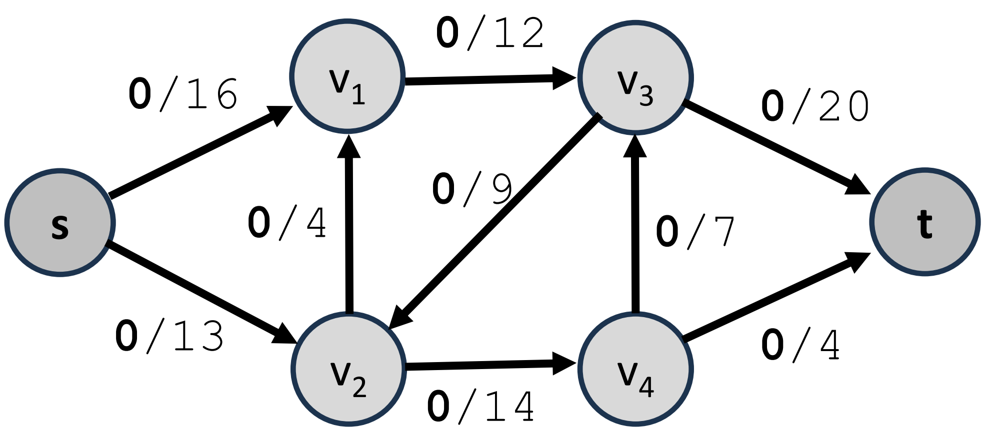
```
Ford-Fulkerson(G, s, t):
para cada aresta uv em G.E
    uv.f = 0
enqto existir caminho aumentante P de s a t:
    para cada uv em P:
        se uv em G.E:
            uv.f = uv.f + cf(P)        # ida
        senão:
            vu.f = vu.f - cf(P)        # volta
```



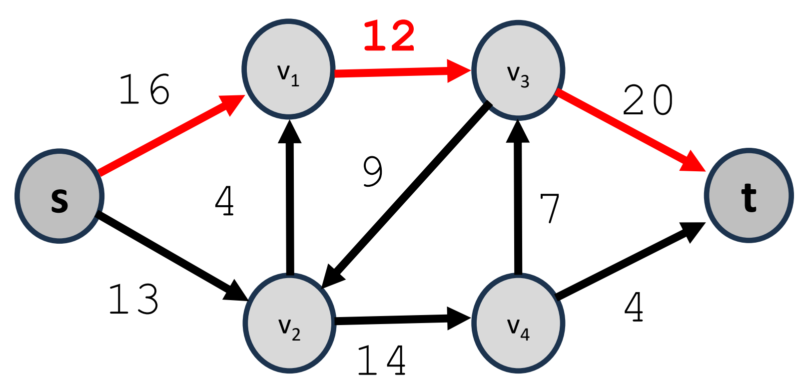
residual



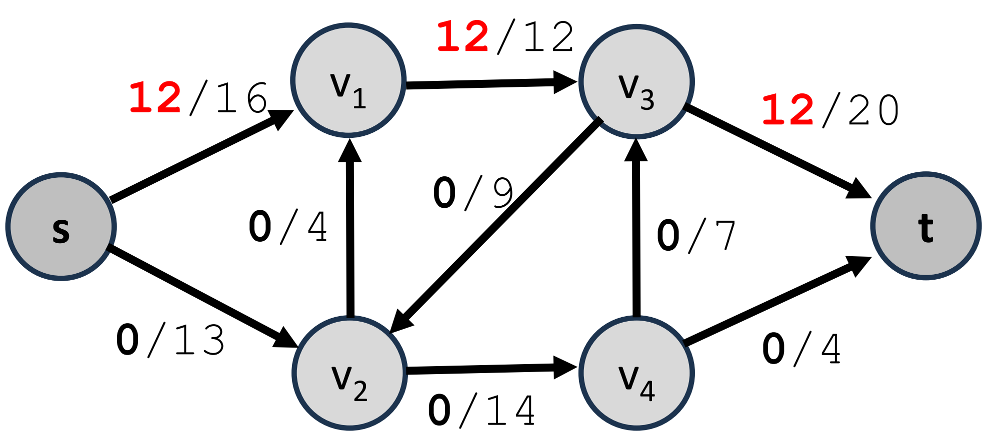
```
Ford-Fulkerson(G, s, t):
  para cada aresta uv em G.E
      uv.f = 0
  enqto existir caminho aumentante P de s a t:
      para cada uv em P:
          se uv em G.E:
              uv.f = uv.f + cf(P)          # ida
          senão:
              vu.f = vu.f - cf(P)          # volta
```



