Algoritmos em Grafos

Rômulo César Silva

Unioeste

Agosto de 2021

unioeste
Universidade Estadual do Oeste do Paraná

Sumário

Busca em Profundidade (Depth-First-Search) - DFS

Estruturas de dados:

- $cor[u]$: usado para indicar em que etapa se econtra a exploração do vértice u:
  - branco: não explorado
  - cinza: explorado, mas ainda sem explorar os adjacentes
  - preto: explorado e vértices adjacentes também explorados
- $\pi[v] = u$ indica que u precede v no grafo de busca
- $d[u]$: *timestamp* de descoberta de u
- $f[u]$: *timestamp* de término da exploração de u e vértices adjacentes a u
- $Adj[u]$: vértices adjacentes a u

Busca em Profundidade (Depth-First-Search) - DFS

```
 1: procedure DFS(G)
 2:     for each u ∈ V[G] do
 3:         cor[u] ← branco
 4:         π[u] ← NIL
 5:     end for
 6:     timestamp ← 0                          ▷ variável global
 7:     for each u ∈ V[G] do
 8:         if cor[u] = branco then
 9:             DFS_visit(u)
10:         end if
11:     end for
12: end procedure
```

Busca em Profundidade (Depth-First-Search) - DFS

```
 1: procedure DFS_VISIT(u)
 2:     cor[u] ← cinza
 3:     timestamp ← timestamp + 1
 4:     d[u] ← timpestamp
 5:     for each v ∈ Adj[u] do
 6:         if cor[v] = branco then
 7:             π[v] ← u
 8:             DFS_visit(v)                ▷ explora o vértice adjacente
 9:         end if
10:     end for
11:     cor[u] ← preto                      ▷ terminou adjacências de u
12:     timestamp ← timestamp + 1
13:     f[u] ← timestamp
14: end procedure
```

Busca em Profundidade (Depth-First-Search) - DFS

Complexidade:

- procedimento DFS:
  - linhas 02 a 05:$\Theta(V)$
  - linhas 07 a 11, excluindo a chamada de DFS_visit: $\Theta(V)$
- procedimento DFS_visit:
  - executado exatamente uma vez para cada vértice
  - linhas 05 a 10: $\Theta(E)$, pois corresponde às adjacências de cada vértice (número de arestas)
- procedimento DFS incluindo chamada DFS_visit: $\Theta(V + E)$

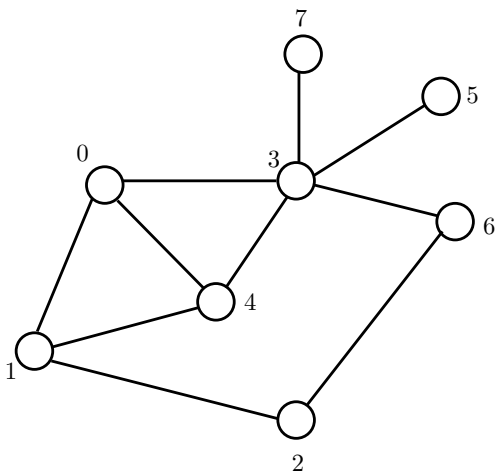
Busca em Profundidade (Depth-First-Search) - DFS

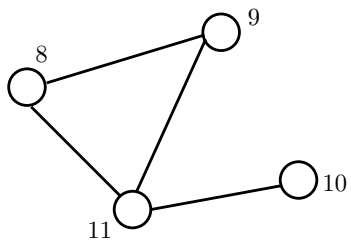
Observações:

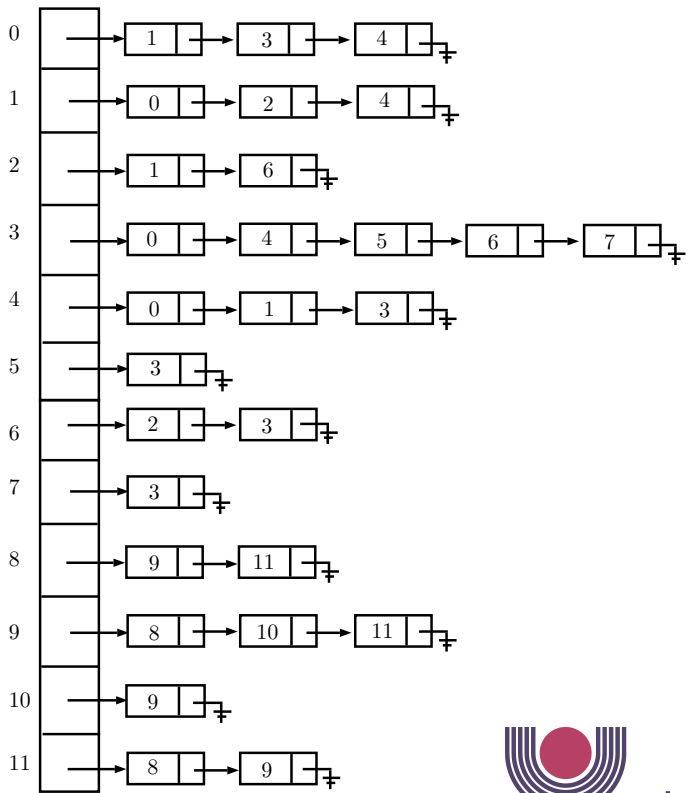
- caso o grafo seja conexo, as arestas escolhidas fazem parte da árvore de busca em profundidade
- caso o grafo seja desconexo, as arestas escolhidas fazem parte da floresta de busca em profundidade: nesse caso DFS_visit é chamada mais de uma vez

Busca em Profundidade (Depth-First-Search) - DFS

Considere o grafo abaixo tal que seja usada a representação de listas de adjacência em que os vértices estão inseridos nas listas em ordem crescente, conforme a figura abaixo.

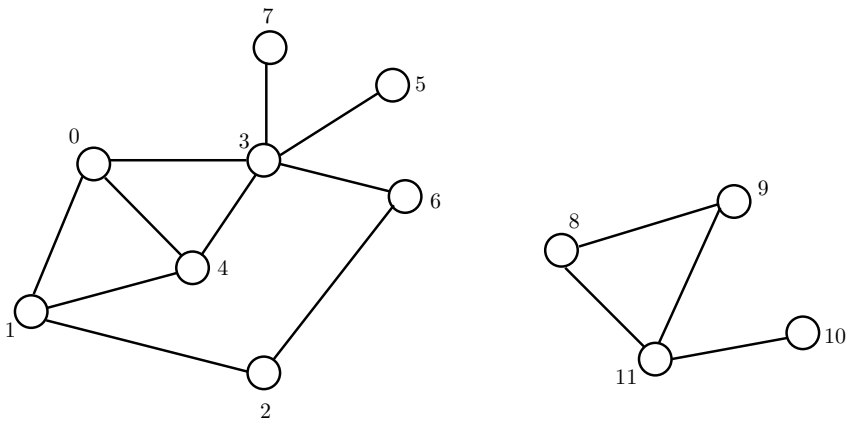






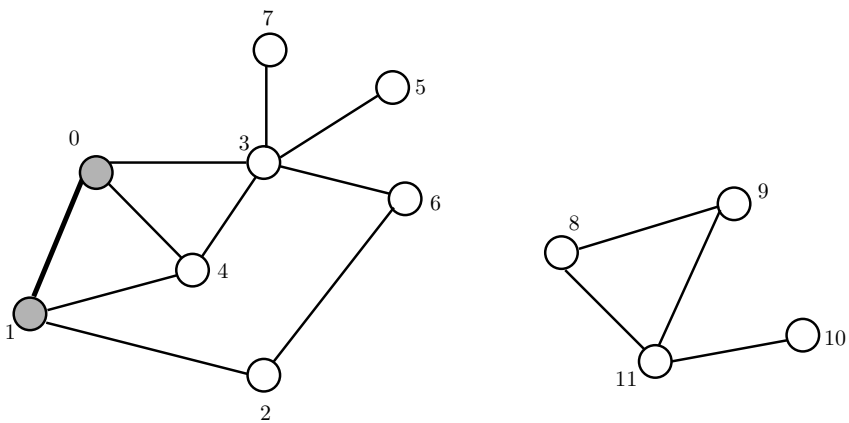
Busca em Profundidade (Depth-First-Search) - DFS

Após a execução das linhas 02-06 de DFS, teremos os seguintes valores nas variáveis:



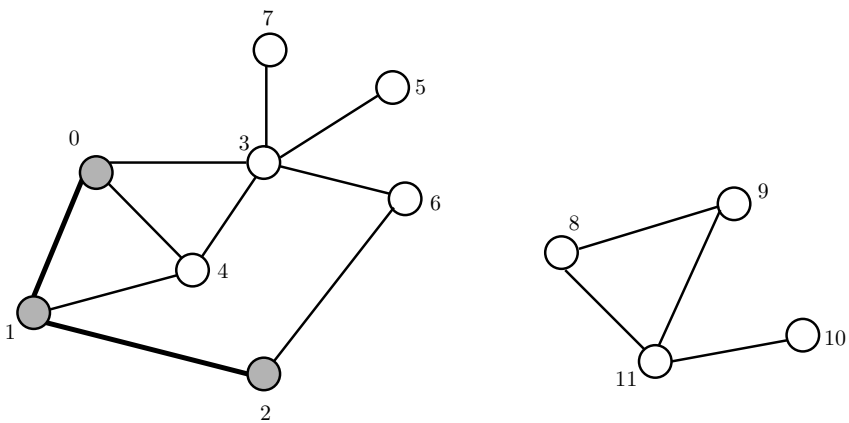
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	B	NIL		
1	B	NIL		
2	B	NIL		
3	B	NIL		
4	B	NIL		
5	B	NIL		
6	B	NIL		
7	B	NIL		
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



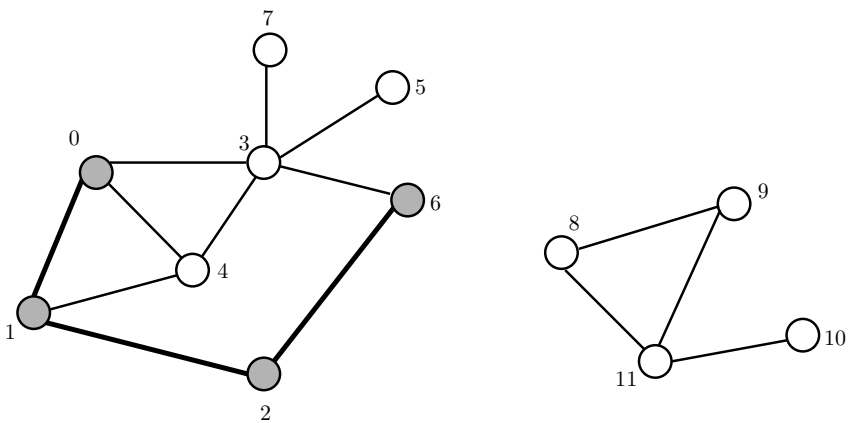
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	B	NIL		
3	B	NIL		
4	B	NIL		
5	B	NIL		
6	B	NIL		
7	B	NIL		
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



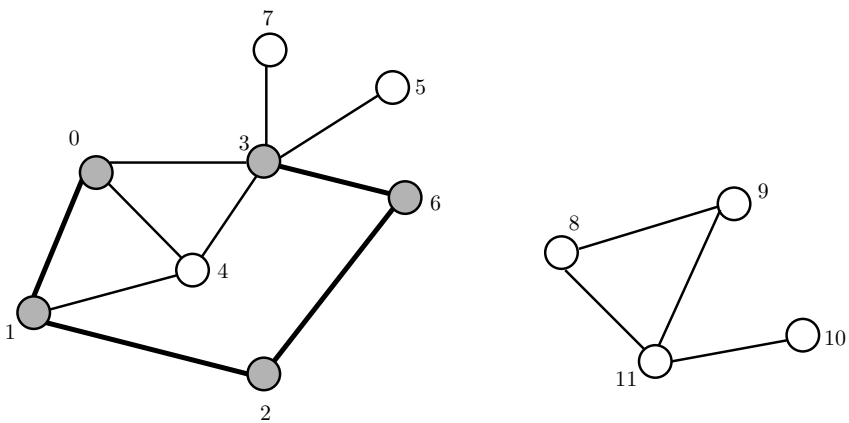
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	B	NIL		
4	B	NIL		
5	B	NIL		
6	B	NIL		
7	B	NIL		
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



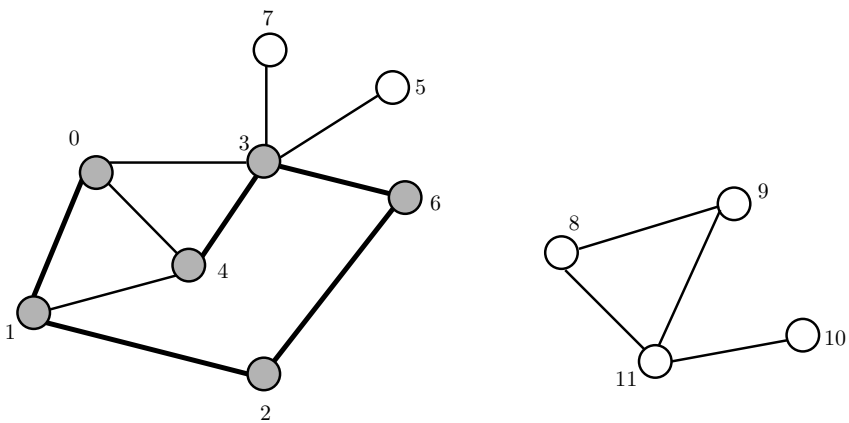
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	B	NIL		
4	B	NIL		
5	B	NIL		
6	C	2	4	
7	B	NIL		
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



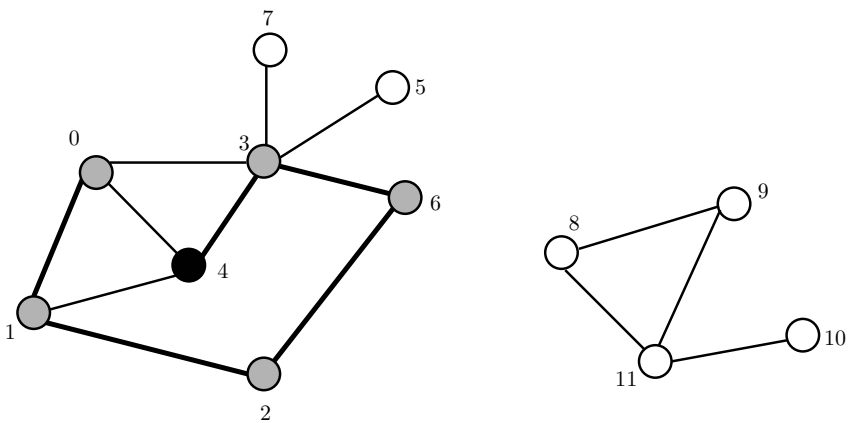
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	C	6	5	
4	B	NIL		
5	B	NIL		
6	C	2	4	
7	B	NIL		
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



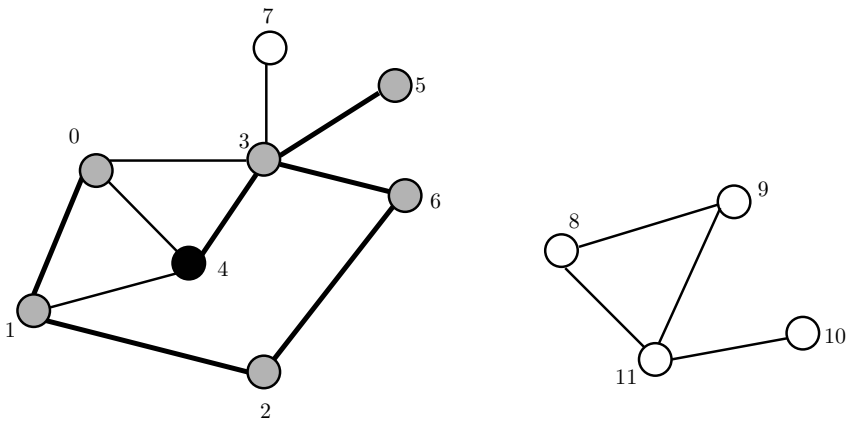
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	C	6	5	
4	C	3	6	
5	B	NIL		
6	C	2	4	
7	B	NIL		
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



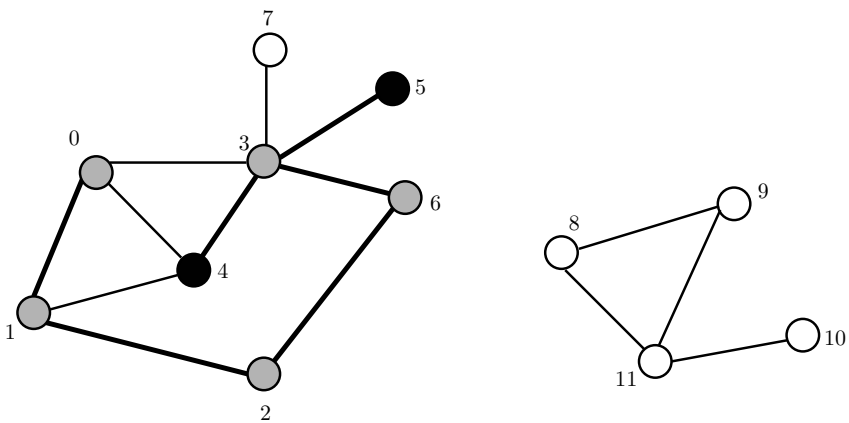
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	C	6	5	
4	P	3	6	7
5	B	NIL		
6	C	2	4	
7	B	NIL		
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



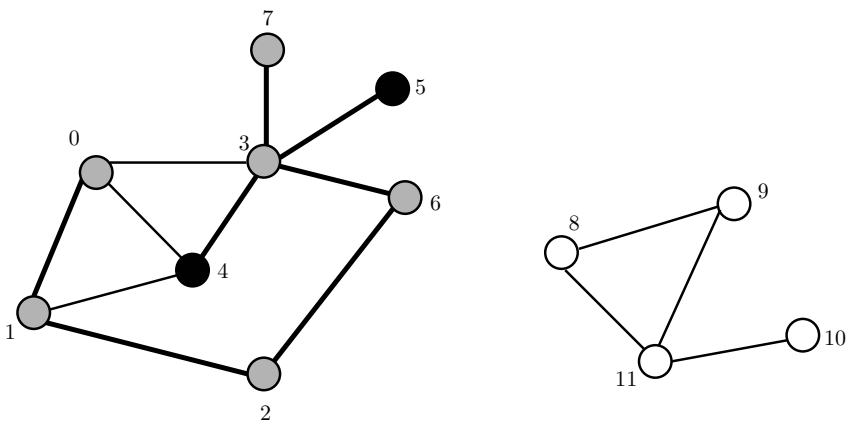
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	C	6	5	
4	P	3	6	7
5	C	3	8	
6	C	2	4	
7	B	NIL		
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



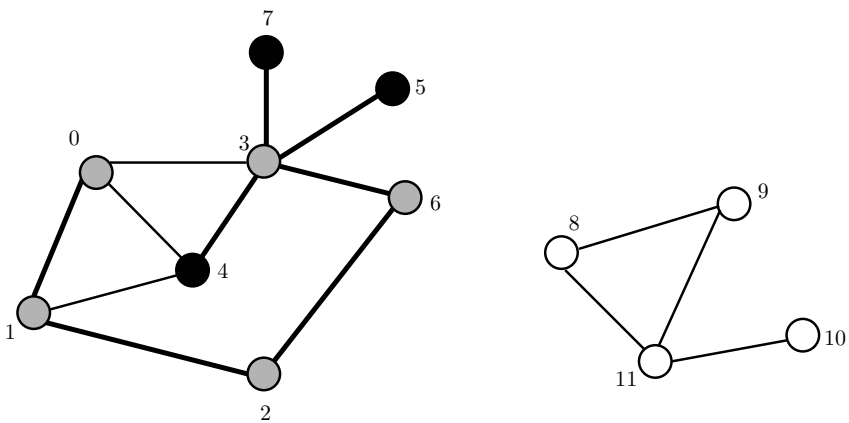
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	C	6	5	
4	P	3	6	7
5	P	3	8	9
6	C	2	4	
7	B	NIL		
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



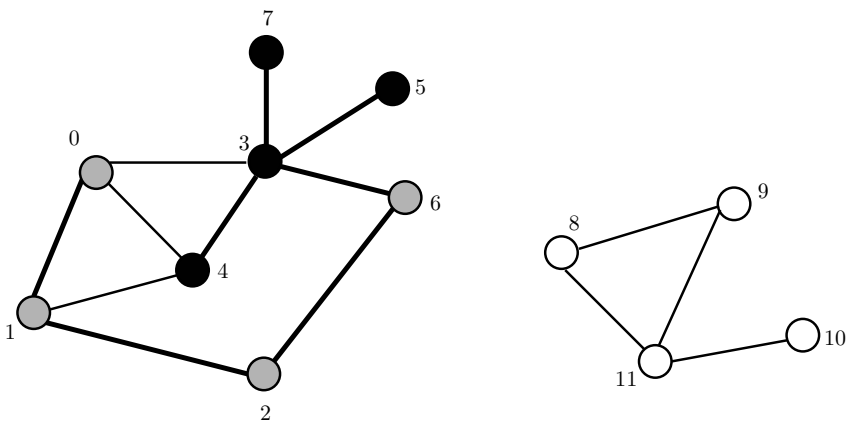
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	C	6	5	
4	P	3	6	7
5	P	3	8	9
6	C	2	4	
7	C	3	10	
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



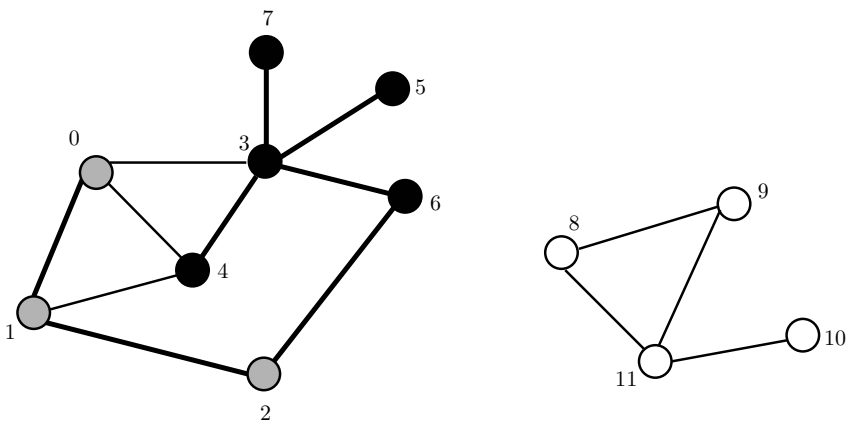
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	C	6	5	
4	P	3	6	7
5	P	3	8	9
6	C	2	4	
7	P	3	10	11
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