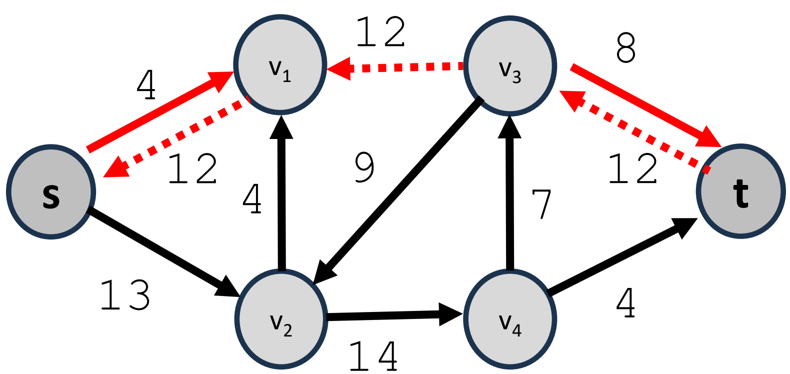
residual



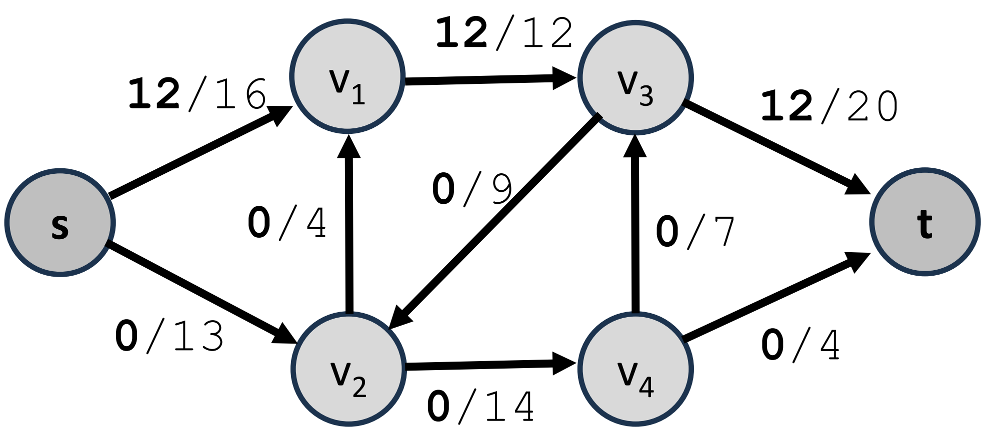
```
Ford-Fulkerson(G, s, t):
  para cada aresta uv em G.E
        uv.f = 0
  enqto existir caminho aumentante P de s a t:
        para cada uv em P:
              se uv em G.E:
                    uv.f = uv.f + cf(P)        # ida
              senão:
                    vu.f = vu.f - cf(P)        # volta
```



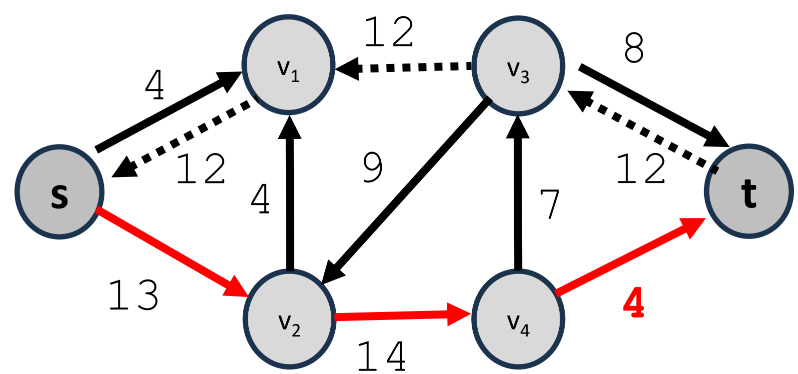
residual



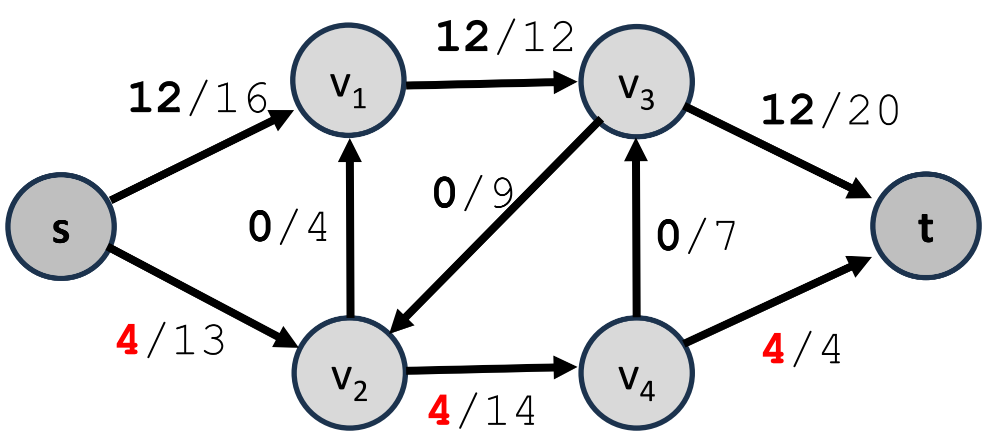
```
Ford-Fulkerson(G, s, t):
  para cada aresta uv em G.E
       uv.f = 0
  enqto existir caminho aumentante P de s a t:
       para cada uv em P:
            se uv em G.E:
                 uv.f = uv.f + cf(P)        # ida
            senão:
                 vu.f = vu.f - cf(P)        # volta
```