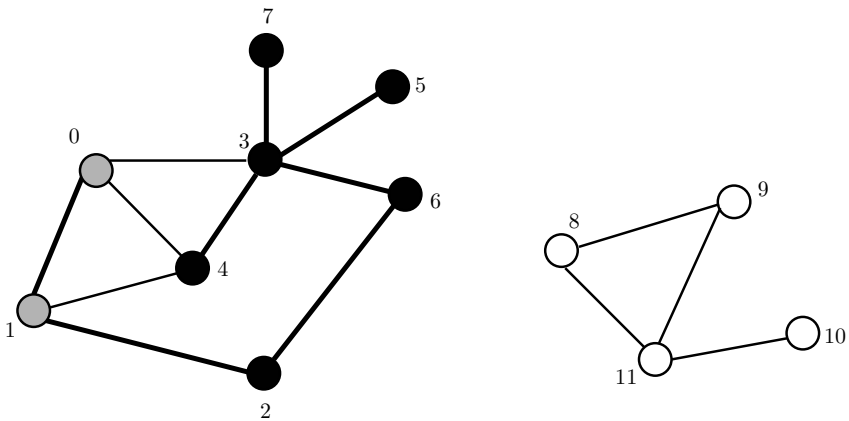
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	C	2	4	
7	P	3	10	11
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



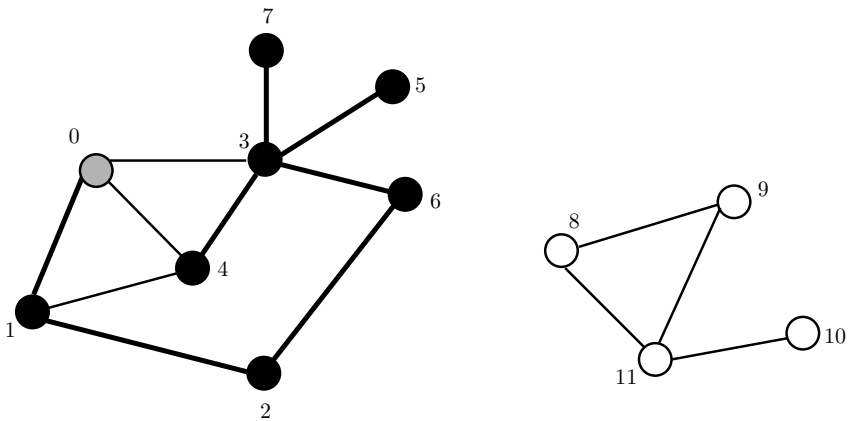
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	C	1	3	
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



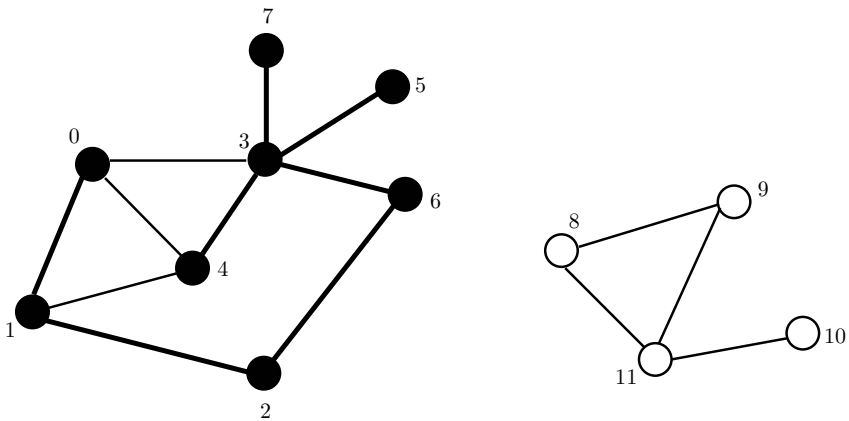
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	C	0	2	
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



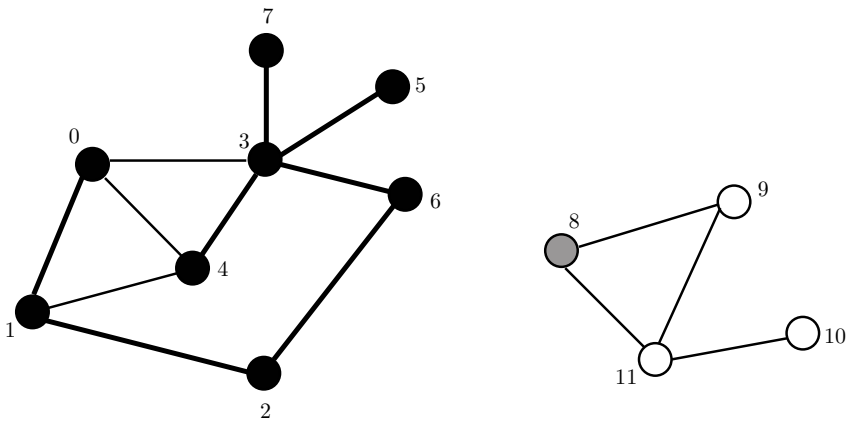
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	C	NIL	1	
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



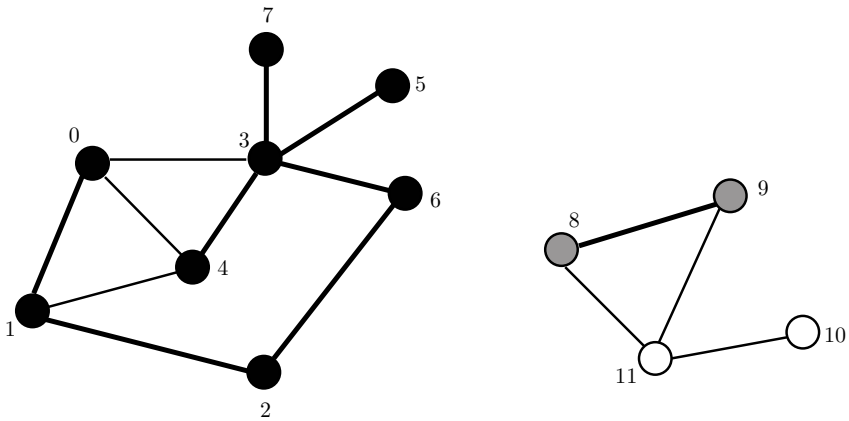
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	P	NIL	1	16
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	B	NIL		
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



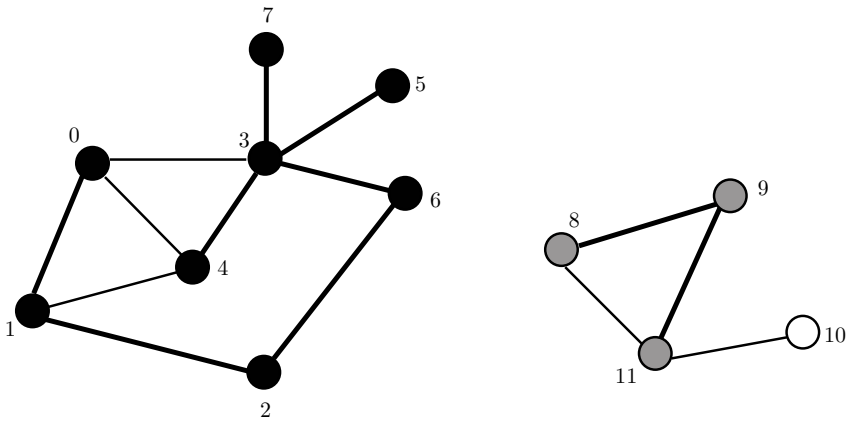
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	P	NIL	1	16
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	C	NIL	17	
9	B	NIL		
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



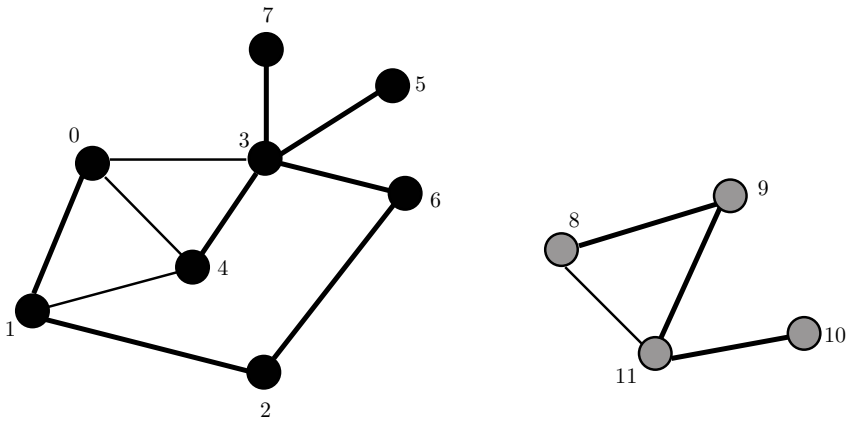
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	P	NIL	1	16
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	C	NIL	17	
9	C	8	18	
10	B	NIL		
11	B	NIL		

Busca em Profundidade (Depth-First-Search) - DFS



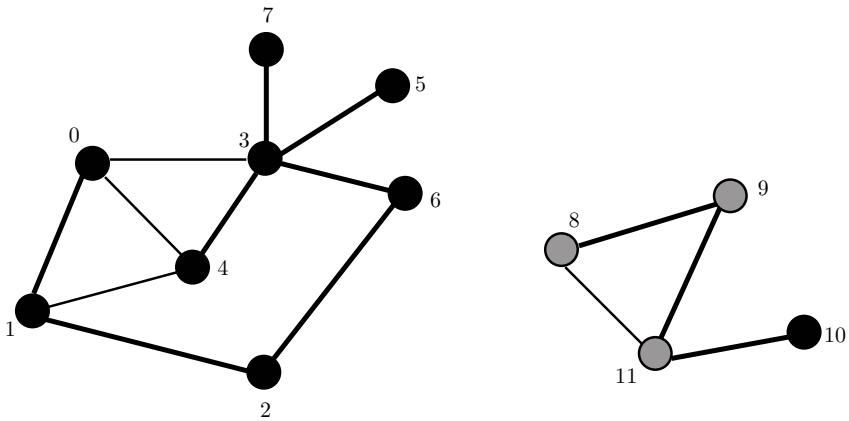
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	P	NIL	1	16
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	C	NIL	17	
9	C	8	18	
10	B	NIL		
11	C	9	19	

Busca em Profundidade (Depth-First-Search) - DFS



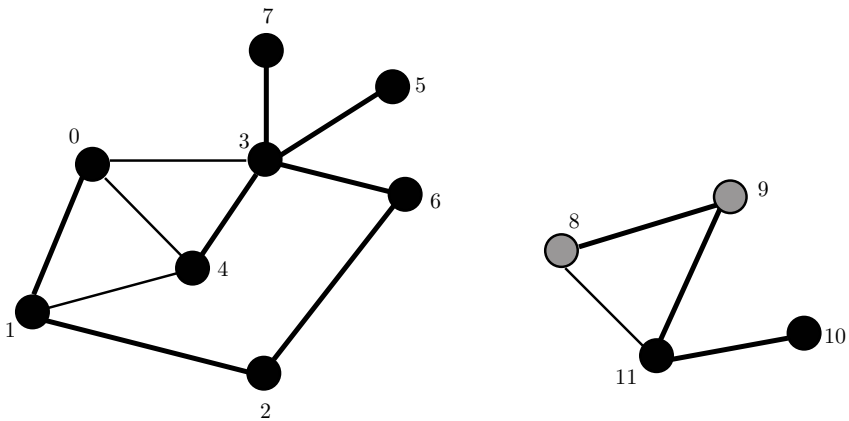
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	P	NIL	1	16
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	C	NIL	17	
9	C	8	18	
10	C	11	20	
11	C	9	19	

Busca em Profundidade (Depth-First-Search) - DFS



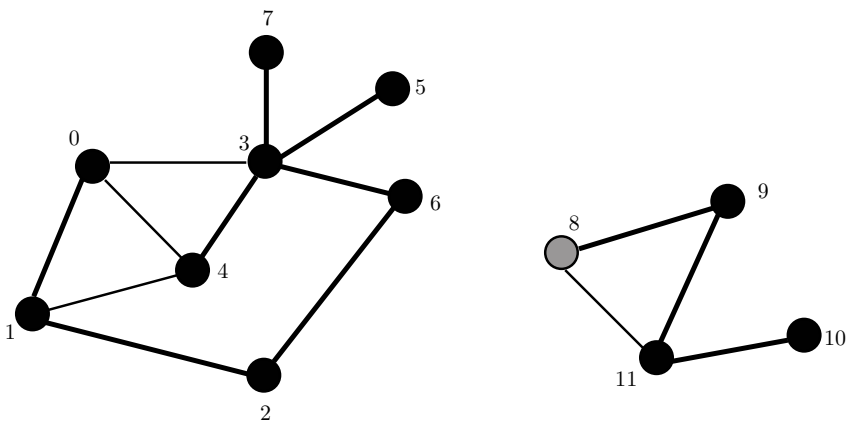
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	P	NIL	1	16
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	C	NIL	17	
9	C	8	18	
10	P	11	20	21
11	C	9	19	

Busca em Profundidade (Depth-First-Search) - DFS



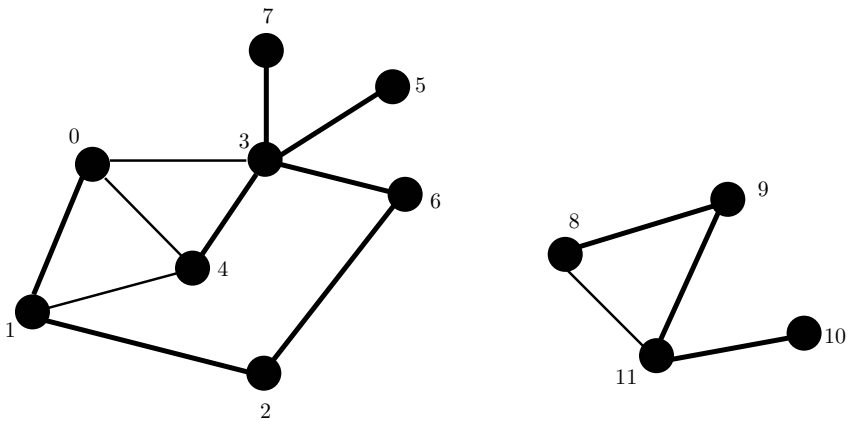
u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	P	NIL	1	16
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	C	NIL	17	
9	C	8	18	
10	P	11	20	21
11	P	9	19	22

Busca em Profundidade (Depth-First-Search) - DFS



u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	P	NIL	1	16
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	C	NIL	17	
9	P	8	18	23
10	P	11	20	21
11	P	9	19	22

Busca em Profundidade (Depth-First-Search) - DFS



u	$cor[u]$	$\pi[u]$	$d[u]$	$f[u]$
0	P	NIL	1	16
1	P	0	2	15
2	P	1	3	14
3	P	6	5	12
4	P	3	6	7
5	P	3	8	9
6	P	2	4	13
7	P	3	10	11
8	P	NIL	17	24
9	P	8	18	23
10	P	11	20	21
11	P	9	19	22

Busca em Profundidade (Depth-First-Search) - DFS

Propriedades:
Para quaisquer 2 vértices u e v, alguma das seguinte situações ocorre:

- os intervalos $[d[u], f[u]]$ e $[d[v], f[v]]$ são inteiramente disjuntos
- o intervalo $[d[u], f[u]]$ está inteiramente dentro do intervalo $[d[v], f[v]]$. Nesse caso, u é descendente de v na árvore de busca em profundidade
- o intervalo $[d[v], f[v]]$ está inteiramente dentro do intervalo $[d[u], f[u]]$. Nesse caso, v é descendente de u na árvore de busca em profundidade

Busca em Profundidade (Depth-First-Search) - DFS

A Busca em Profundidade pode ser usada para classificar as arestas em:

- **tree edge** (aresta da árvore): aresta escolhida durante o processo de busca. (u, v) é aresta da árvore de busca se v foi descoberto pela exploração da aresta (u, v)
- **back edge** (aresta de retorno): são aquelas arestas (u, v) tal que v é ancestral de u na árvore de busca
- **forward edge**: são aquelas arestas (u, v) tal que v é descendente de u na árvore de busca
- **cross edge**: todas as outras arestas

Busca em Profundidade (Depth-First-Search) - DFS

(u, v) pode ser classificada de acordo com a cor do vértice v, atingido quando a aresta é explorada pela primeira vez:

- branco: aresta da árvore
- cinza: aresta de retorno
- preto: *forward* ou *cross*

Busca em Profundidade (Depth-First-Search) - DFS

Observações:

- em grafo não-orientado, cada aresta é aresta da ávore ou aresta de retorno.
- (u, v):
  - é aresta da árvore ou *forward* se e somente $d[u] < d[v] < f[v] < f[u]$
  - é aresta de retorno se e somente se $d[v] < d[u] < f[u] < f[v]$
  - é *cross* se e somente se $d[v] < f[v] < d[u] < f[u]$

Busca em Largura (Breadth-First-Search) - BFS

Estruturas de dados:

- $cor[u]$: usado para indicar em que etapa se econtra a exploração do vértice u:
  - branco: não explorado
  - cinza: explorado, mas ainda sem explorar os adjacentes
  - preto: explorado e vértices adjacentes também explorados
- $\pi[v] = u$ indica que u precede v no grafo de busca
- $d[u]$: distância do vértice inicial s até o vértice u
- Q: fila que gerencia os nós de cor cinza
- $Adj[u]$: vértices adjacentes a u

Busca em Largura (Breadth-First-Search) - BFS

```
1:  procedure BFS_VISIT(G, s)
2:      ▷ s é vertice inicial da busca
3:      for each u ∈ V[G] − {s} do
4:          cor[u] ← branco
5:          d[u] ← ∞
6:          π[u] ← NIL
7:      end for
8:      cor[s] ← cinza
9:      π[s] ← NIL
10:     d[s] ← 0
11:     Q ← {s}
12:     while Q ≠ ∅ do
13:         u ← head[Q]                          ▷ Obtém a cabeça da fila
14:         for each v ∈ Adj[u] do
15:             if cor[v] = branco then
16:                 cor[v] ← cinza
17:                 d[v] ← d[u] + 1
18:                 π[v] ← u
19:                 enqueue(Q,v)                 ▷ insere v na fila
20:             end if
21:         end for
22:         dequeue(Q)                           ▷ remove a cabeça da fila
23:         cor[u] ← preto
24:     end while
25: end procedure
```

Busca em Largura (Breadth-First-Search) - BFS

Complexidade:

- linhas 03 a 07:$\Theta(V)$
- as operações de fila (*enqueue* e *dequeue*) tomam tempo $O(1)$ cada. Como cada vértice é enfileirado/desenfileirado uma única vez, o total é $O(V)$
- como a lista de adjacência de cada vértice é varrida uma única vez, o total é $O(E)$
- tempo total de BFS: $O(V + E)$

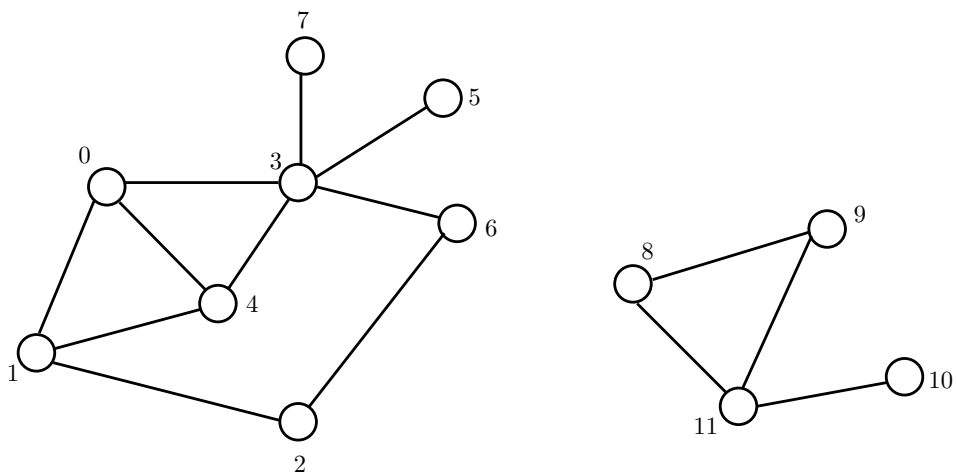
Busca em Largura (Breadth-First-Search) - BFS

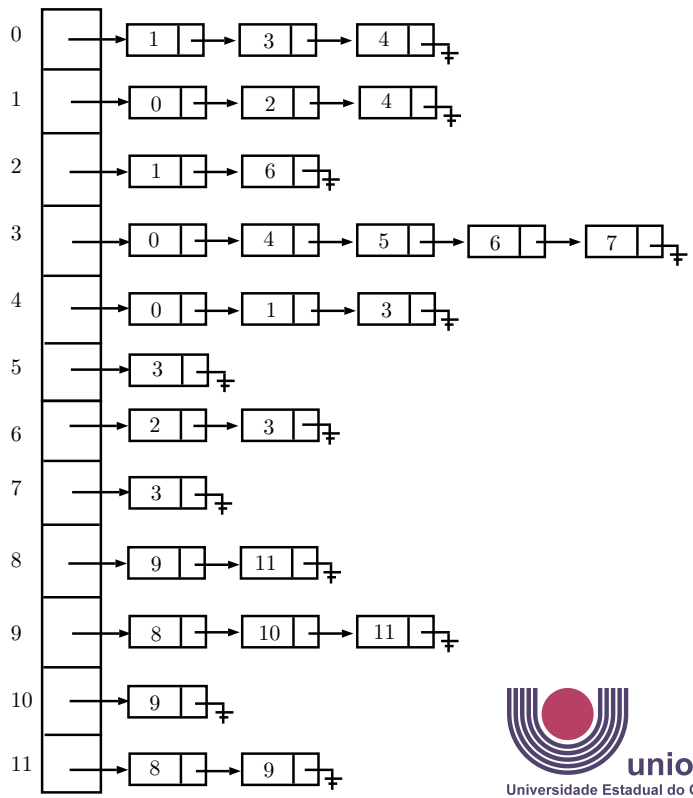
Observações:

- o procedimento BFS constrói uma árvore de busca em largura
- as arestas u, v) escolhidas fazem parte do menor caminho enre o vértice inicial s e os demais vértices
- se o grafo for desconexo, os vértices v que não estão na mesma componente conexa do vértice inicial s terão $d[v] = \infty$

Busca em Largura (Breadth-First-Search) - BFS

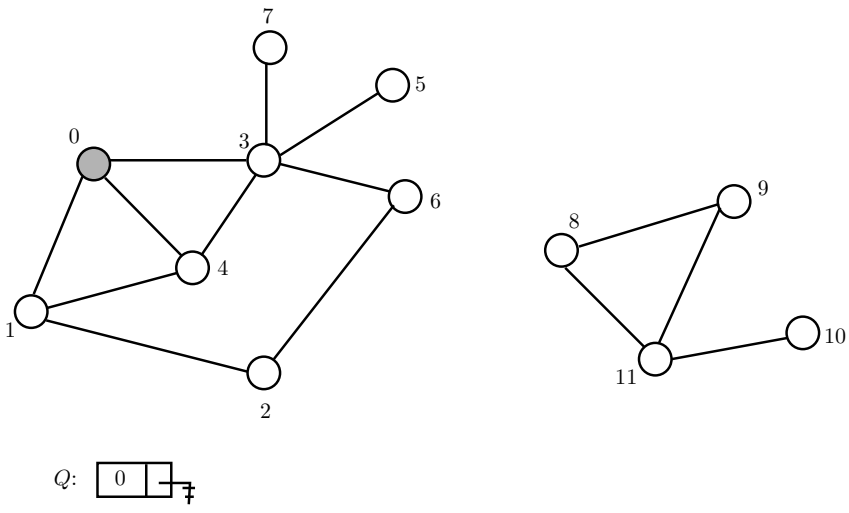
Considere o grafo abaixo tal que seja usada a representação de listas de adjacência em que os vértices estão inseridos nas listas em ordem crescente, conforme a figura abaixo, e que a busca se inicie pelo vértice 0 (zero).