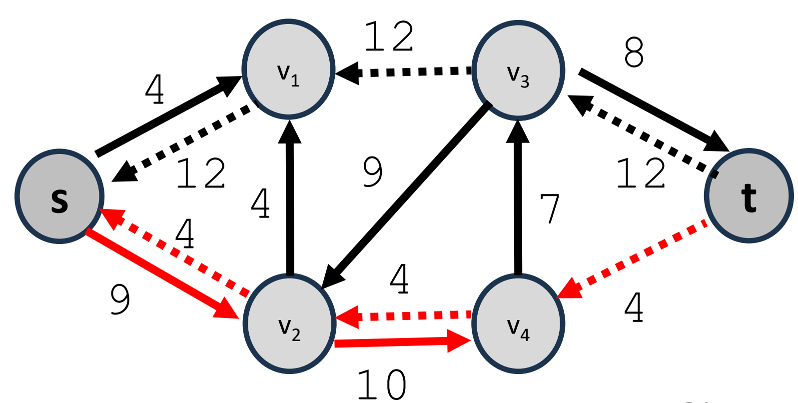
residual



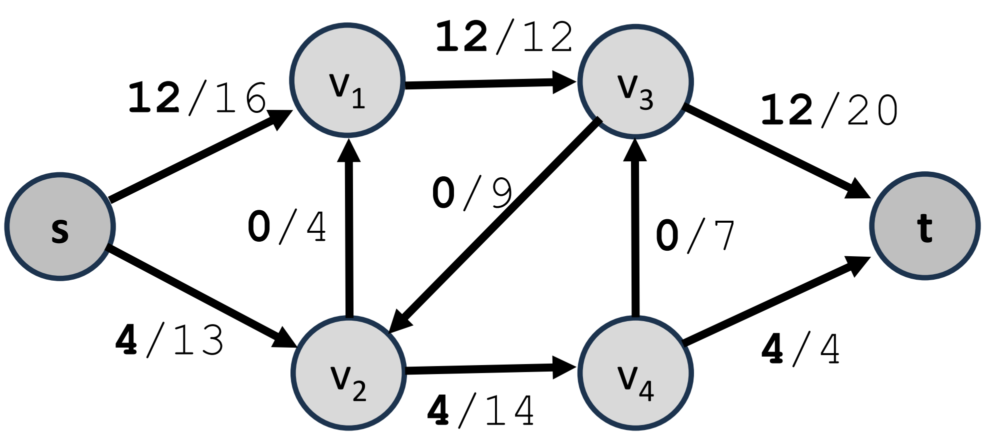
```
Ford-Fulkerson(G, s, t):
    para cada aresta uv em G.E
        uv.f = 0
    enqto existir caminho aumentante P de s a t:
        para cada uv em P:
            se uv em G.E:
                uv.f = uv.f + cf(P)        # ida
            senão:
                vu.f = vu.f - cf(P)        # volta
```



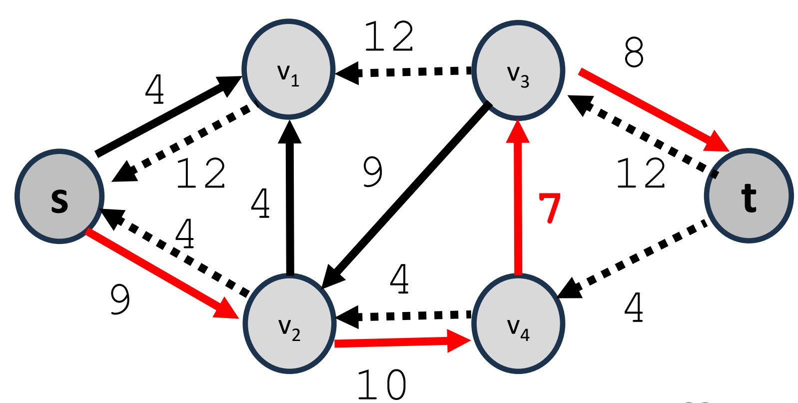
residual



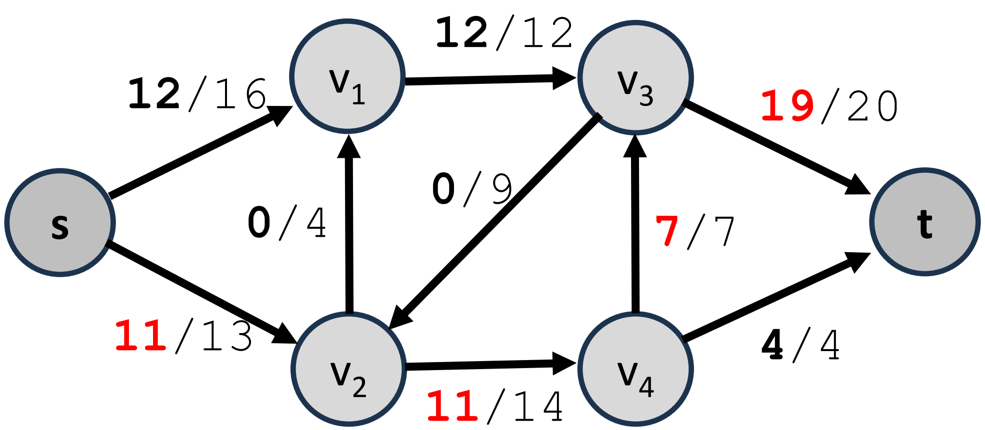
```
Ford-Fulkerson(G, s, t):
    para cada aresta uv em G.E
        uv.f = 0
    enqto existir caminho aumentante P de s a t:
        para cada uv em P:
            se uv em G.E:
                uv.f = uv.f + cf(P)          # ida
            senão:
                vu.f = vu.f - cf(P)          # volta
```