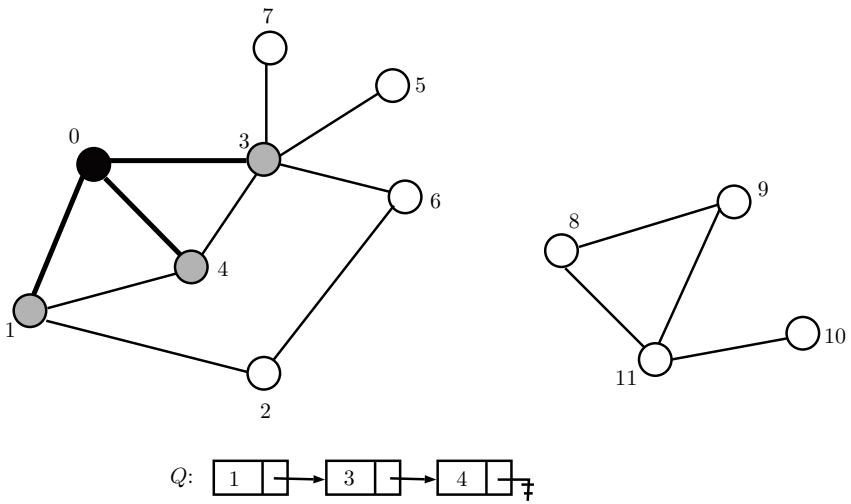
Busca em Largura (Breadth-First-Search) - BFS

Após a execução de BFS até a linha 11:



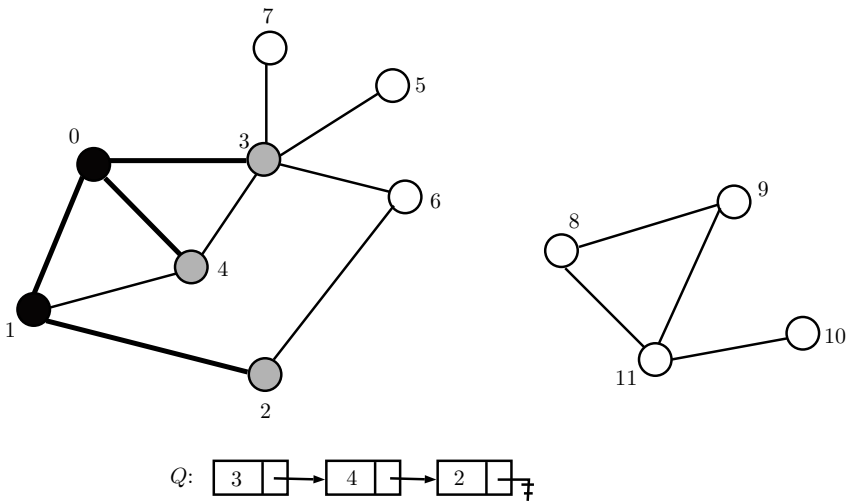
u	$cor[u]$	$\pi[u]$	$d[u]$
0	C	NIL	0
1	B	NIL	∞
2	B	NIL	∞
3	B	NIL	∞
4	B	NIL	∞
5	B	NIL	∞
6	B	NIL	∞
7	B	NIL	∞
8	B	NIL	∞
9	B	NIL	∞
10	B	NIL	∞
11	B	NIL	∞

Busca em Largura (Breadth-First-Search) - BFS



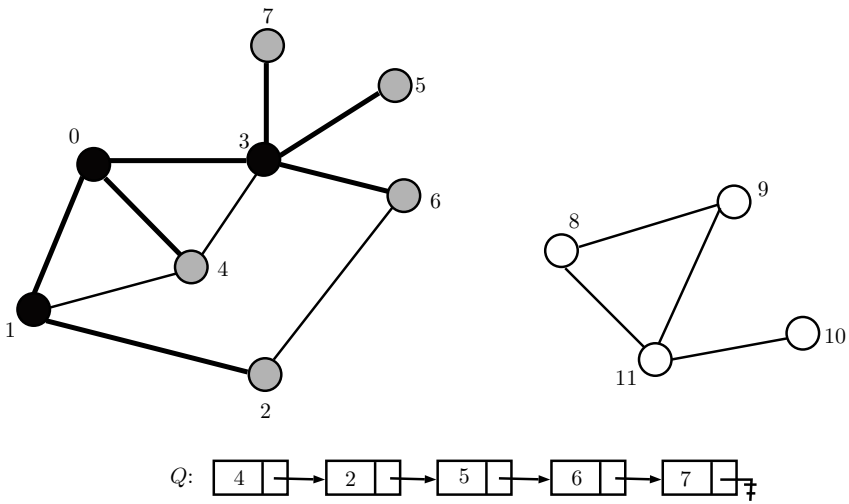
u	$cor[u]$	$\pi[u]$	$d[u]$
0	P	NIL	0
1	C	0	1
2	B	NIL	∞
3	C	0	1
4	C	0	1
5	B	NIL	∞
6	B	NIL	∞
7	B	NIL	∞
8	B	NIL	∞
9	B	NIL	∞
10	B	NIL	∞
11	B	NIL	∞

Busca em Largura (Breadth-First-Search) - BFS



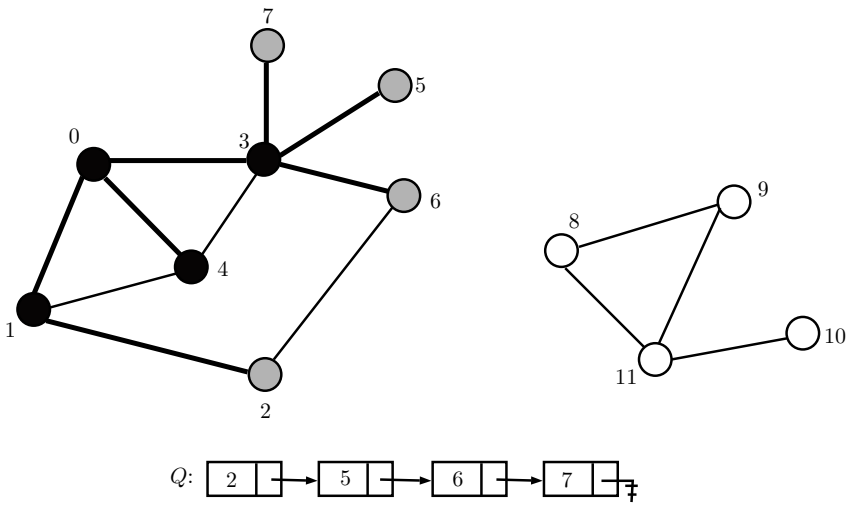
u	$cor[u]$	$\pi[u]$	$d[u]$
0	P	NIL	0
1	P	0	1
2	C	1	2
3	C	0	1
4	C	0	1
5	B	NIL	∞
6	B	NIL	∞
7	B	NIL	∞
8	B	NIL	∞
9	B	NIL	∞
10	B	NIL	∞
11	B	NIL	∞

Busca em Largura (Breadth-First-Search) - BFS



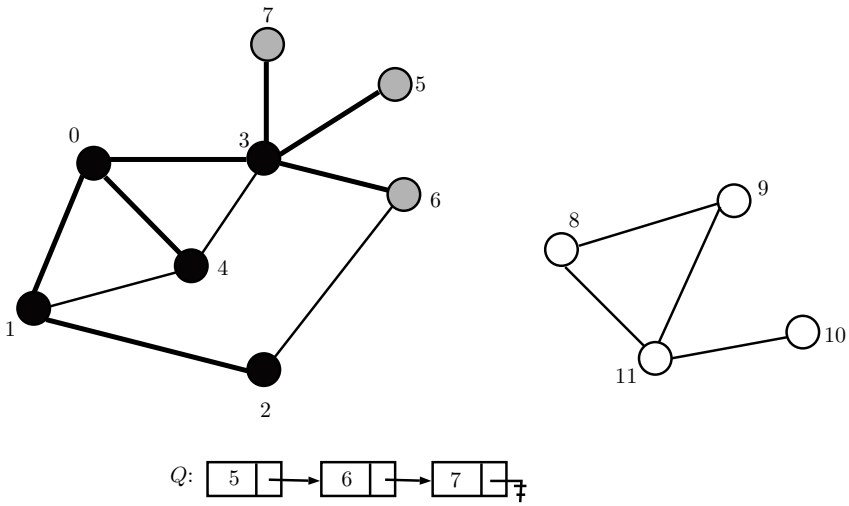
u	$cor[u]$	$\pi[u]$	$d[u]$
0	P	NIL	0
1	P	0	1
2	C	1	2
3	P	0	1
4	C	0	1
5	C	3	2
6	C	3	2
7	C	3	2
8	B	NIL	∞
9	B	NIL	∞
10	B	NIL	∞
11	B	NIL	∞

Busca em Largura (Breadth-First-Search) - BFS



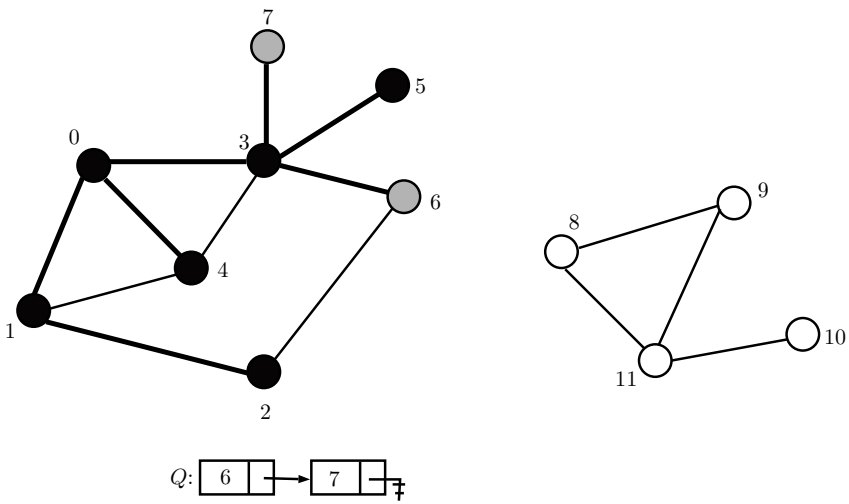
u	$cor[u]$	$\pi[u]$	$d[u]$
0	P	NIL	0
1	P	0	1
2	C	1	2
3	P	0	1
4	P	0	1
5	C	3	2
6	C	3	2
7	C	3	2
8	B	NIL	∞
9	B	NIL	∞
10	B	NIL	∞
11	B	NIL	∞

Busca em Largura (Breadth-First-Search) - BFS



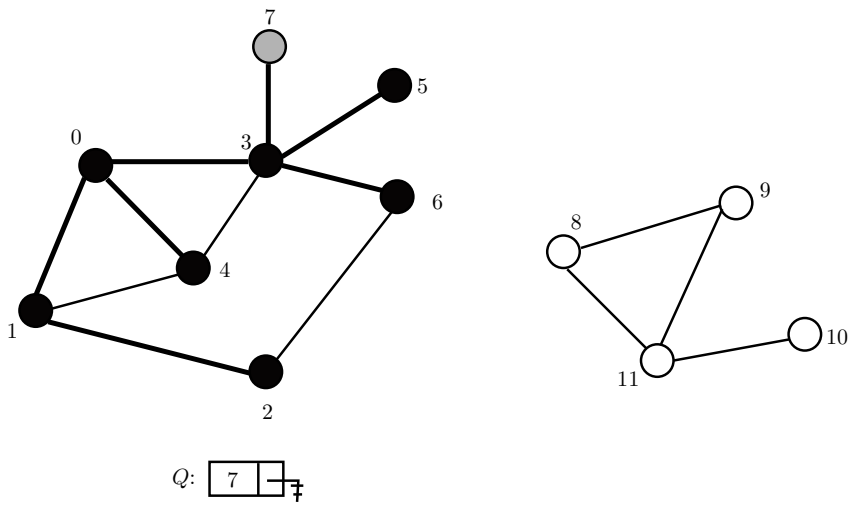
u	$cor[u]$	$\pi[u]$	$d[u]$
0	P	NIL	0
1	P	0	1
2	P	1	2
3	P	0	1
4	P	0	1
5	C	3	2
6	C	3	2
7	C	3	2
8	B	NIL	∞
9	B	NIL	∞
10	B	NIL	∞
11	B	NIL	∞

Busca em Largura (Breadth-First-Search) - BFS



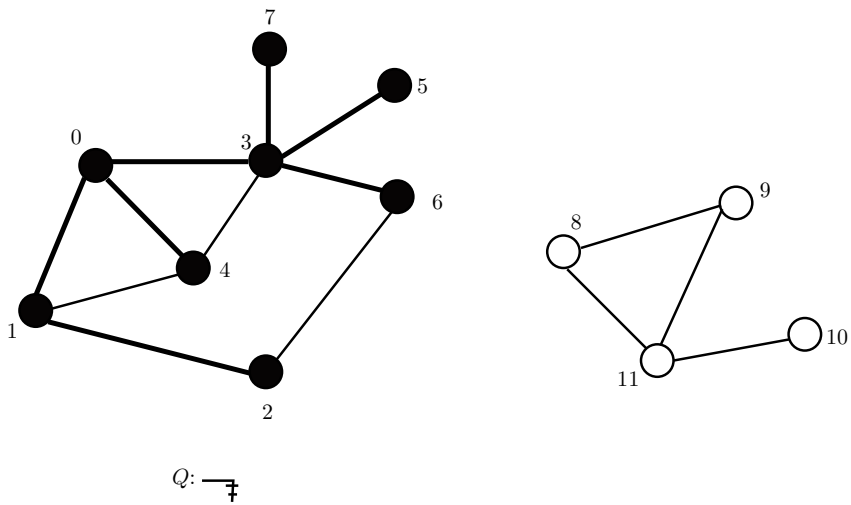
u	$cor[u]$	$\pi[u]$	$d[u]$
0	P	NIL	0
1	P	0	1
2	P	1	2
3	P	0	1
4	P	0	1
5	P	3	2
6	C	3	2
7	C	3	2
8	B	NIL	∞
9	B	NIL	∞
10	B	NIL	∞
11	B	NIL	∞

Busca em Largura (Breadth-First-Search) - BFS



u	$cor[u]$	$\pi[u]$	$d[u]$
0	P	NIL	0
1	P	0	1
2	P	1	2
3	P	0	1
4	P	0	1
5	P	3	2
6	P	3	2
7	C	3	2
8	B	NIL	∞
9	B	NIL	∞
10	B	NIL	∞
11	B	NIL	∞

Busca em Largura (Breadth-First-Search) - BFS



u	$cor[u]$	$\pi[u]$	$d[u]$
0	P	NIL	0
1	P	0	1
2	P	1	2
3	P	0	1
4	P	0	1
5	P	3	2
6	P	3	2
7	P	3	2
8	B	NIL	∞
9	B	NIL	∞
10	B	NIL	∞
11	B	NIL	∞

Busca em Largura (Breadth-First-Search) - BFS

Assumindo que o procedimento BFS já tenha sido executado, o procedimento abaixo imprime os vértices que compõem o menor caminho entre o vértice inicial s e o vértice v:

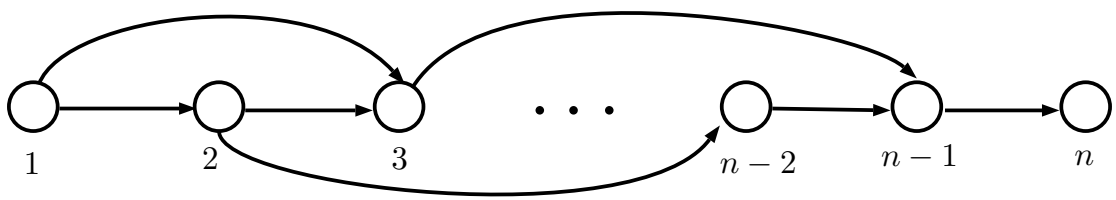
```
1:  procedure ImprimeCaminho(G, s, v)
2:      if v = s then
3:          print(s)
4:      else
5:          if π[v] = NIL then
6:              print não existe caminho de s para v
7:          else
8:              ImprimeCaminho(G,s,π[v])
9:              print(v)
10:         end if
11:     end if
12: end procedure
```

Ordenação Topológica

Ordenação Topológica

Dado um grafo orientado acíclico $G = (V, E)$ com n vértices, uma **ordenação topológica** em G consiste em rotular os vértices de 1 até n tal que, se o vértice v possui número k, então todos os vértices que podem ser atingidos a partir de v por um caminho orientado possuem numeros maior que k.

Esquema:



Ordenação Topológica

Observação:

- se $G = (V, E)$ é grafo orientado acíclico, então existe pelo menos um vértice cujo grau de entrada é zero (**fonte**)
- se $G = (V, E)$ é grafo orientado acíclico, então existe pelo menos um vértice cujo grau de saída é zero (**ralo**)

Ideia usando busca em profundidade: após calcular $f[v]$, inserir o vértice v na cabeça de uma lista.

Ordenação Topológica

```
1: function OrdenacaoTopologica(G)
2:     ▷ entrada: grafo G
3:     ▷ saída: lista L com os nós na ordem topológica
4:     L ← ∅                                          ▷ lista de vértices
5:     DFS(G)        ▷ para calcular f[v] e logo após inserir v na cabeça de L
6:     return L
7: end function
```

Complexidade:

- inserção na lista: $O(1)$
- DFS modificada: $O(V + E)$
- ordenação topológica (total): $O(V + E)$

Ordenação Topológica - 2^{o} algoritmo

Estruturas de dados:

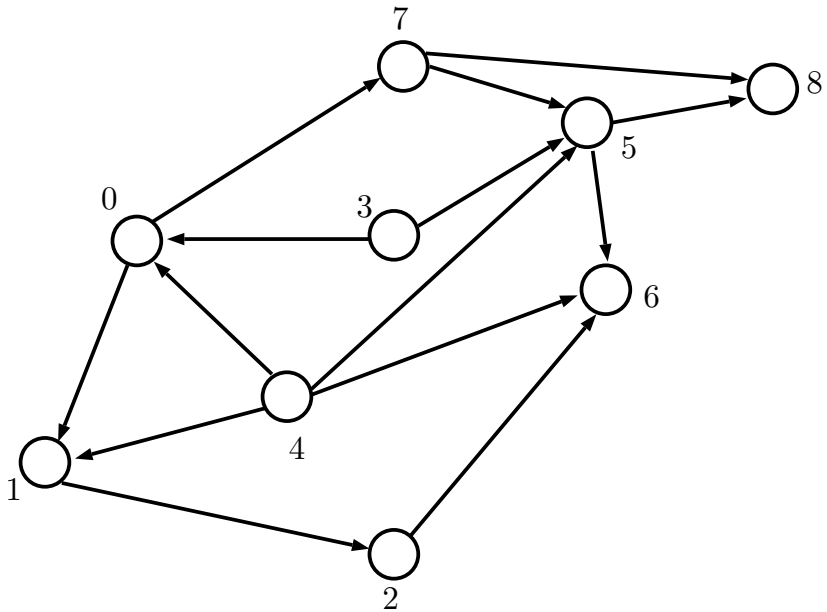
- g_{in}: vetor com grau de entrada de todos os vértices
- Q: fila
- *ordem*: variável global para controlar a ordem de vértices
- $v.ordem$: ordem do vértice v na ordenação topológica

Ordenação Topológica - 2º algoritmo

```
 1: procedure OrdTopologica(G)
 2:     ▷ entrada: grafo G
 3:     ordem ← 1
 4:     Q ← ∅                                              ▷ fila vazia
 5:     calcular g_in ∀v ∈ V[G]
 6:     for i ← 1, ..., n do
 7:         if g_in[i] = 0 then
 8:             enqueue(Q, v_i)
 9:         end if
10:     end for
11:     while Q ≠ ∅ do
12:         v ← dequeue(Q)
13:         v.ordem ← ordem
14:         ordem ← ordem + 1
15:         for each (v, w) ∈ E do                         ▷ varre cada aresta partindo de v
16:             g_in[w] ← g_in[w] − 1
17:             if g_in[w] = 0 then
18:                 enqueue(Q, w)
19:             end if
20:         end for
21:     end while
22: end procedure
```

Ordenação Topológica - 2º algoritmo

Considere o grafo abaixo, já exibindo os valores das variáveis após a execução das linhas 03-10:

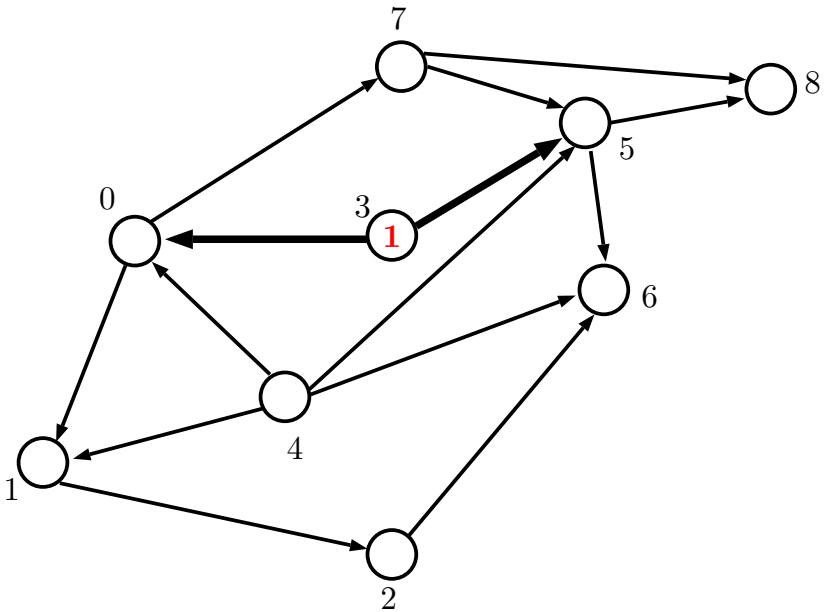


g_{in} :

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

Q: 3 → 4 → ∄

Ordenação Topológica - 2º algoritmo

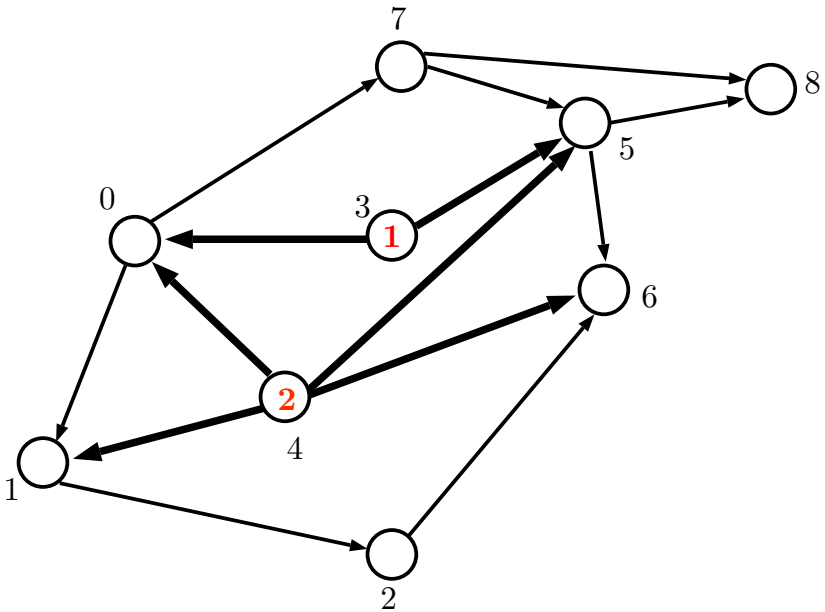


g_{in} :

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

Q: | 4 |

Ordenação Topológica - 2º algoritmo

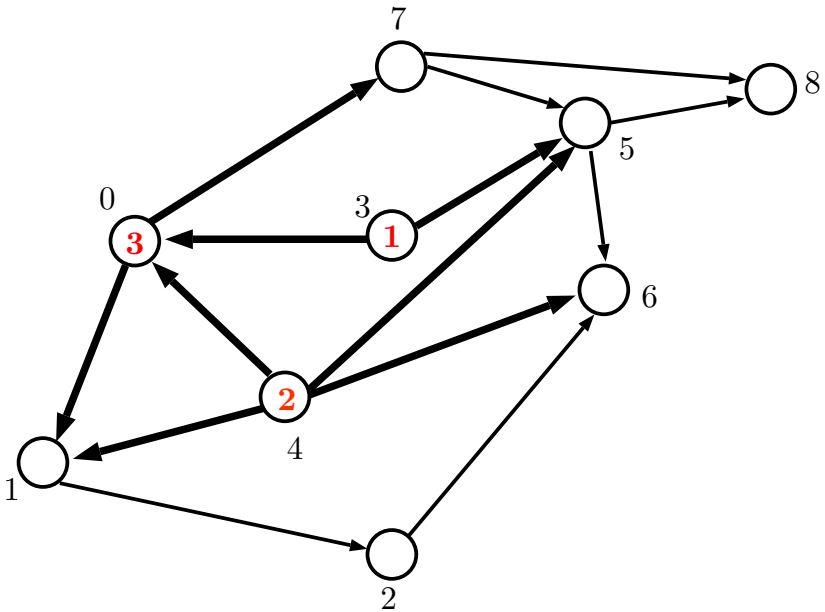


g_{in} :

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

Q: | 0 |

Ordenação Topológica - 2º algoritmo

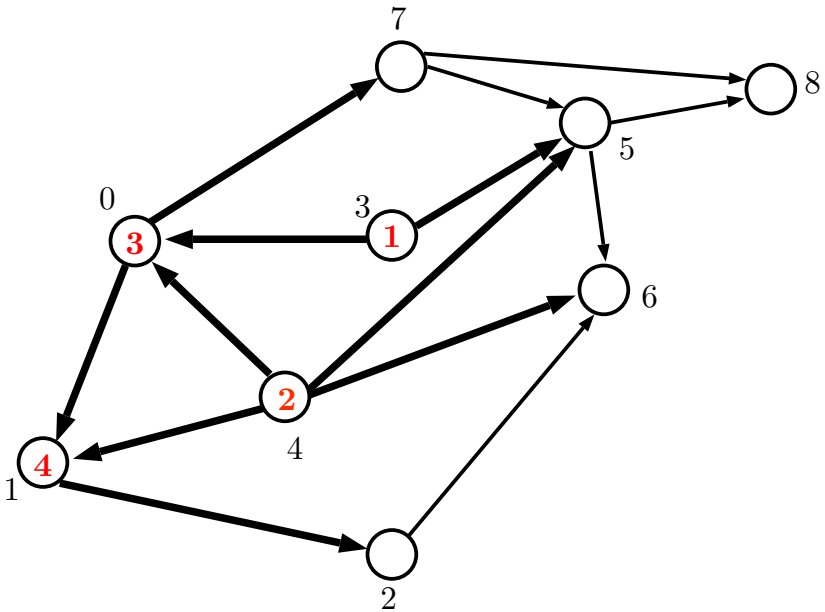


g_{in} :

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

Q: 1 → 7 ⫫

Ordenação Topológica - 2º algoritmo

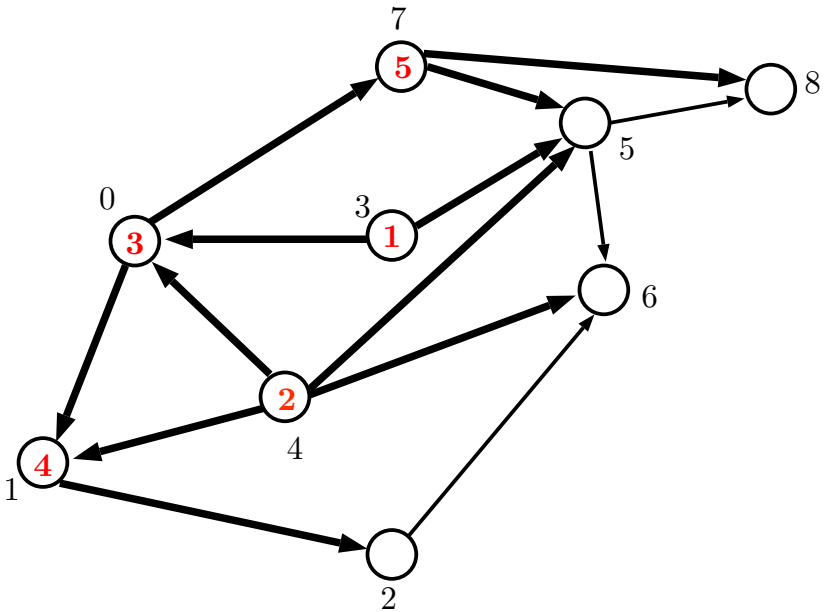


g_{in} :

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

Q: 7 → 2 ⏚

Ordenação Topológica - 2º algoritmo

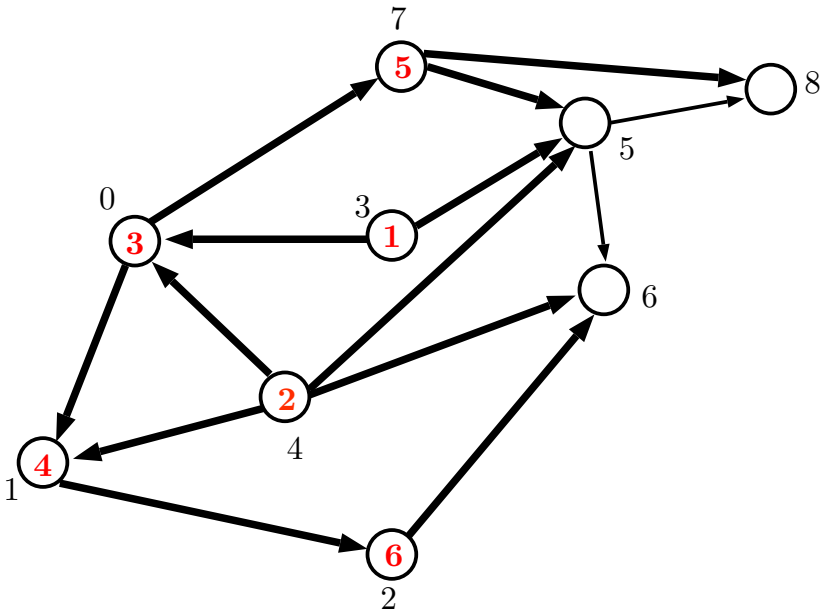


g_{in} :

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

Q: 2 → 5 ⫫

Ordenação Topológica - 2º algoritmo

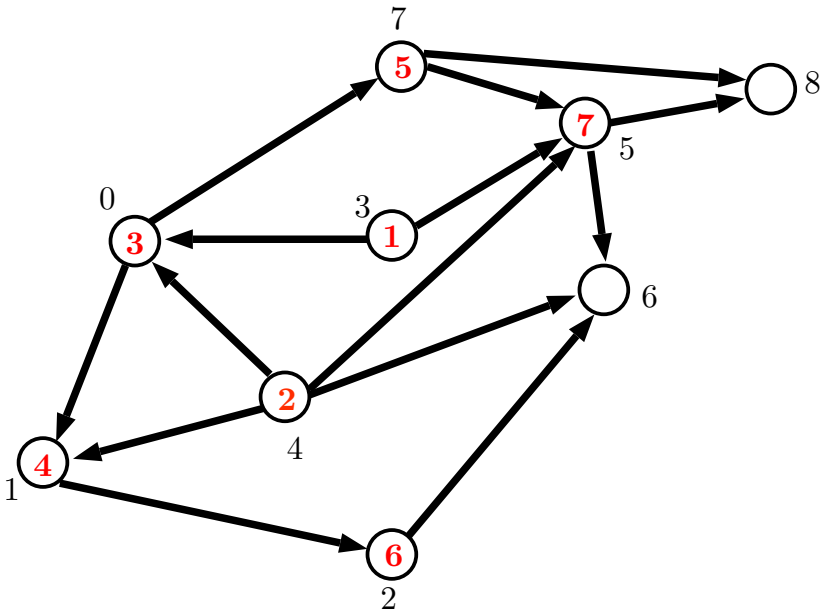


g_{in} :

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

Q: 5 ⏚

Ordenação Topológica - 2º algoritmo

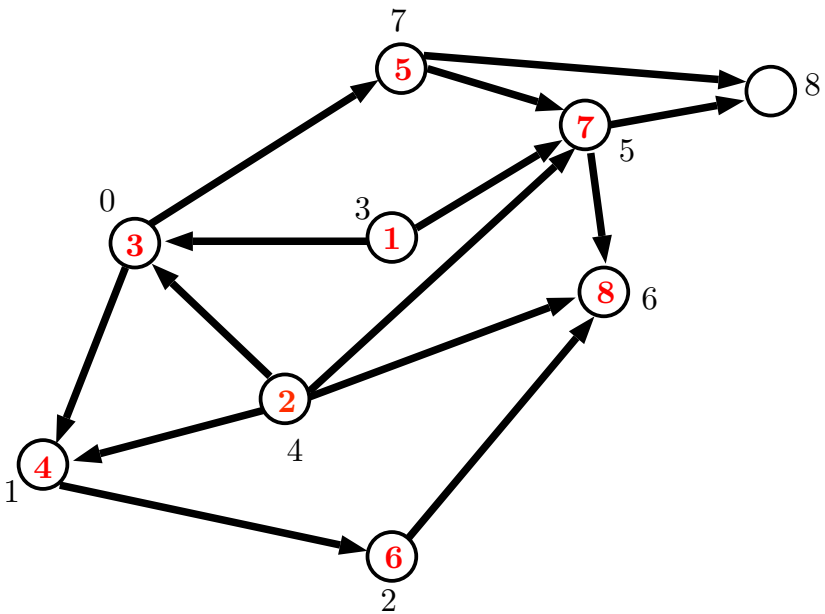


g_{in} :

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

Q: 6 → 8

Ordenação Topológica - 2º algoritmo

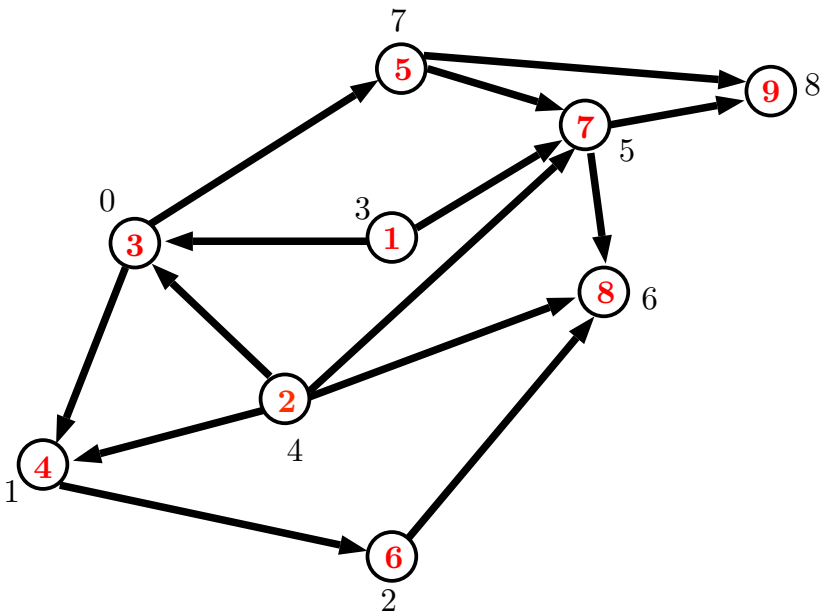


g_{in} :

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

Q: 8

Ordenação Topológica - 2º algoritmo

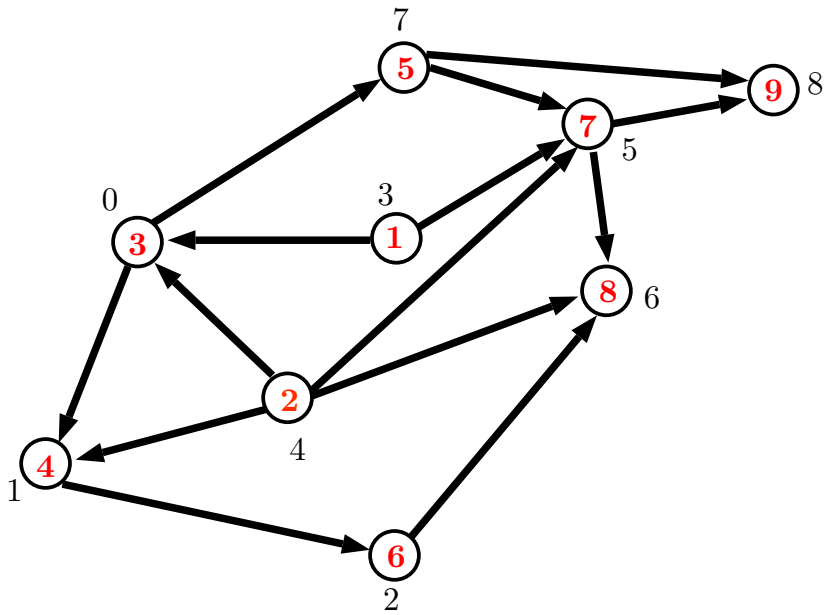


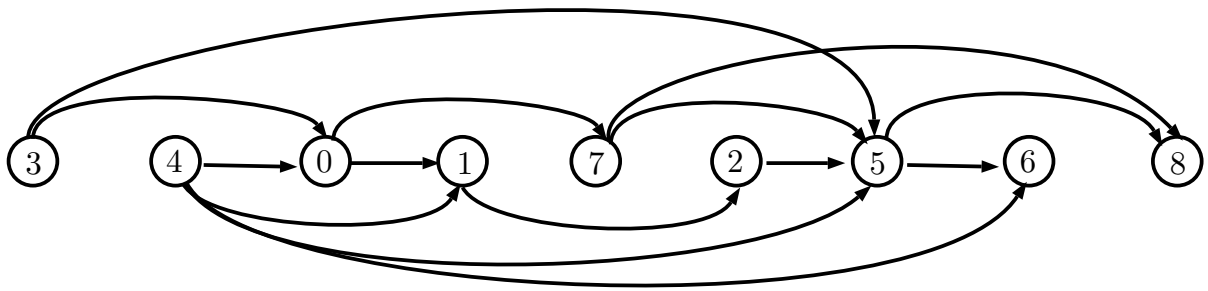
g_{in} :

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

Q: ⊣

Ordenação Topológica - 2º algoritmo





Ordenação Topológica - 2^{o} algoritmo

- como todo grafo orientado acíclico tem pelo menos um vértice cujo grau de entrada é zero, este vértice deve ser o primeiro na ordem topológica
- a fila controla a ordem de exploração dos vértices
- ao se explorar a aresta (u, v) decrementa-se o grau de entrada do vértice v

Complexidade: $O(V + E)$

Componentes Conexas

- somente para grafos não-orientados
- usar DFS modificada: criar variável para contar quantas árvores são obtidas pela busca

Componentes Conexas

```
1:  procedure ComponentesConexas(G)
2:      ▷ entrada: grafo G
3:      nroComp ← 0                                   ▷ variável global
4:      DFS_cc(G)
5:  end procedure
```

```
1:  procedure DFS_cc(G)
2:      ▷ DFS modificada
3:      for each u ∈ V[G] do
4:          cor[u] ← branco
5:      end for
6:      for each u ∈ G[G] do
7:          if cor[u] = branco then
8:              DFS_visit(u)
9:              nroComp ← nroComp + 1          ▷ uma componente foi explorada
10:         end if
11:     end for
12: end procedure
```

Componentes Fortemente Conexas

Grafo Transposto

Seja $G = (V, E)$ grafo orientado. O grafo transposto de G, denotado por G^T, é dado por $G^T = (V, E^T)$, em que $E^T = \{(u, v) | (v, u) \in E\}$.