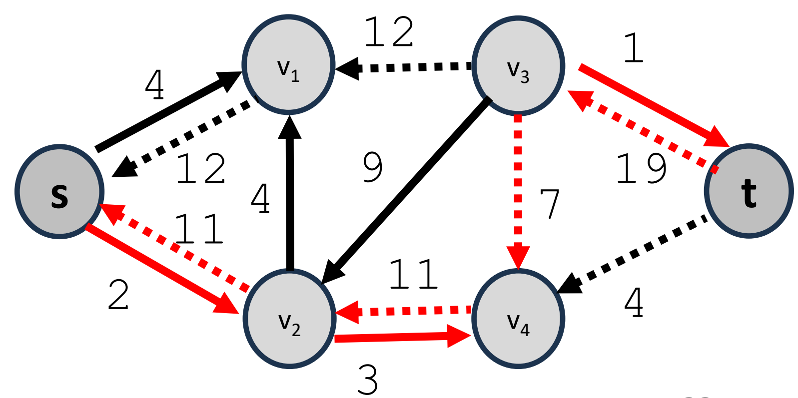
residual



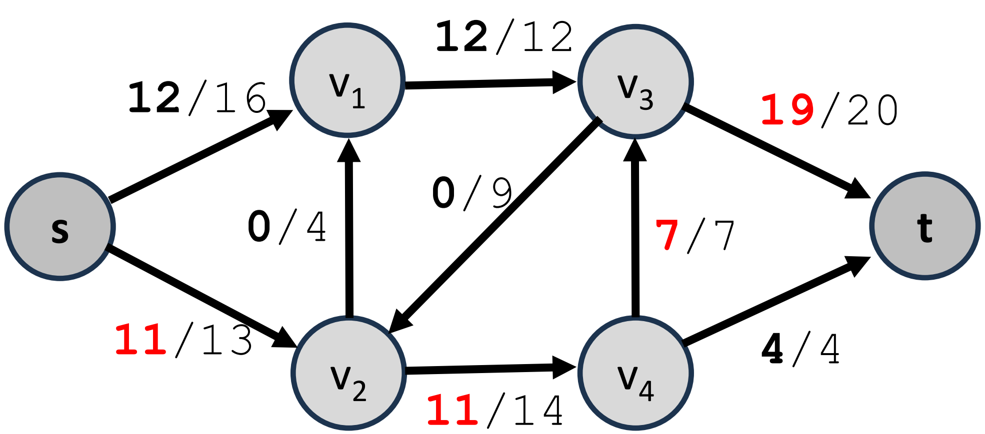
```
Ford-Fulkerson(G, s, t):
    para cada aresta uv em G.E
        uv.f = 0
    enqto existir caminho aumentante P de s a t:
        para cada uv em P:
            se uv em G.E:
                uv.f = uv.f + cf(P)        # ida
            senão:
                vu.f = vu.f - cf(P)        # volta
```



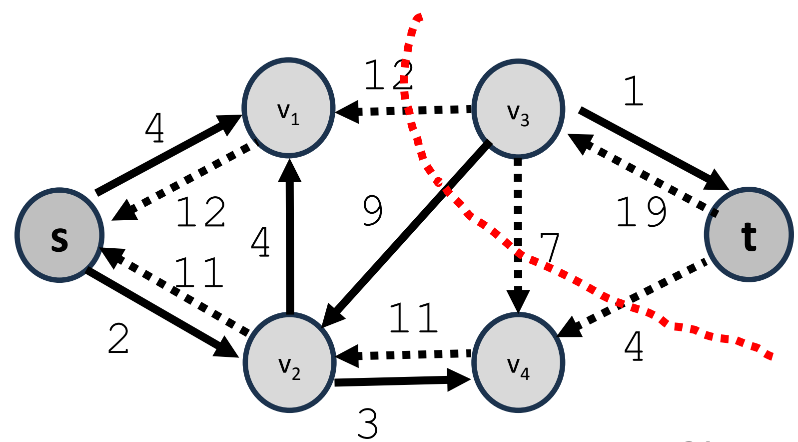
residual



```
Ford-Fulkerson(G, s, t):
    para cada aresta uv em G.E
        uv.f = 0
    enqto existir caminho aumentante P de s a t:
        para cada uv em P:
            se uv em G.E:
                uv.f = uv.f + cf(P)        # ida
            senão:
                vu.f = vu.f - cf(P)        # volta
```