- o grafo transporto G^T tem as mesmas componentes fortemente conexas do grafo G
- fazer uma modificação na DFS para encontrar as componentes fortemente conexas

Componentes Fortemente Conexas

1: **procedure** $\mathrm{CFC}(G)$
2: usar DFS(G) para calcular $f[u]$ $\forall u \in V$
3: calcular G^T
4: usar DFS(G^T), mas no *loop* principal considerar os vértices em ordem decrescente de $f[u]$
5: imprimir os vertices de cada árvore DFS da linha 04 como uma componente fortemente conexa separada
6: **end procedure**

Complexidade:

- DFS(G): $\Theta(V + E)$
- DFS(G^T): $\Theta(V + E)$
- tempo total: $\Theta(V + E)$

Problemas de Menor Caminho

Variantes:

- único destino: de todos os vértices para um único vértice
- entre vértice-origem e vértice-destino
- única origem para todos os vértices
- entre todos os pares de vértices

Problemas de Menor Caminho

Problema de Menor Caminho

Seja $G = (V, E)$ grafo orientado com pesos (custos) associados às arestas através de função $w : E \rightarrow \mathbb{R}$

O peso (custo) do caminho $p =< v_0, v_1, ..., v_k >$ é dado por:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

O menor caminho entre u e v é dado por:

$$\delta(u, v) = \begin{cases} min\{\, w(p) : p \text{ é caminho de } u \text{ para } v\} \\ \infty \quad \text{se não existe caminho de } u \text{ para } v \end{cases}$$

Problemas de Menor Caminho

Observações:

- se há arestas de peso negativo, então se elas participam de um ciclo, o peso do ciclo não pode ser negativo, pois $\delta(u, v)$ seria $-\infty$
- **Propriedade básica do menor caminho**:
  - se $p =< v_1, v_2, ..., v_k >$ é o menor caminho de v_1 para v_k, então $\forall i, j$ com $1 \leq i \leq j \leq k$, o subcaminho $p_{ij} =< v_i, v_{i+1}, ..., v_j >$ é o menor caminho de v_i para v_j
  - resumindo: subcaminhos de menor caminho também são caminhos mínimos
  - problema apresenta o **princípio da subestrutura ótima** em problemas de otimização

Problemas de Menor Caminho

- Estruturas de dados:
  - $d[v]$: custo estimado do menor caminho até v
  - $\pi[v]$: predecessor de v no menor caminho
  - $w(u, v)$: peso da aresta (u, v)
- **relaxação**: método que repetidamente diminui o limite superior do peso do menor caminho, usado em vários algoritmos

Problemas de Menor Caminho

- **Relaxação:**

```
1: procedure RELAX(u, v)
2:     ▷ Relaxação
3:     if d[v] > d[u] + w(u, v) then          ▷ custo pode ser diminuído
4:         d[v] ← d[u] + w(u, v)              ▷ atualiza o custo
5:         π[v] ← u                           ▷ armazena o predecessor
6:     end if
7: end procedure
```

Problemas de Menor Caminho

- variante escolhida: único vértice inicial (origem)

```
1: procedure InicializaOrigem(G, s)
2:     ▷ s é vertice de origem
3:     for each v ∈ V[G] do
4:         d[v] ← ∞
5:         π[v] ← NIL
6:     end for
7:     d[s] ← 0                    ▷ distância da origem para ela mesma é zero
8: end procedure
```

Menor Caminho - Algoritmo de Dijkstra

- Pré-requisitos:
  - único vértice de origem
  - apenas arestas com pesos não-negativos (i.e, $w(u, v) \geq 0$)
- estruturas de dados:
  - Q: fila de prioridade usando $d[v]$
  - S: conjunto de vértices destino com menores caminhos já calculados

Menor Caminho - Algoritmo de Dijkstra

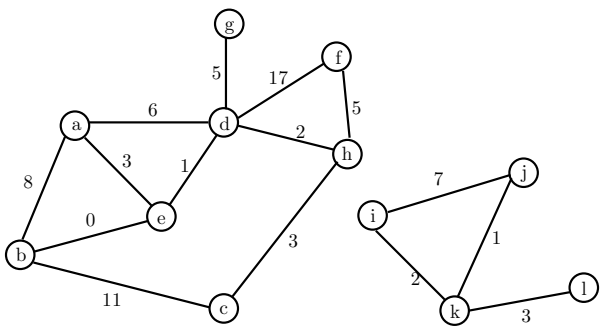
```
 1: procedure DIJKSTA(G, s)
 2:     ▷ s é vertice de origem
 3:     InicializaOrigem(G, s)
 4:     S ← ∅
 5:     Q ← V[G]                          ▷ fila de prioridade usando d[v]
 6:     while Q ≠ ∅ do
 7:         u ← extraiMinimo(Q)
 8:         S ← S ∪ {u}
 9:         for each v ∈ Adj[u] do
10:             Relax(u, v)
11:         end for
12:     end while
13: end procedure
```

Menor Caminho - Algoritmo de Dijkstra

- utiliza uma **estratégia gulosa**: sempre escolhe o vértice mais próximo (menor peso) em $V - S$ para ser incluído em S
- no fim do algoritmo: $d[u] = \delta(s, u)$
- complexidade:
  - $O(V^2)$: se a fila Q é implementada como um vetor
  - $O(E \lg V)$: se a fila Q é implementada como um *heap* binário
  - $O(V \lg V + E)$: se a fila Q é implementada como um *heap* de Fibonacci

Menor Caminho - Algoritmo de Dijkstra

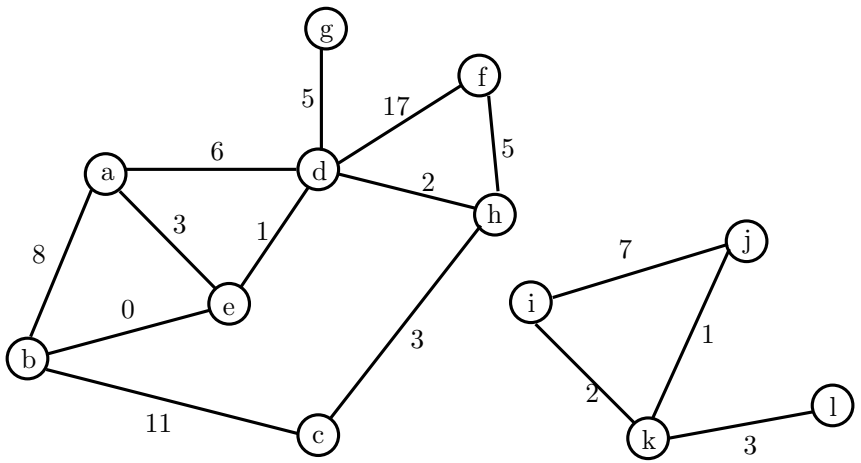
Considere o grafo abaixo tal que seja usada a representação de matriz adjacência:



$$A_G = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \\ k \\ l \end{array} \begin{array}{c} \begin{array}{cccccccccccc} a & b & c & d & e & f & g & h & i & j & k & l \end{array} \\ \begin{bmatrix} 0 & 8 & 0 & 6 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 1 & 17 & 5 & 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 17 & 0 & 0 & 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \end{bmatrix} \end{array}$$

Menor Caminho - Algoritmo de Dijkstra

Considerando o vértice *a* como origem, após a execução das linhas 03-05, antes de entrar no *loop*, as estruturas de dados serão:

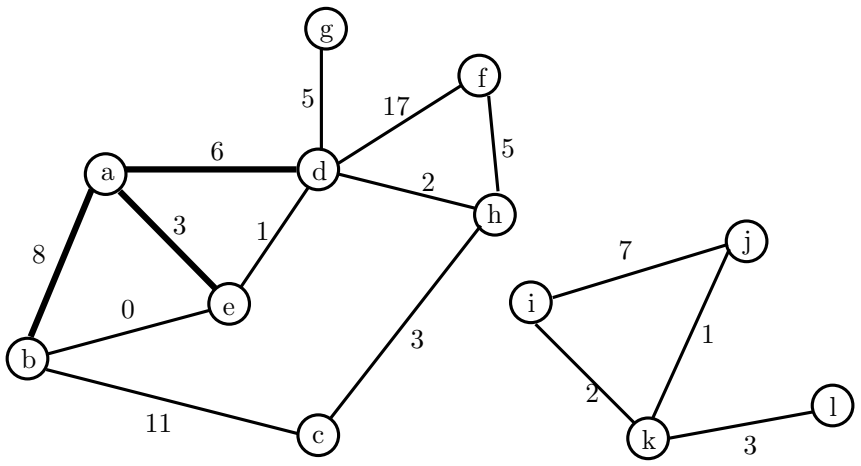


Q: | a | b | c | d | e | f | g | h | i | j | k | l |

S: $\emptyset$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	NIL	∞
c	NIL	∞
d	NIL	∞
e	NIL	∞
f	NIL	∞
g	NIL	∞
h	NIL	∞
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

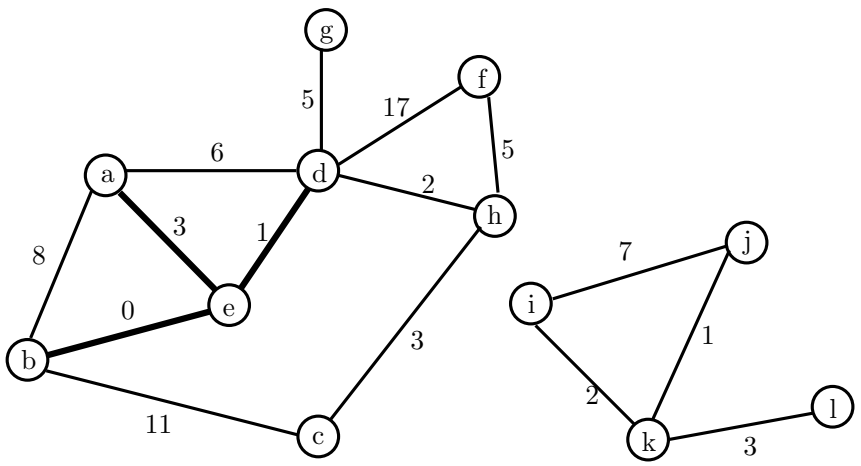


Q: | e | d | b | c | f | g | h | i | j | k | l |

$S : \{a\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	a	8
c	NIL	∞
d	a	6
e	a	3
f	NIL	∞
g	NIL	∞
h	NIL	∞
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

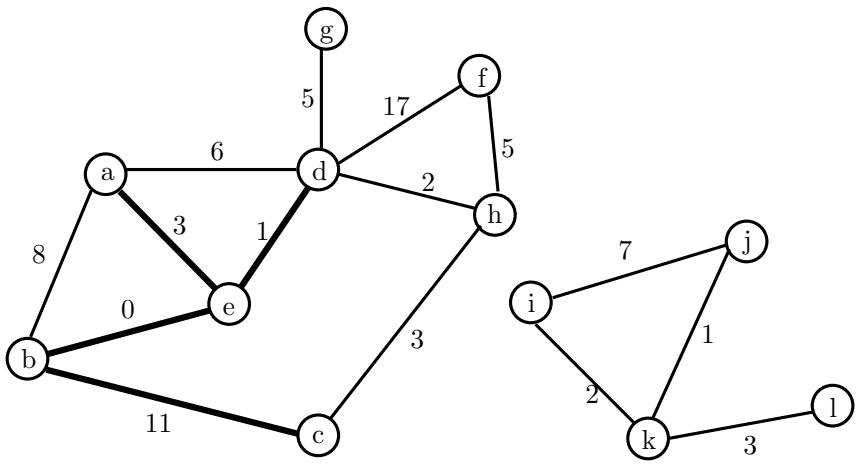


Q: | b | d | c | f | g | h | i | j | k | l |

$S : \{a, e\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	NIL	∞
d	e	4
e	a	3
f	NIL	∞
g	NIL	∞
h	NIL	∞
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

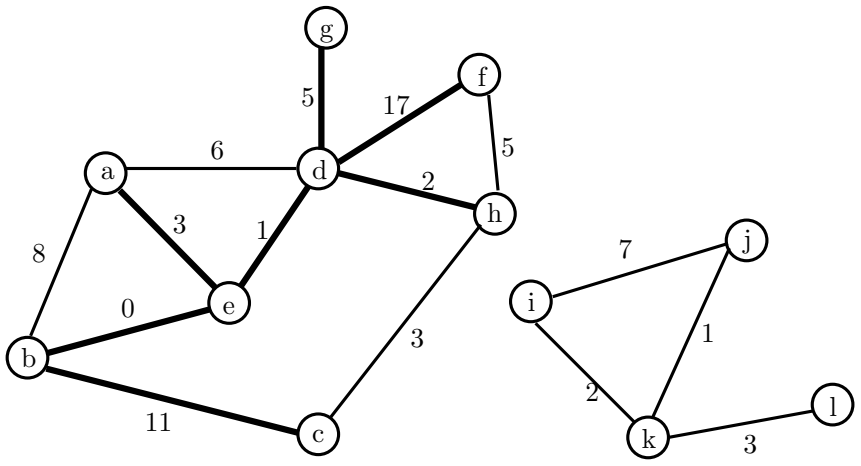


Q: | d | c | f | g | h | i | j | k | l | | | |

$S : \{a, e, b\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	b	14
d	e	4
e	a	3
f	NIL	∞
g	NIL	∞
h	NIL	∞
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

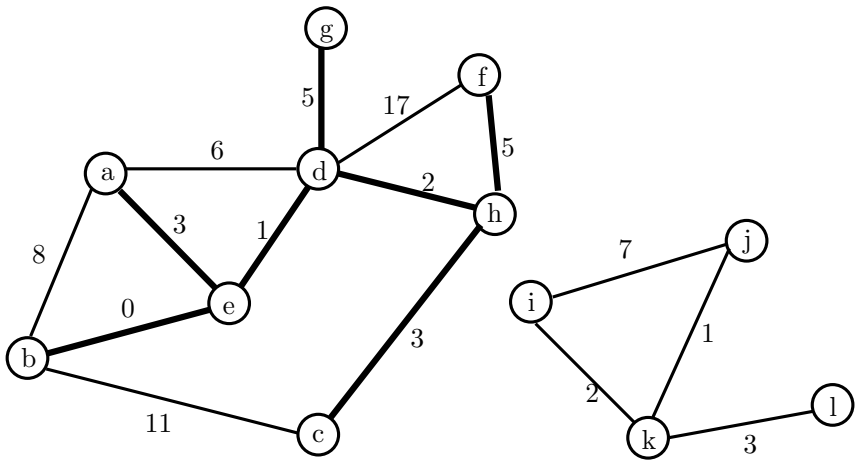


Q: | h | g | c | f | i | j | k | l | | | | |

$S : \{a, e, b, d\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	b	14
d	e	4
e	a	3
f	d	21
g	d	9
h	d	6
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

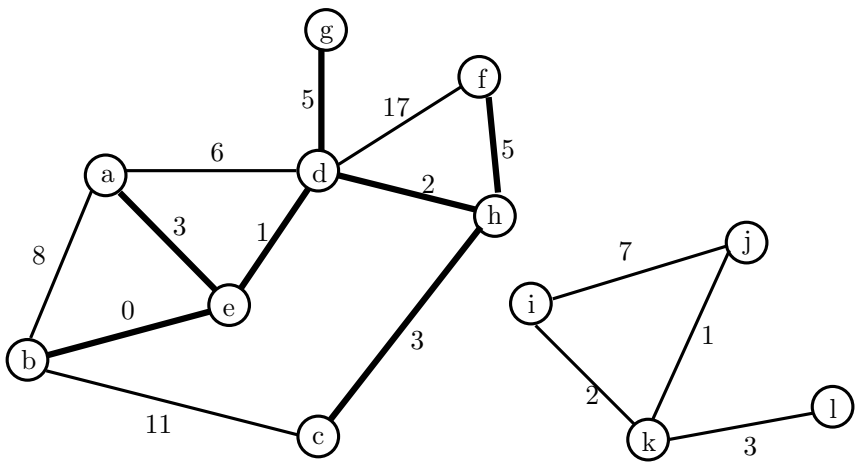


Q: | g | c | f | i | j | k | l | | | | | |

$S : \{a, e, b, d, h\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	h	9
d	e	4
e	a	3
f	h	11
g	d	9
h	d	6
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

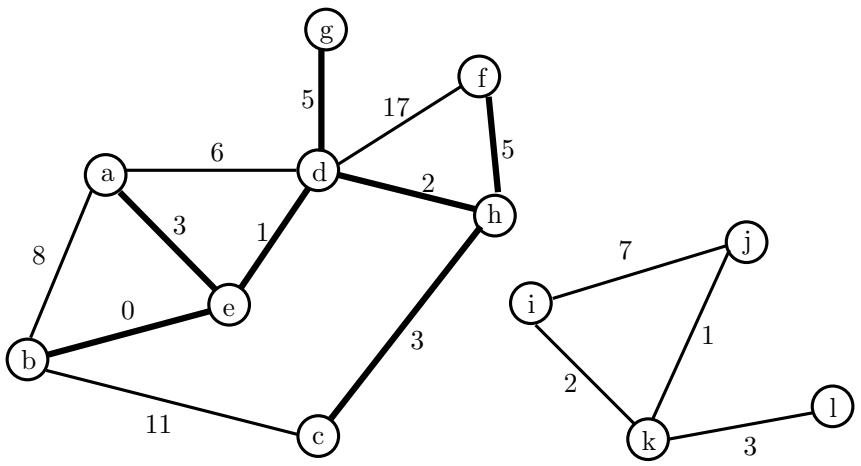


Q: | c | f | i | j | k | l | | | | | | |

$S : \{a, e, b, d, h, g\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	h	9
d	e	4
e	a	3
f	h	11
g	d	9
h	d	6
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

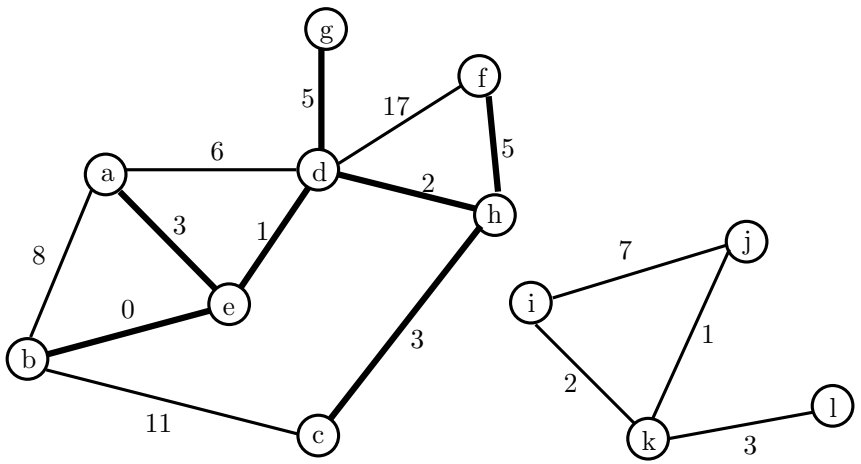


Q: | f | i | j | k | l | | | | | | | |

$S : \{a, e, b, d, h, g, c\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	h	9
d	e	4
e	a	3
f	h	11
g	d	9
h	d	6
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

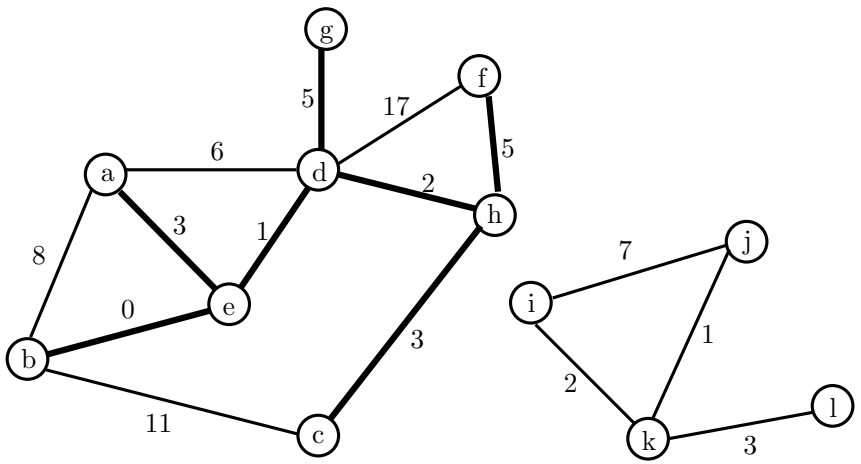


Q: | i | j | k | l | | | | | | | | |

$S : \{a, e, b, d, h.g.c.f\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	h	9
d	e	4
e	a	3
f	h	11
g	d	9
h	d	6
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

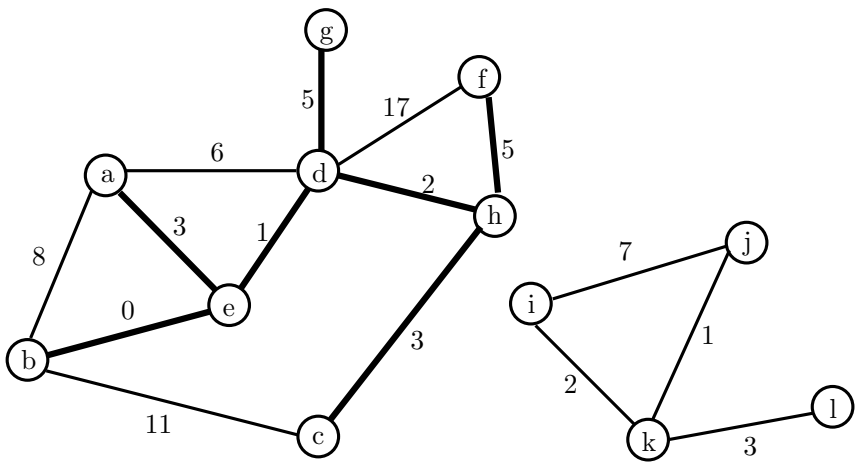


Q: | j | k | l | | | | | | | | | |

$S : \{a, e, b, d, h, g, c, f, i\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	h	9
d	e	4
e	a	3
f	h	11
g	d	9
h	d	6
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

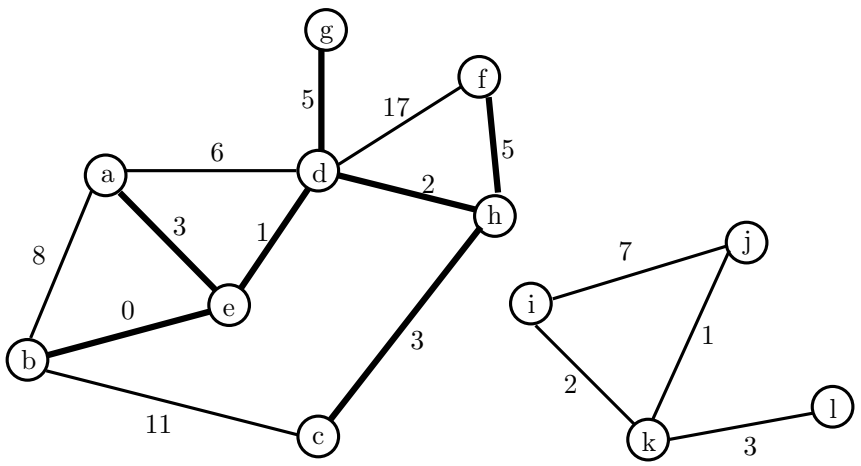


Q: | k | l | | | | | | | | | | |

$S : \{a, e, b, d, h, g, c, f, i, j\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	h	9
d	e	4
e	a	3
f	h	11
g	d	9
h	d	6
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra

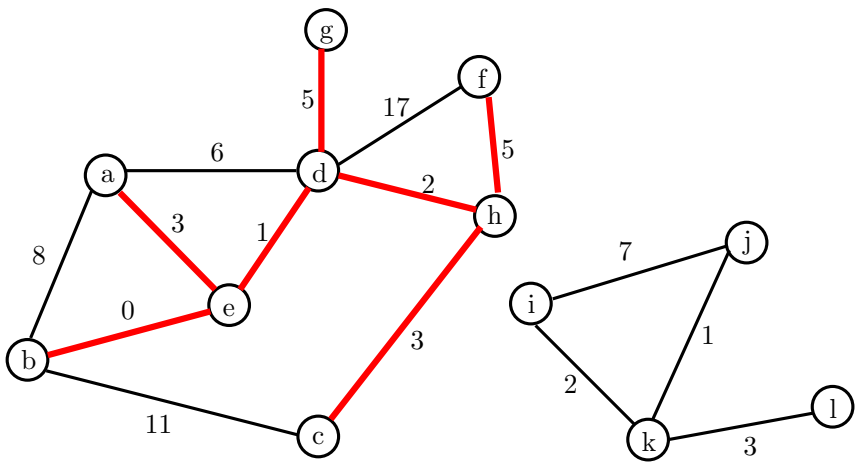


Q: | l | | | | | | | | | | |

$S : \{a, e, b, d, h, g, c, f, i, j, k\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	h	9
d	e	4
e	a	3
f	h	11
g	d	9
h	d	6
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Dijkstra



Q:

$S : \{a, e, b, d, h, g, c, f, i, j, k, l\}$

u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	3
c	h	9
d	e	4
e	a	3
f	h	11
g	d	9
h	d	6
i	NIL	∞
j	NIL	∞
k	NIL	∞
l	NIL	∞