residual



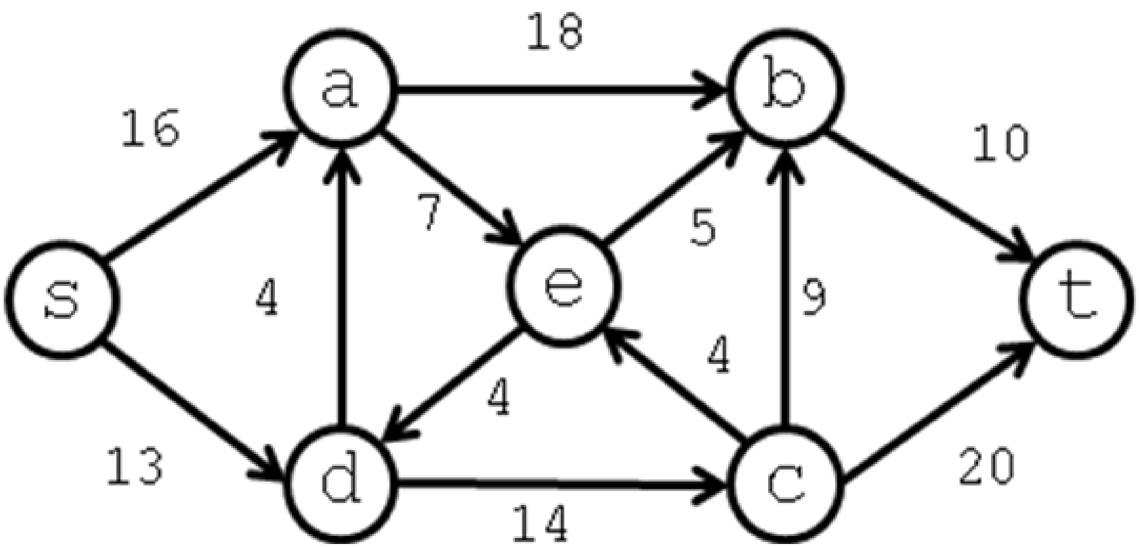
```
Ford-Fulkerson-Method(G, s, t):

 1. Inicialmente, fluxo  f = 0
 2. Enquanto existir um caminho aumentante P:
 3.      Incremente o fluxo f (usando P)
 4. Devolva f
```

Exercícios

- Simule o algoritmo de Ford-Fulkerson para calcular fluxo máximo:

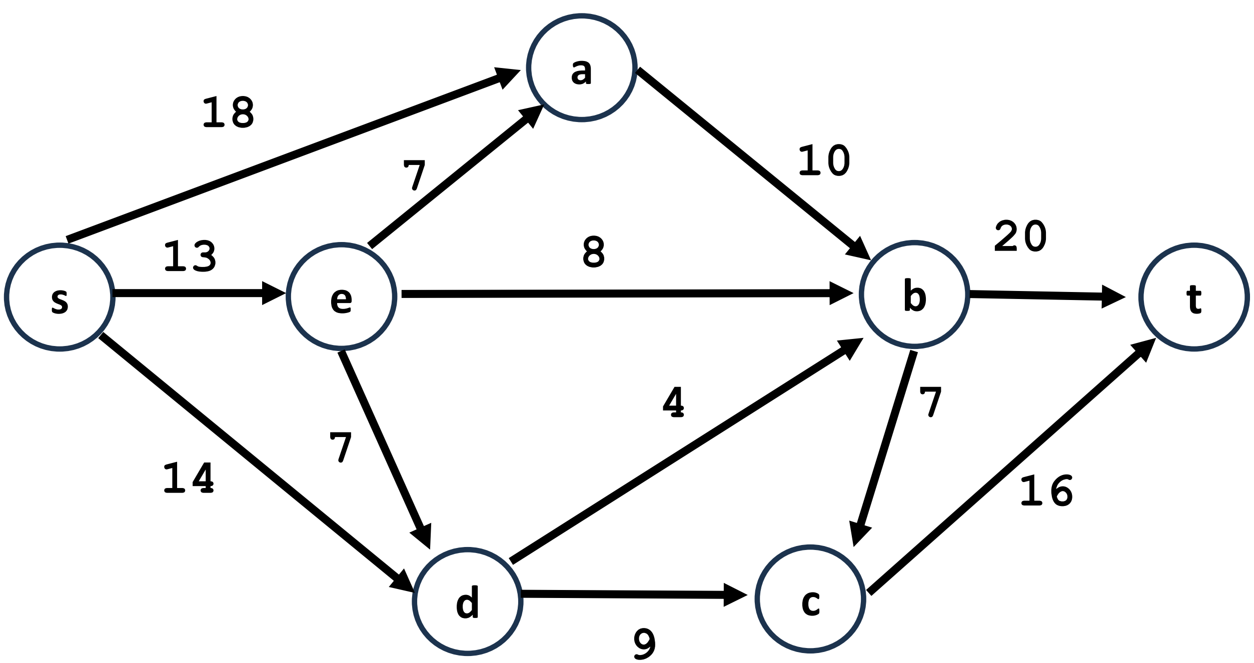
(a)