Menor Caminho - Algoritmo de Bellman-Ford

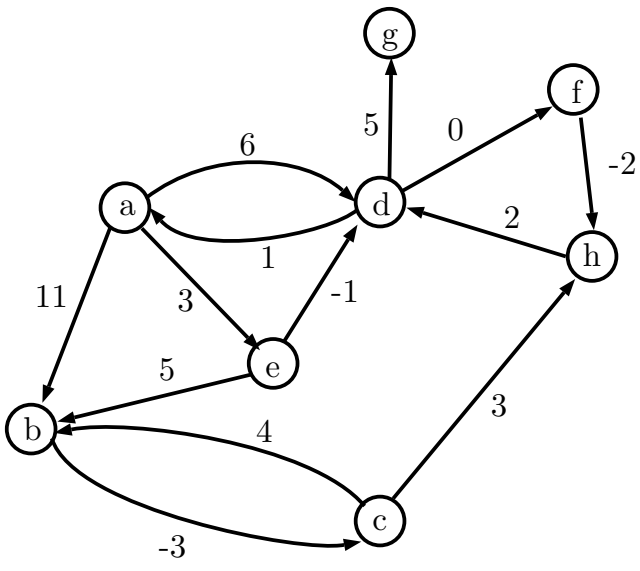
- pré-requisito: único vértice de origem
- grafo orientado
- arestas podem ter peso negativo
- caso exista ciclo de peso negativo, o algoritmo retorna *false*
- complexidade: $O(VE)$

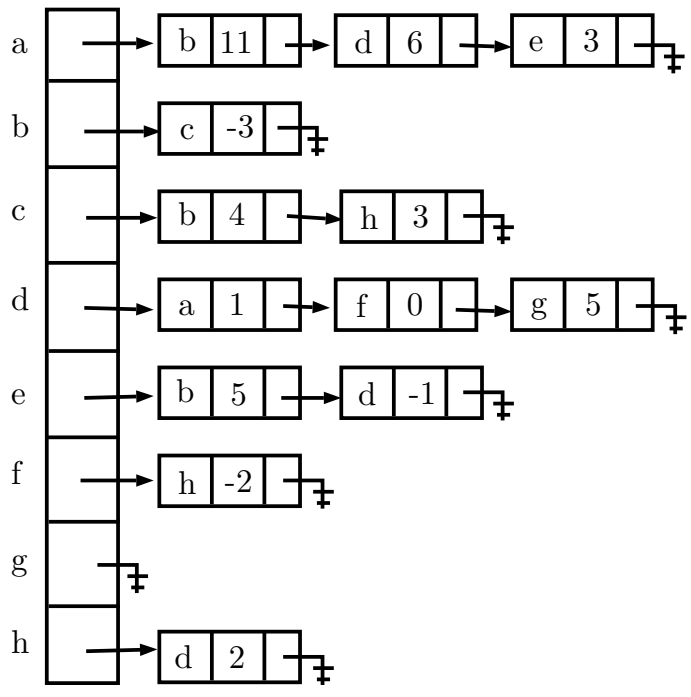
Menor Caminho - Algoritmo de Bellman-Ford

```
1:  function BellmanFord(G, s)
2:      ▷ s é vertice de origem
3:      InicializaOrigem(G, s)
4:      for i ← 1, ..., |V| − 1 do
5:          for each (u, v) ∈ E do
6:              Relax(u, v)
7:          end for
8:      end for
9:      for each (u, v) ∈ E do
10:         if d[v] > d[u] + w(u, v) then
11:             return false                ▷ existe ciclo de peso negativo
12:         end if
13:     end for
14:     return true                         ▷ cálculo efetuado com êxito
15: end function
```

Menor Caminho - Algoritmo de Bellman-Ford

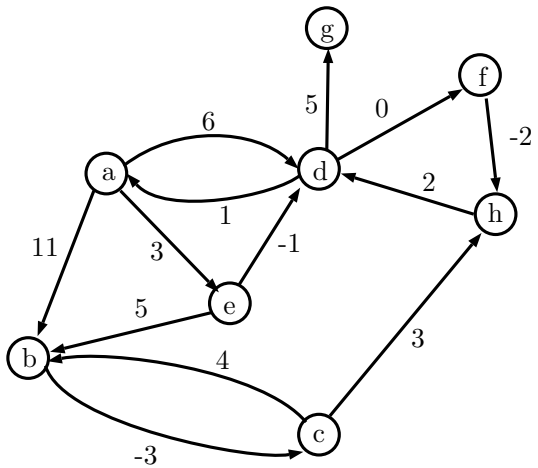
Considere o grafo abaixo tal que o vértice de origem seja *a* e seja usada a representação de listas de adjacência:





Menor Caminho - Algoritmo de Bellman-Ford

Após a execução da linha 03, as estruturas de dados terão:

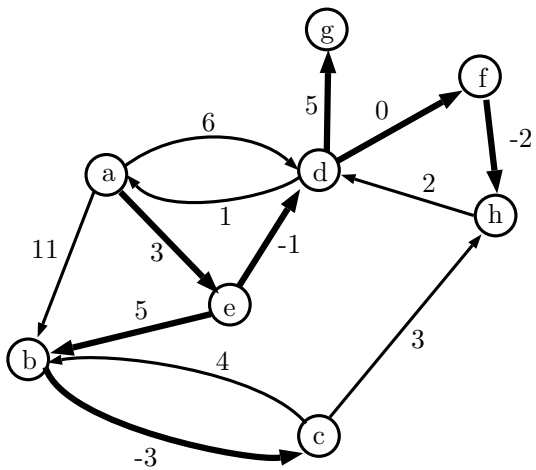


u	$\pi[u]$	$d[u]$
a	NIL	0
b	NIL	∞
c	NIL	∞
d	NIL	∞
e	NIL	∞
f	NIL	∞
g	NIL	∞
h	NIL	∞

Menor Caminho - Algoritmo de Bellman-Ford

Considerando que as arestas serão varridas na ordem lexicográfica:
$\{(a,b),(a,d),(a,e),(b,c),(c,b),(c,h),(d,a),(d,f),(d,g),$
$(e,b),(e,d),(f,h),(h,d)\}$

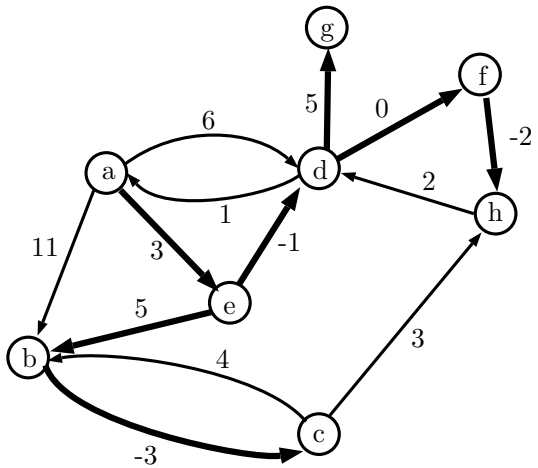
$i = 1$:



u	$\pi[u]$	$d[u]$
a	NIL	0
b	~~NIL~~ ~~a~~ e	~~∞~~ ~~11~~ 8
c	~~NIL~~ b	~~∞~~ 8
d	~~NIL~~ ~~a~~ e	~~∞~~ ~~6~~ 2
e	~~NIL~~ a	~~∞~~ 3
f	~~NIL~~ d	~~∞~~6
g	~~NIL~~ d	~~∞~~ 11
h	~~NIL~~ ~~c~~ f	~~∞~~ ~~11~~ 4

Menor Caminho - Algoritmo de Bellman-Ford

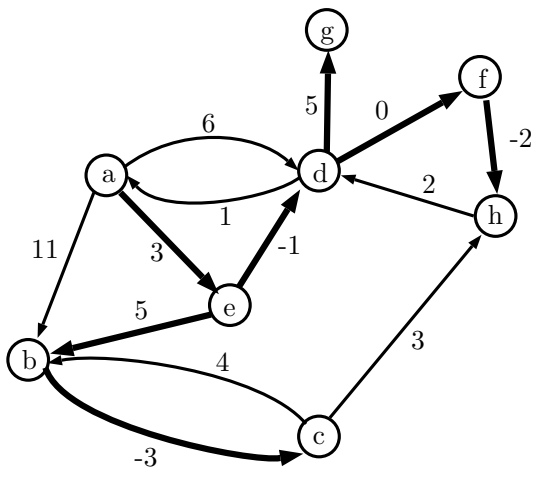
$i = 2$:



u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	8
c	~~b~~ b	~~8~~ 5
d	e	2
e	a	3
f	~~d~~ d	~~6~~ 2
g	~~d~~ d	~~11~~ 7
h	~~f~~ f	~~4~~ 0

Menor Caminho - Algoritmo de Bellman-Ford

A partir de $i = 3$, não há mais alterações, ficando os seguintes valores:



u	$\pi[u]$	$d[u]$
a	NIL	0
b	e	8
c	b	5
d	e	2
e	a	3
f	d	2
g	d	7
h	f	0

Menor Caminho - Algoritmo de Bellman-Ford

Observações:

- o número de repetições do *loop* externo é $V - 1$ porque o tamanho máximo de um caminho mínimo é $V - 1$ arestas
- como em muitos grafos o número máximo de arestas em qualquer caminho mínimo é substancialmente menor que $V - 1$, algumas poucas rodadas de atualizações são necessárias. Portanto, faz sentido adicionar uma verificação extra no algoritmo para terminá-lo imediatamente depois de qualquer rodada em que nenhuma atualização ocorreu.
- as linhas 09-13 fazem uma rodada adicional para detectar a presença de ciclo negativo, detectado pela possiblidade de melhorar o caminho mínimo mais que $V - 1$ vezes

Menor Caminho - grafos orientados acíclicos

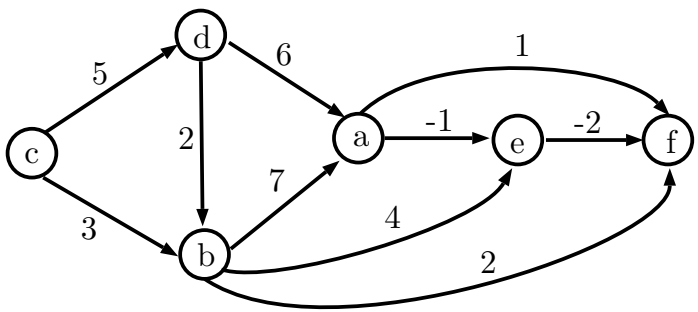
- pré-requisito: único vértice origem
- o problema de menores caminhos em grafos orientados acíclicos está sempre bem definido, pois não existem ciclos. Logo não há ciclos negativos
- se existe caminho de u para v, então u precede v na ordem topológica, e também no menor caminho
- Complexidade: $O(V + E)$

Menor Caminho - grafos orientados acíclicos

```
1: procedure MenorCaminhoGrafoOrientadoAciclico(G, s)
2:     ▷ s é vertice de origem
3:     InicializaOrigem(G, s)
4:     for each u tomado em ordem topológica do
5:         for each v ∈ Adj[u] do
6:             Relax(u, v)
7:         end for
8:     end for
9: end procedure
```

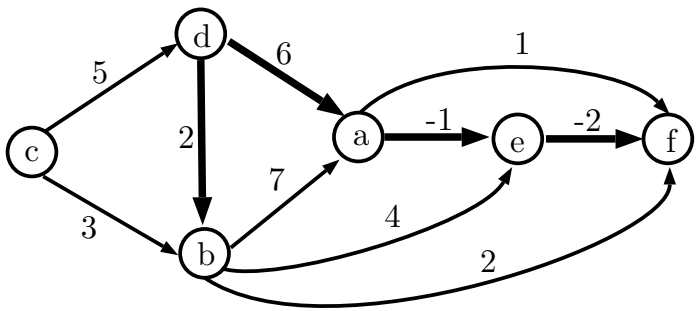
Menor Caminho - grafos orientados acíclicos

Considere o grafo da figura abaixo. Uma ordem topológica válida é c, d, b, a, e, f.
Considere o vértice d como origem. Os valores nas estruturas de dados após a execução da linha 03:



u	$\pi[u]$	$d[u]$
a	NIL	∞
b	NIL	∞
c	NIL	∞
d	NIL	0
e	NIL	∞
f	NIL	∞

Menor Caminho - grafos orientados acíclicos



u	$\pi[u]$	$d[u]$
a	~~NIL~~ d	~~∞~~ 6
b	~~NIL~~ d	~~∞~~ 2
c	NIL	∞
d	NIL	0
e	~~NIL~~ ~~b~~ a	~~∞~~ ~~6~~ 5
f	~~NIL~~ ~~b~~ e	~~∞~~ ~~4~~ 3

Menor Caminho - grafos orientados acíclicos

Observação: o algoritmo anterior pode ser usado para calcular o caminho mais longo em grafo orientados acíclicos. Basta que se troquem os sinais dos pesos das arestas. Assim, o menor caminho no grafo com os pesos modificados corresponderá ao caminho mais longo no grafo original.

Menor Caminho - Algoritmo de Floyd-Warshall

- pré-requisito: ausência de ciclos negativos
- calcula o menor caminho entre todos os pares de vértices
- estrutura de dados: matriz adjacência modificada $W = w_{ij}$, onde:

$$w_{ij} = \begin{cases} 0 & \text{se} \quad i = j \\ w(i,j) & \text{se} \quad i \neq j \text{ e } (i,j) \in E \\ \infty & \text{se} \quad i \neq j \text{ e } (i,j) \notin E \end{cases}$$

Menor Caminho - Algoritmo de Floyd-Warshall

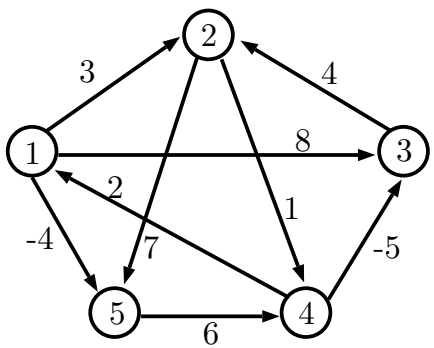
- usa a técnica de Programação Dinâmica
- estrutura de dados: matriz $D^{(k)}$, que armazena o custo do menor caminho com até k vértices intermediários
- complexidade: $O(V^3)$

Menor Caminho - Algoritmo de Floyd-Warshall

```
1:  function FloydWarshall(W)
2:      ▷ entrada: W é matriz adjacência modificada
3:      ▷ saída: matriz D com os valores dos menores caminhos
4:      D^(0) ← W                                   ▷ atribuição de matrizes
5:      for k ← 1, ..., |V| do
6:          for i ← 1, ..., |V| do
7:              for j ← 1, ..., |V| do
8:                  d_ij^(k) ← min{d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1)}
9:              end for
10:         end for
11:     end for
12:     return D
13: end function
```

Menor Caminho - Algoritmo de Floyd-Warshall

Considere o grafo abaixo. Aplicando o algoritmo, a matriz não sofre mais alterações a partir de $D^{(4)}$



Menor Caminho - Algoritmo de Floyd-Warshall

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \quad D^{(4)} = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

Fecho Transitivo

Fecho Transitivo

Dado grafo orientado $G = (V, E)$, o **fecho transitivo** de G é definido como sendo o grafo $G^* = (V, E^*)$ onde:

$$E^* = \{(i, j) : \text{existe caminho de } i \text{ para } j \text{ em } G\}$$

Basicamente corresponde a determinar se existe caminho entre todos os pares de vértices.

Fecho Transitivo

- ideia do algoritmo: atribuir peso 1 a cada aresta e executar o algoritmo de Floyd-Warshall
- estrutura de dados: matriz booleana $t_{ij}^{(k)}$, que indica se existe caminho em G de i para j com no máximo k arestas.
- complexidade: $O(V^3)$

Fecho Transitivo de Grafos Orientados

```
1:  function FechoTransitivo(G)
2:      ▷ saída: matriz T contendo o fecho transitivo
3:      for k ← 1, ..., |V| do                          ▷ inicialização da matriz T
4:          for i ← 1, ..., |V| do
5:              if i = j or (i, j) ∈ E then
6:                  t_ij^(0) ← 1
7:              else
8:                  t_ij^(0) ← 0
9:              end if
10:         end for
11:     end for
12:     for k ← 1, ..., |V| do
13:         for i ← 1, ..., |V| do
14:             for j ← 1, ..., |V| do
15:                 t_ij^(k) ← t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
16:             end for
17:         end for
18:     end for
19:     return T^(|V|)
20: end function
```

Árvore Geradora Mínima

Árvore Geradora

Dado um grafo $G = (V, E)$, uma **árvore geradora** é um subgrafo $T = (V, E^T)$ tal que $E^T \subseteq E$ e T é árvore.

Árvore Geradora Mínima

Seja $G = (V, E)$ grafo não orientado conexo tal que esteja associado pesos (custos) não negativos às arestas de G. A **árvore geradora mínima** é a árvore geradora de peso total mínimo, isto é:

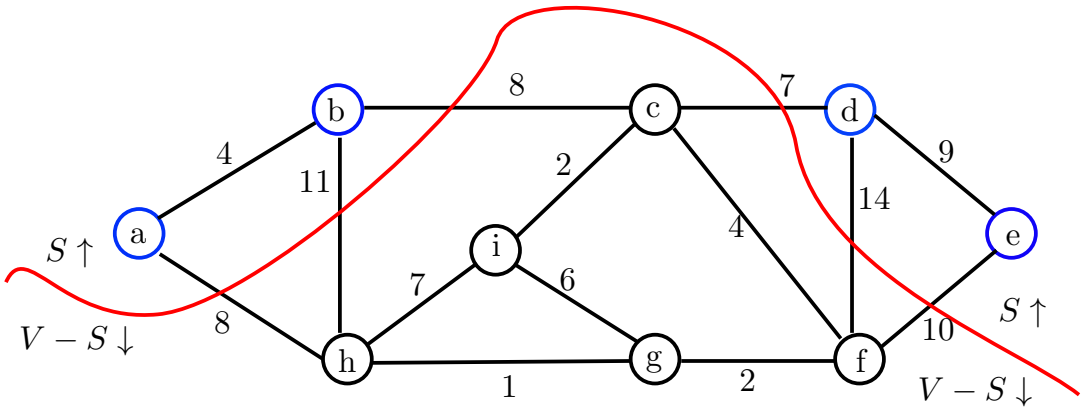
$$w(T) = \sum_{(u,v) \in T} w(u, v) \text{ é mínimo}$$

Árvore Geradora Mínima

Definições auxiliares

- Um **corte** $[S, V - S]$ de um grafo $G = (V, E)$ é uma partição de V
- uma aresta (u, v) cruza o corte $[S, V - S]$ se um extremo está em S e o outro extremo está em $V - S$
- um corte $[S, V - S]$ respeita um conjunto A de arestas, se nenhuma aresta em A cruza o corte

Árvore Geradora Mínima



- aresta (c, d) cruza o corte
- o conjunto $A = \{(a, b), (c, i), (c, f), (f, g), (g, h)\}$ respeita o corte

Árvore Geradora Mínima - algoritmo de Kruskal

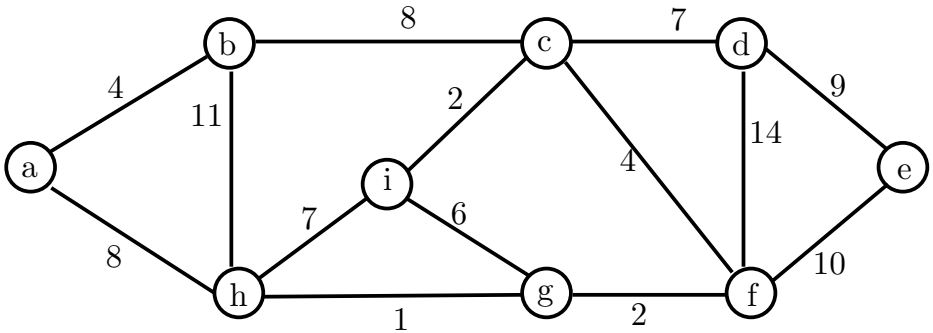
Algoritmo de Kruskal

- definir um conjunto de aresta A
- adicionar arestas "seguras" ao conjunto A até formar a árvore geradora mínima
- as arestas candidatas são arestas de peso mínimo que cruzam algum corte que respeita o conjunto A
- complexidade: $O(E \lg E)$

Árvore Geradora Mínima - algoritmo de Kruskal

```
1:  function KRUSKALL(G, w)
2:      ▷ entrada: grafo G e pesos das arestas w
3:      ▷ saída: conjunto A contendo as arestas da árvore geradora mínima
4:      A ← ∅
5:      for each v ∈ V[G] do
6:          criaConjunto(v)                    ▷ conjunto contendo somente o vértice v
7:      end for
8:      ordene as arestas de E em ordem não decrescente de w
9:      for each (u, v) ∈ E em ordem de peso w do
10:         if conj(u) ≠ conj(v) then                          ▷ subárvores distintas
11:             A ← A ∪ {(u, v)}
12:             Uniao(conj(u), conj(v))
13:         end if
14:     end for
15:     return A
16: end function
```

Árvore Geradora Mínima

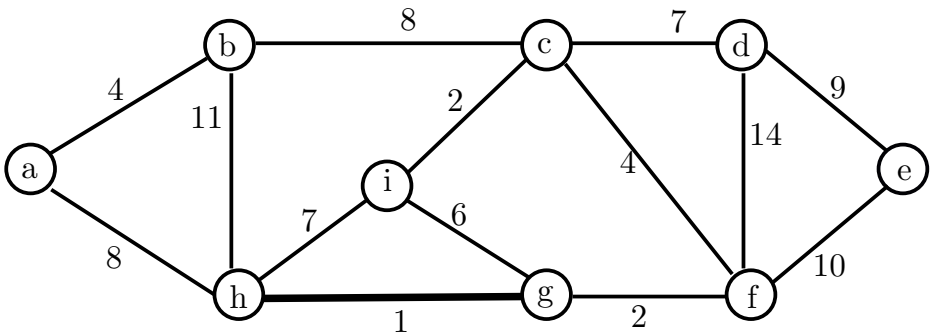


- valores das estruturas de dados após execução das linhas 04-08:
  - conjuntos de vértices: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$
  - arestas em ordem não decrescente:

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

  - $A = \emptyset$

Árvore Geradora Mínima

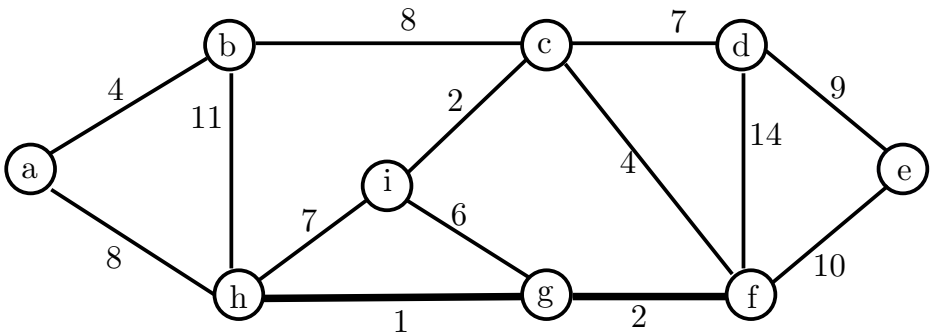


- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}$
  - arestas em ordem não decrescente:

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

  - $A = \{(g, h)\}$

Árvore Geradora Mínima

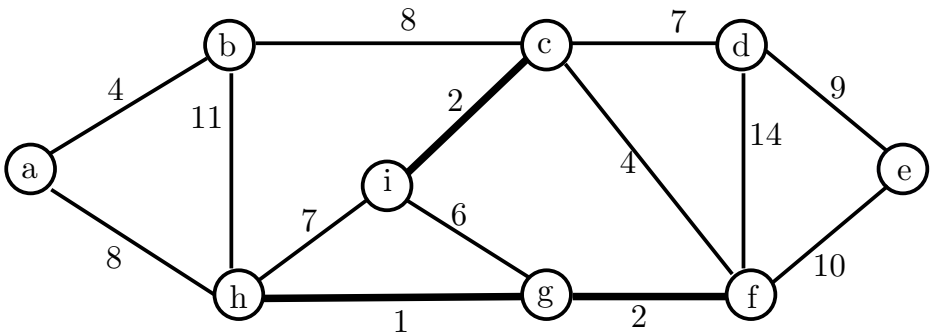


- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f, g, h\}, \{i\}$
  - arestas em ordem não decrescente:

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

  - $A = \{(g, h), (f, g)\}$

Árvore Geradora Mínima

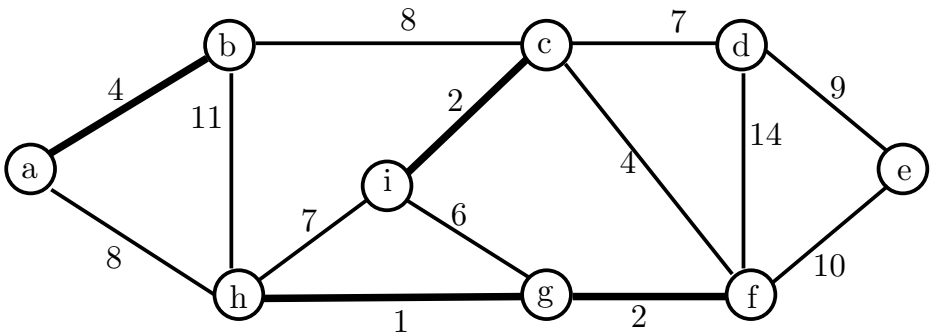


- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$
  - arestas em ordem não decrescente:

4	4	6	7	7	8	8	9	10	11	14
(a, b)	(c, f)	(g, i)	(c, d)	(h, i)	(a, h)	(b, c)	(d, e)	(e, f)	(b, h)	(d, f)

  - $A = \{(g, h), (f, g), (c, i)\}$

Árvore Geradora Mínima

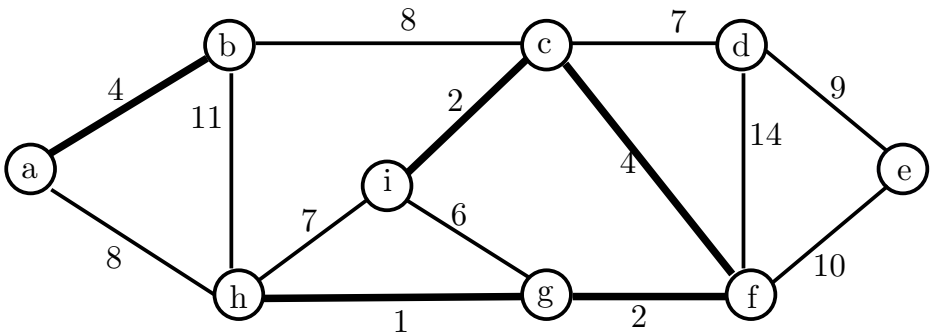


- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$
  - arestas em ordem não decrescente:

4	6	7	7	8	8	9	10	11	14
(c, f)	(g, i)	(c, d)	(h, i)	(a, h)	(b, c)	(d, e)	(e, f)	(b, h)	(d, f)

  - $A = \{(g, h), (f, g), (c, i), (a, b)\}$

Árvore Geradora Mínima

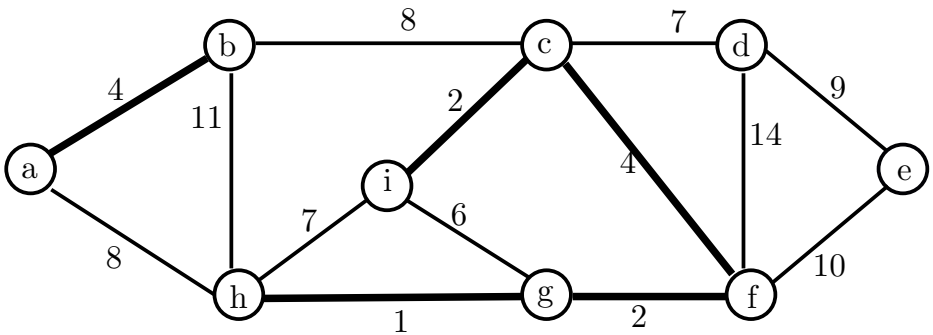


- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b\}, \{c, f, g, h, i\}, \{d\}, \{e\}$
  - arestas em ordem não decrescente:

6	7	7	8	8	9	10	11	14
(g, i)	(c, d)	(h, i)	(a, h)	(b, c)	(d, e)	(e, f)	(b, h)	(d, f)

  - $A = \{(g, h), (f, g), (c, i), (a, b), (c, f)\}$

Árvore Geradora Mínima

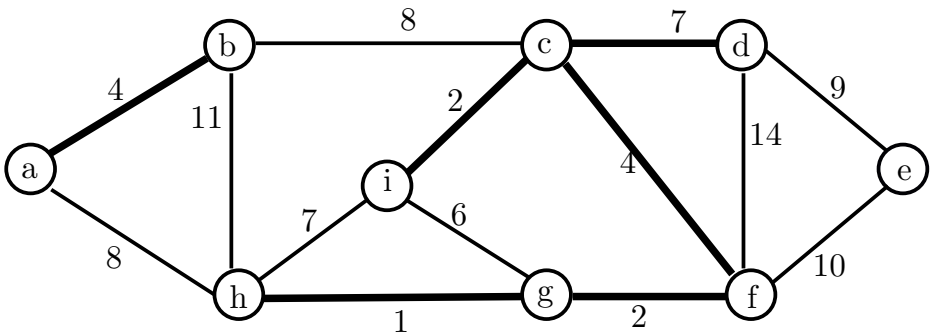


- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b\}, \{c, f, g, h, i\}, \{d\}, \{e\}$
  - arestas em ordem não decrescente:

7	7	8	8	9	10	11	14
(c, d)	(h, i)	(a, h)	(b, c)	(d, e)	(e, f)	(b, h)	(d, f)

  - $A = \{(g, h), (f, g), (c, i), (a, b), (c, f)\}$

Árvore Geradora Mínima

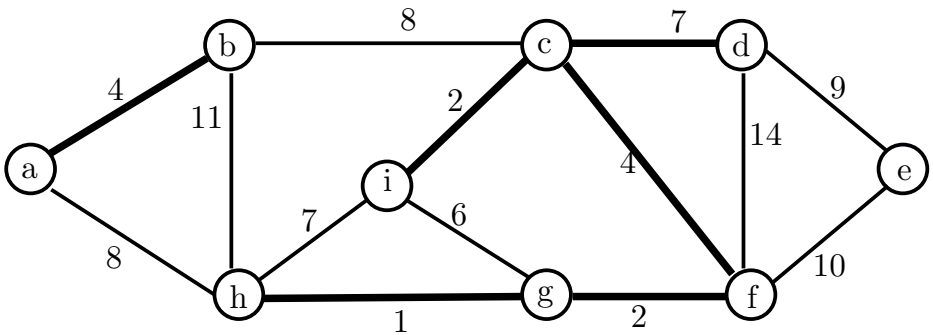


- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b\}, \{c, d, f, g, h, i\}, \{e\}$
  - arestas em ordem não decrescente:

7	8	8	9	10	11	14
(h, i)	(a, h)	(b, c)	(d, e)	(e, f)	(b, h)	(d, f)

  - $A = \{(g, h), (f, g), (c, i), (a, b), (c, f), (c, d)\}$

Árvore Geradora Mínima

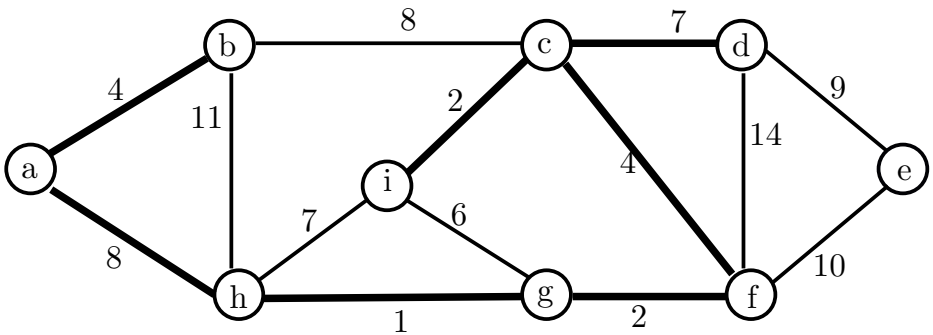


- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b\}, \{c, d, f, g, h, i\}, \{e\}$
  - arestas em ordem não decrescente:

8	8	9	10	11	14
(a, h)	(b, c)	(d, e)	(e, f)	(b, h)	(d, f)

  - $A = \{(g, h), (f, g), (c, i), (a, b), (c, f), (c, d)\}$

Árvore Geradora Mínima

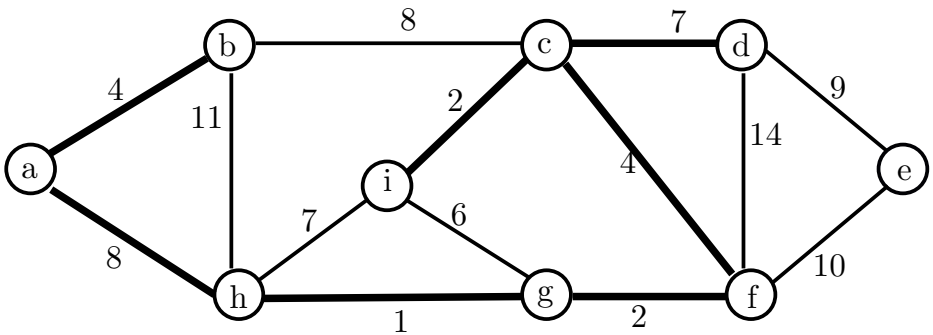


- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b, c, d, f, g, h, i\}, \{e\}$
  - arestas em ordem não decrescente:

8	9	10	11	14
(b, c)	(d, e)	(e, f)	(b, h)	(d, f)

  - $A = \{(g, h), (f, g), (c, i), (a, b), (c, f), (c, d), (a, h)\}$

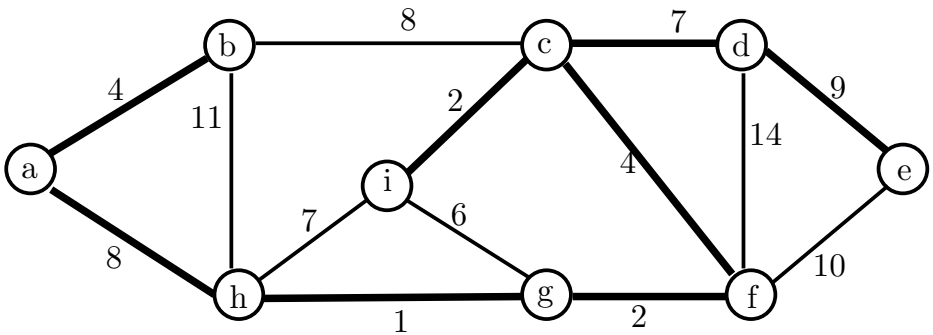
Árvore Geradora Mínima



- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b, c, d, f, g, h, i\}, \{e\}$
  - arestas em ordem não decrescente:

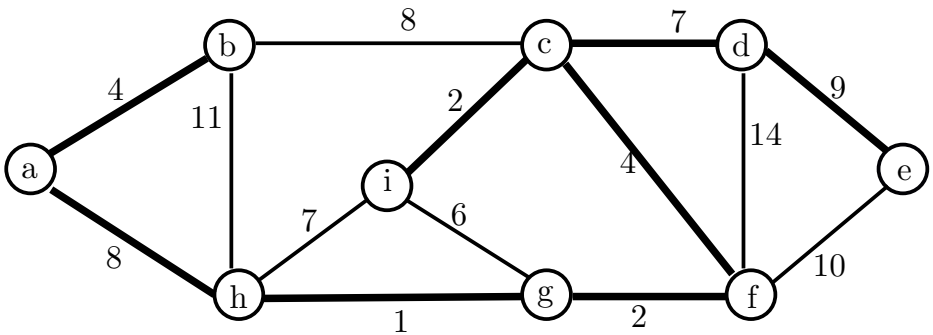
    $\underset{(d,e)}{9}$ $\underset{(e,f)}{10}$ $\underset{(b,h)}{11}$ $\underset{(d,f)}{14}$
  - $A = \{(g, h), (f, g), (c, i), (a, b), (c, f), (c, d), (a, h)\}$

Árvore Geradora Mínima



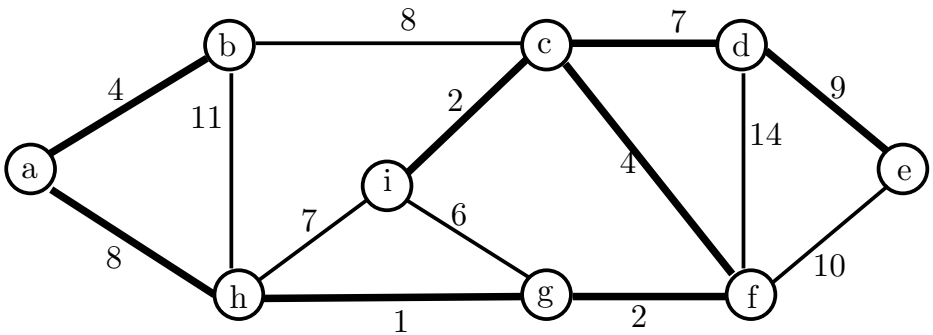
- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b, c, d, e, f, g, h, i\}$
  - arestas em ordem não decrescente:
    $\overset{10}{(e, f)}$ $\overset{11}{(b, h)}$ $\overset{14}{(d, f)}$
  - $A = \{(g, h), (f, g), (c, i), (a, b), (c, f), (c, d), (a, h), (d, e)\}$

Árvore Geradora Mínima



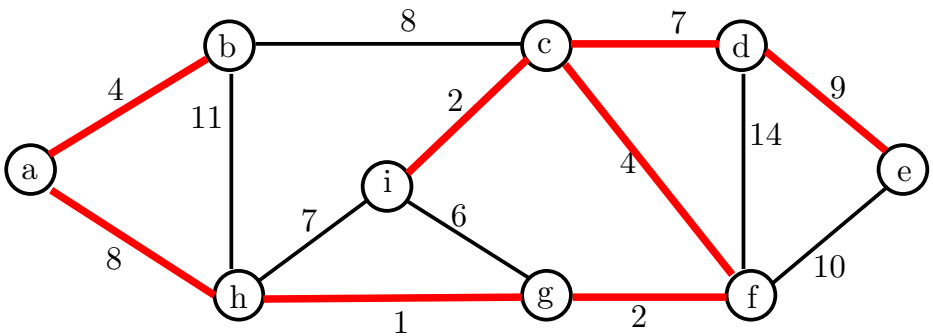
- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b, c, d, e, f, g, h, i\}$
  - arestas em ordem não decrescente:
    $\overset{11}{(b, h)}$ $\overset{14}{(d, f)}$
  - $A = \{(g, h), (f, g), (c, i), (a, b), (c, f), (c, d), (a, h), (d, e)\}$

Árvore Geradora Mínima



- valores das estruturas de dados a cada iteração do *loop* das linhas 09-14:
  - conjuntos de vértices: $\{a, b, c, d, e, f, g, h, i\}$
  - arestas em ordem não decrescente:
    $\overset{14}{(d, f)}$
  - $A = \{(g, h), (f, g), (c, i), (a, b), (c, f), (c, d), (a, h), (d, e)\}$

Árvore Geradora Mínima



- resultado final:
  $A = \{(g, h), (f, g), (c, i), (a, b), (c, f), (c, d), (a, h), (d, e)\}$
- peso mínimo total: $1 + 2 + 2 + 4 + 4 + 7 + 8 + 9 = 37$

Árvore Geradora Mínima - algoritmo de Prim

Algoritmo de Prim

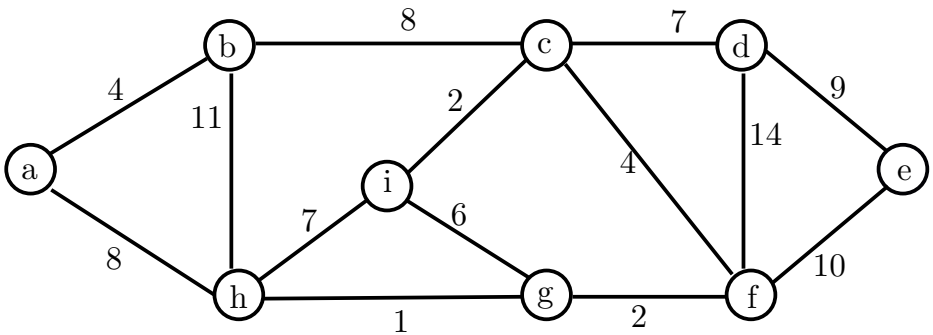
- semelhante ao algoritmo de Dijkstra para menor caminho
- uso de fila de prioridade Q para armazenar vértices que ainda não fazem parte da árvore: baseada no campo $key[v]$ que contém peso mínimo de cada aresta conectando v a um vértice da árvore
- fornecido vértice inicial (raiz da árvore)
- complexidade: $O(E \lg V)$

Árvore Geradora Mínima - algoritmo de Prim

```
1:  procedure PRIM(G, w, r)
2:      ▷ entrada: grafo G, pesos das arestas w e vértice origem r
3:      ▷ saída: árvore geradora mínima
4:      Q ← V[G]
5:      for each u ∈ V[G] do
6:          key[u] ← ∞
7:      end for
8:      key[r] ← 0
9:      π[r] ← NIL                                   ▷ raiz da árvore
10:     while Q ≠ ∅ do
11:         u ← extraiMinimo(Q)
12:         for each v ∈ Adj[u] do
13:             if v ∈ Q and w(u, v) < key[v] then
14:                 π[v] ← u                  ▷ u precede v na árvore geradora mínima
15:                 key[v] ← w(u, v)
16:             end if
17:         end for
18:     end while
19: end procedure
```

Árvore Geradora Mínima - algoritmo de Prim

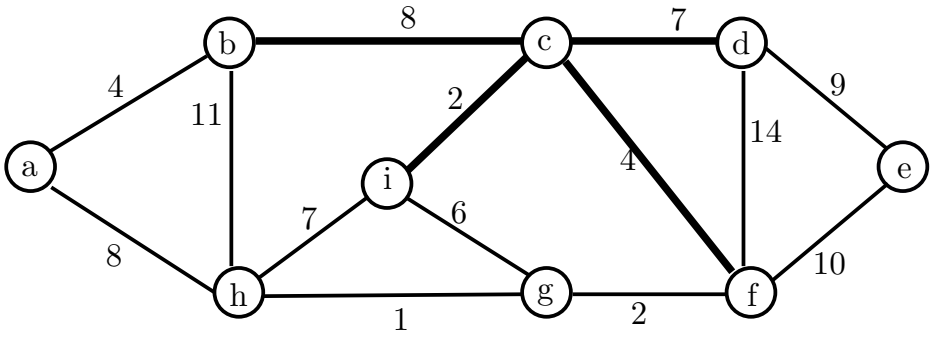
Seja o grafo abaixo. Considere o vértice c como origem.