Exercícios

- Simule o algoritmo de Ford-Fulkerson para calcular fluxo máximo:

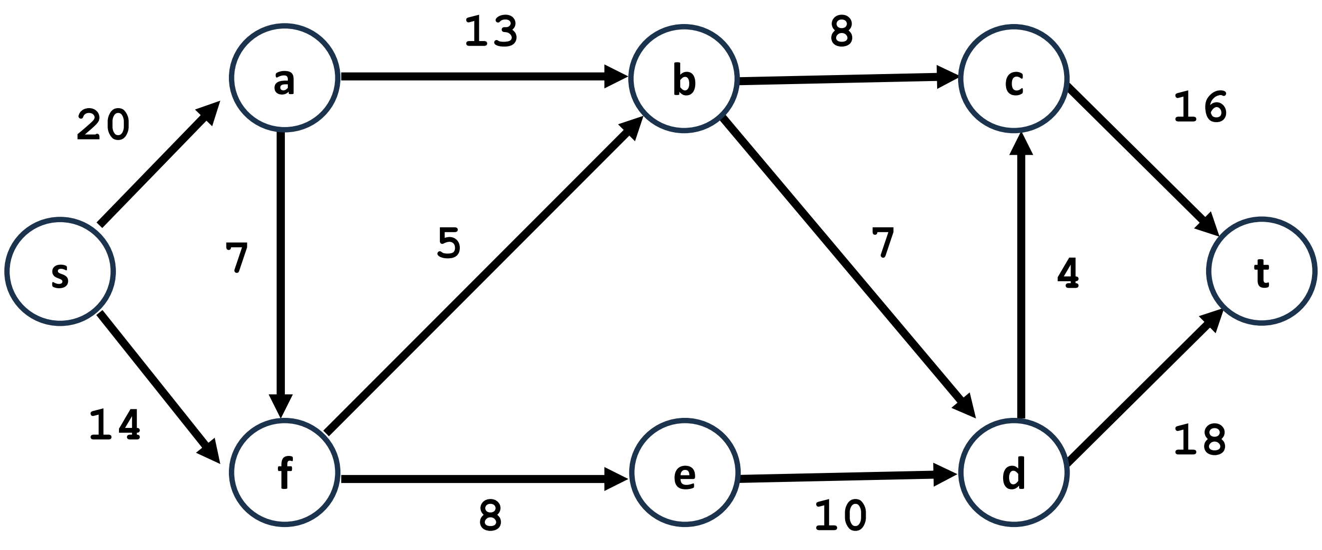
(b)



Exercícios

- Simule o algoritmo de Ford-Fulkerson para calcular fluxo máximo:

(c)



Exercício Programa

- 17-fluxoMaximo.py