- valores das estruturas após a execução das linhas 04-09:
  - $Q : c, a, b, d, e, f, g, h, i$

vértice	π	*key*
a		∞
b		∞
c	NIL	0
d		∞
e		∞
f		∞
g		∞
h		∞
i		∞

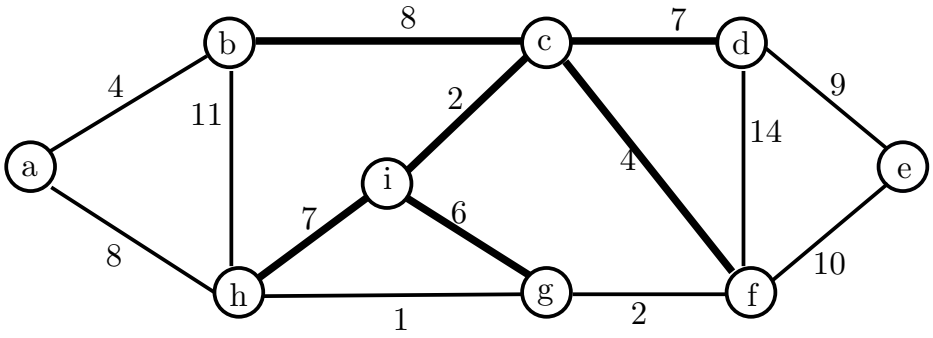
Árvore Geradora Mínima - algoritmo de Prim



- valores das estruturas ao fim de cada iteração do laço *while*:
  - $Q : i, f, d, b, a, e, g, h$

vértice	π	*key*
a		∞
b	c	8
c	NIL	0
d	c	7
e		∞
f	c	4
g		∞
h		∞
i	c	2

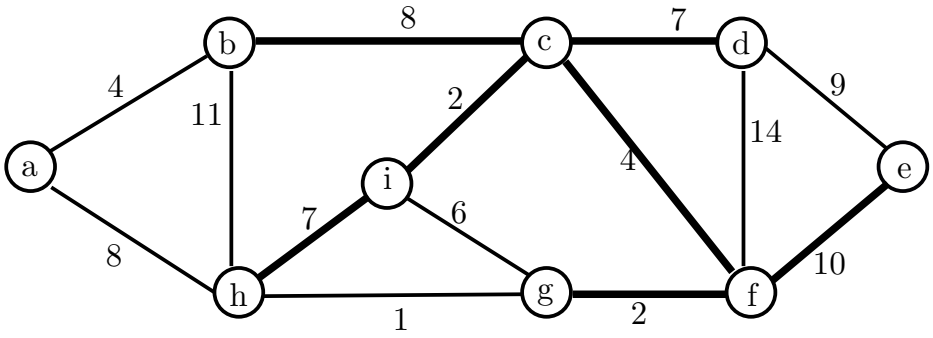
Árvore Geradora Mínima - algoritmo de Prim



- valores das estruturas ao fim de cada iteração do laço *while*:
  - $Q : f, g, d, h, b, a, e$

vértice	π	*key*
a		∞
b	c	8
c	NIL	0
d	c	7
e		∞
f	c	4
g	i	6
h	i	7
i	c	2

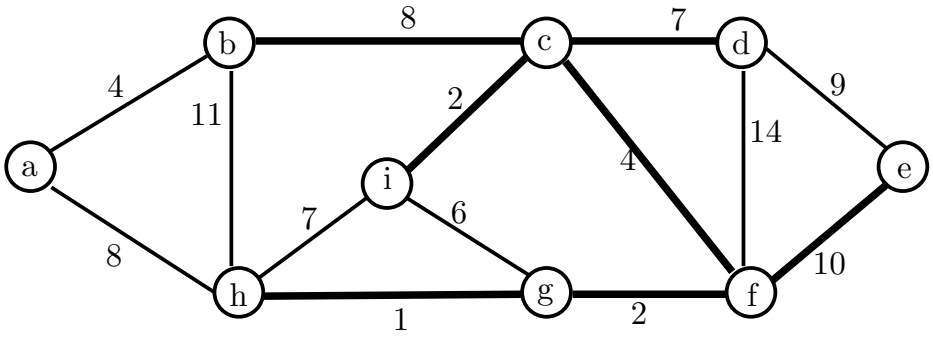
Árvore Geradora Mínima - algoritmo de Prim



- valores das estruturas ao fim de cada iteração do laço *while*:
  - $Q : g, d, h, b, e, a$

vértice	π	*key*
a		∞
b	c	8
c	NIL	0
d	c	7
e	f	10
f	c	4
g	f	2
h	i	7
i	c	2

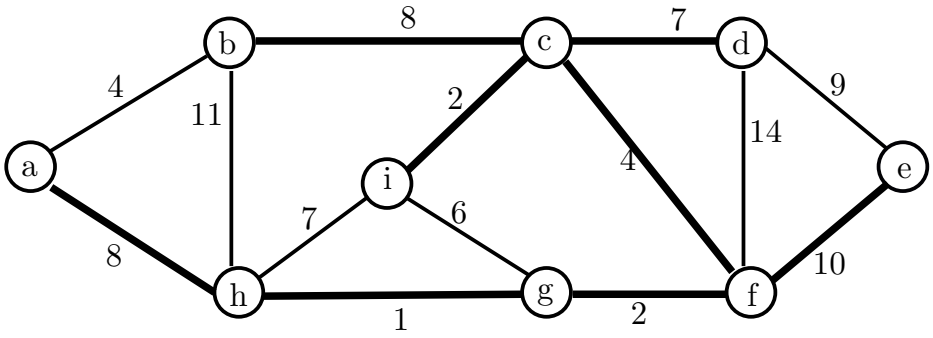
Árvore Geradora Mínima - algoritmo de Prim



- valores das estruturas ao fim de cada iteração do laço *while*:
  - $Q : h, d, b, e, a$

vértice	π	*key*
a		∞
b	c	8
c	NIL	0
d	c	7
e	f	10
f	c	4
g	f	2
h	g	1
i	c	2

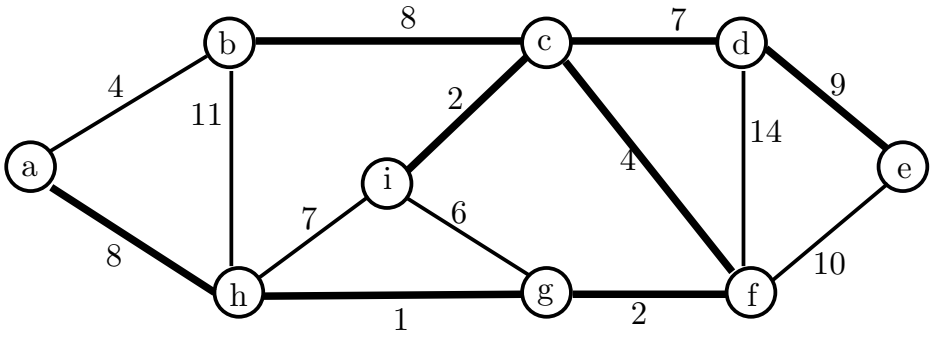
Árvore Geradora Mínima - algoritmo de Prim



- valores das estruturas ao fim de cada iteração do laço *while*:
  - $Q : d, b, a, e$

vértice	π	*key*
a	h	8
b	c	8
c	NIL	0
d	c	7
e	f	10
f	c	4
g	f	2
h	g	1
i	c	2

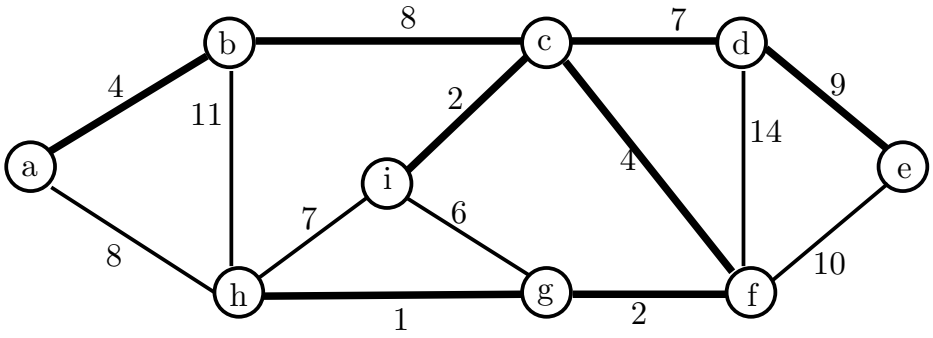
Árvore Geradora Mínima - algoritmo de Prim



- valores das estruturas ao fim de cada iteração do laço *while*:
  - $Q : b, a, e$

vértice	π	*key*
a	h	8
b	c	8
c	NIL	0
d	c	7
e	d	9
f	c	4
g	f	2
h	g	1
i	c	2

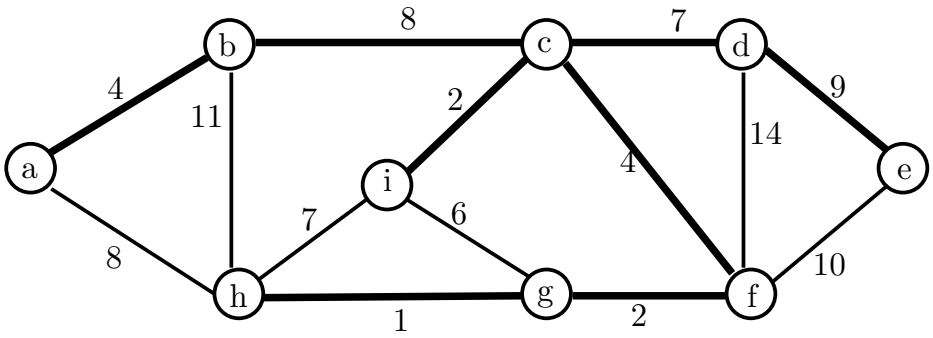
Árvore Geradora Mínima - algoritmo de Prim



- valores das estruturas ao fim de cada iteração do laço *while*:
  - $Q : a, e$

vértice	π	*key*
a	b	4
b	c	8
c	NIL	0
d	c	7
e	d	9
f	c	4
g	f	2
h	g	1
i	c	2

Árvore Geradora Mínima - algoritmo de Prim

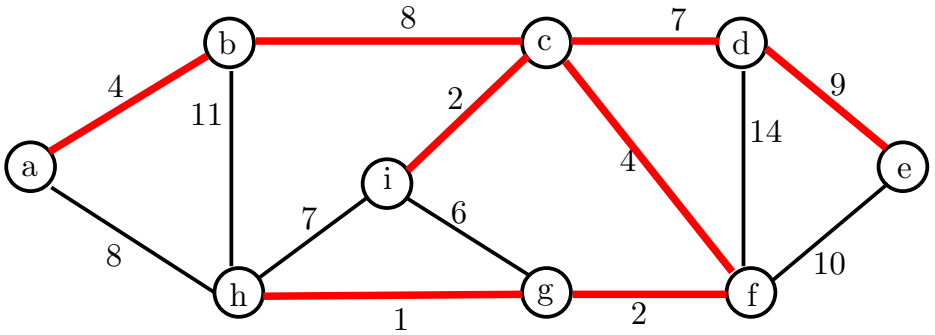


- valores das estruturas ao fim de cada iteração do laço *while*:
  - Q : e

vértice	π	*key*
a	b	4
b	c	8
c	NIL	0
d	c	7
e	d	9
f	c	4
g	f	2
h	g	1
i	c	2

Árvore Geradora Mínima - algoritmo de Prim

resultado final:



- peso total mínimo: $4 + 8 + 7 + 9 + 4 + 2 + 1 + 2 = 37$
- arestas da árvore:
  $\{(a, b), (b, c), (d, c), (e, d), (f, c), (g, f), (h, g), (i, c)\}$ew

vértice	π	*key*
a	b	4
b	c	8
c	NIL	0
d	c	7
e	d	9
f	c	4
g	f	2
h	g	1
i	c	2

Fluxo em Redes

- exemplo de aplicação: dado um conjunto de manilhas (ou canos) interligados, pretende-se determinar o fluxo máximo de água que pode passar por cada ponto

Fluxo em Redes

Definições

Seja $G = (V, E)$ grafo orientado com 2 vértices distintos s (**fonte**) e t (**ralo**), isto é, $g_{in}(s) = 0$ e $g_{out}(t) = 0$, e cada aresta em E está associada uma capacidade c não-negativa.

- a capacidade c de uma aresta é a quantidade máxima de fluxo que pode passar por ela
- **fluxo** é uma função f nas arestas tal que:
  - $0 \leq f(e) \leq c(e)\ \forall e \in E$, isto é, o fluxo não excede a capacidade da aresta
  - $\forall u, v, w \in V - \{s, t\}$: $\sum f(u, v) = \sum (v, w)$, ou seja, o fluxo total que entra em v também sai dele, exceto para s e t
- consequência: $\forall u, v \in V$: $\sum f(s, v) = \sum f(v, t)$, isto é, tudo o que sai da fonte chega ao ralo

Fluxo em Redes

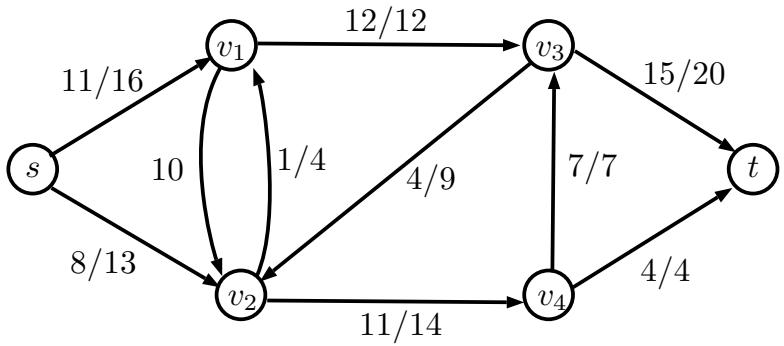
A solução do problema corresponde a maximizar o fluxo.

Fluxo

$f : V \times V \rightarrow \mathbb{R}$ tal que :

- $f(u, v) \leq c(u, v)\ \forall u, v \in V$
- $f(u, v) = -f(v, u)\ \forall u, v \in V$

Fluxo em Redes



- fluxo saindo de s: $11 + 8 = 19$
- capacidade de s: $16 + 13 = 29$
- fluxo passando na aresta $(v_1, v_2) = 0$
- capacidade da aresta (v_1, v_2): 10
- fluxo chegando em t: $15 + 4 = 19$
- capacidade de t: $20 + 4 = 24$

Fluxo em Redes

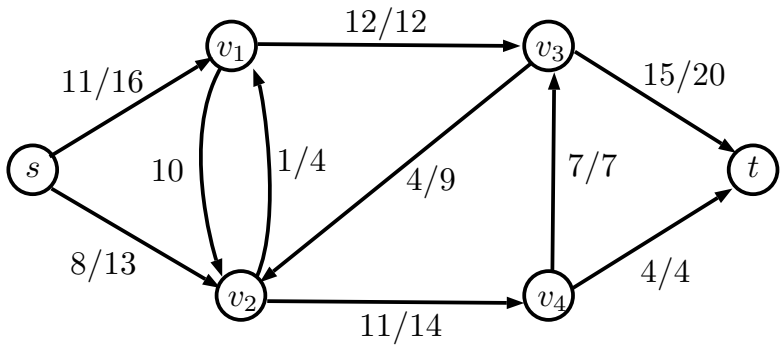
Capacidade Residual

Consiste de aresta que admitem mais fluxo, ou seja: dado um par de vértices u e v, a quantidade de fluxo adicional que podemos empurrar de u para v sem exceder a capacidade $c(u, v)$ é a capacidade residual da aresta (u, v) dado por:

$$c_f(u, v) = c(u, v) - f(u, v)$$

Observação: se $(u, v) \notin E$ então $c(u, v) = 0$

Fluxo em Redes



- $c_f(s, v_1) = 16 - 11 = 5$
- $c_f(v_1, s) = 0 - (-11) = 11$

Fluxo em Redes

Rede Residual

A **rede residual** de um grafo G, induzida pelo fluxo f é $G_f = (V, E_f)$ onde $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$

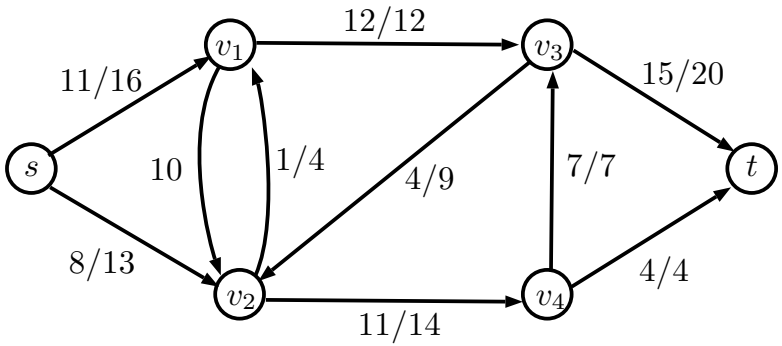
Observação: pode acontecer que $E_f \not\subseteq E$

Caminho Aumentante

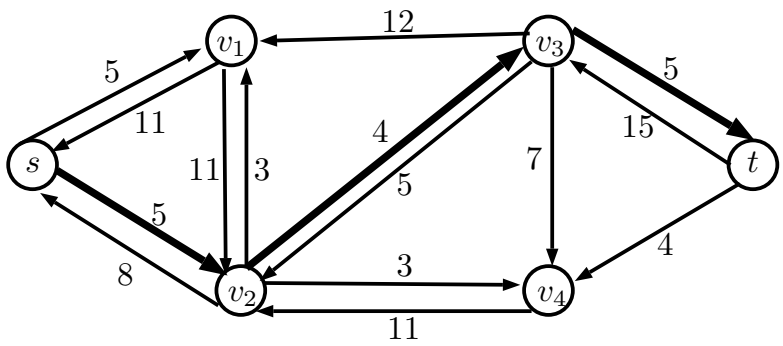
Um **caminho aumentante** é um caminho simmples de s para t na rede residual. Sua capacidade é:
$c_f(p) = min\{c_f(u, v) : (u, v)$ está em $p\}$

Um fluxo é máximo se não há caminho aumentante.

Fluxo em Redes



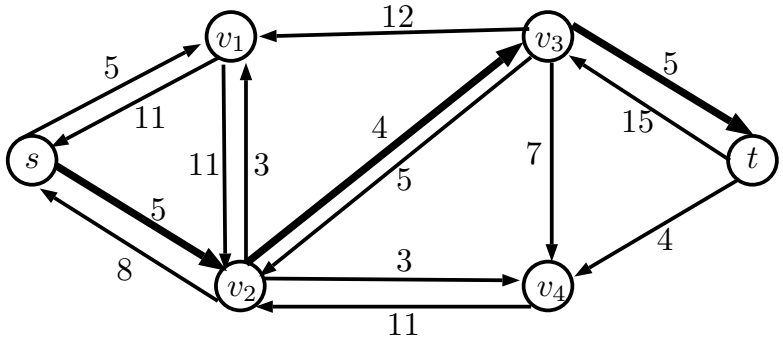
G



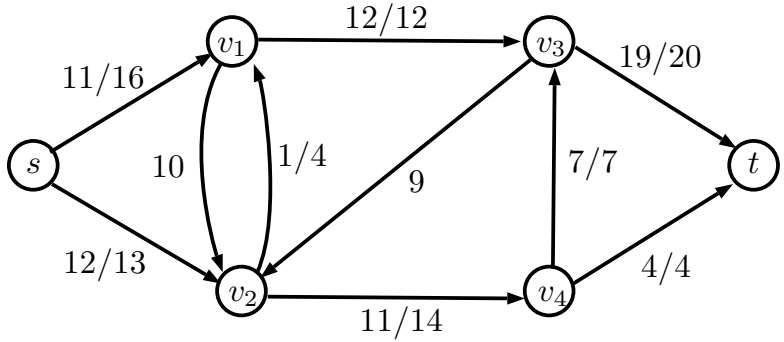
G_f

- exemplo de rede G e rede residual G_f correspondente
- arestas mais espessas indicam um caminho aumentante em G_f da fonte ao ralo, cuja capacidade residual é 4, o que significa que ainda é possível empurrar 4 unidades de fluxo usando esse caminho

Fluxo em Redes



G_f



G resultante

Fluxo em Redes

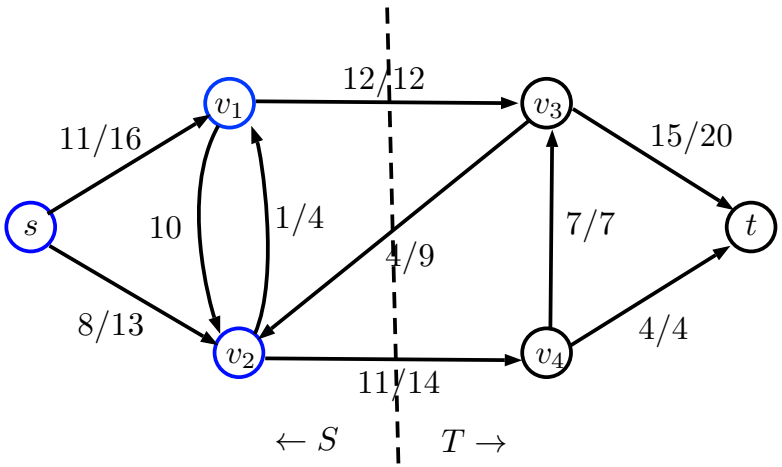
- seja um corte $[S, T]$ em uma rede de fluxos $G = (V, E)$ tal que $T = V - S$ e $s \in S$ e $t \in T$
- definimos $c([S, T])$ a capacidade do corte $[S, T]$ dado por: $\sum c(e)$, tal que $e \in [S, T]$

Teorema

Para qualquer fluxo f e corte $[S, T]$ tem-se que $val\ f \leq c([S, T])$ onde:

$$val\ f = \sum \sum f(u, v) \leq \sum \sum c(u, v)\ \forall u \in S, \forall v \in T$$

Fluxo em Redes



- o fluxo que cruza o corte $f[S, T] = 19$
- capacidade do corte $c([S, T]) = 26$

Fluxo em Redes

Teorema do Fluxo Máximo-Corte Mínimo

Se f é fluxo máximo e $[S, T]$ é corte tal que $val\ f = c([S, T])$ então $c([S, T])$ é mínimo.

- G_f não contém caminho aumentante
- $|f| = c([S, T])$ para algum corte $[S, T]$ de G

Fluxo em Redes - algoritmo de Ford-Fulkerson

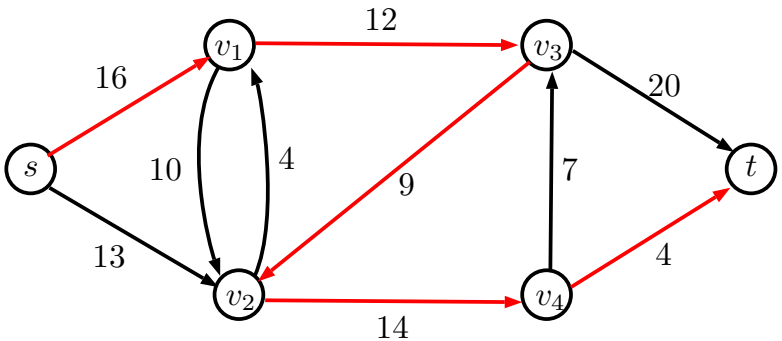
- inicializar fluxo com zero
- enquanto existir caminho aumentante, aumentar o fluxo f usando o caminho aumentante

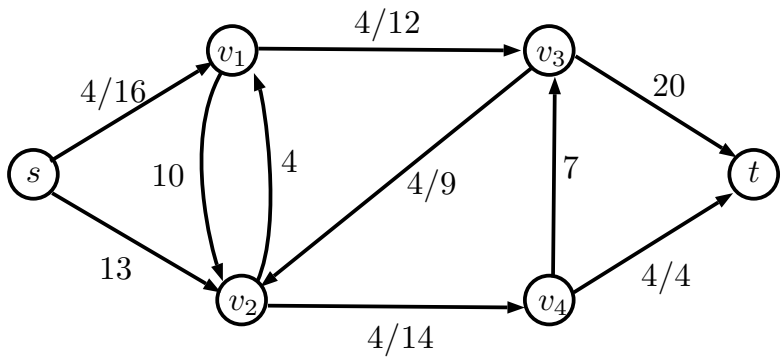
Fluxo em Redes - algoritmo de Ford-Fulkerson

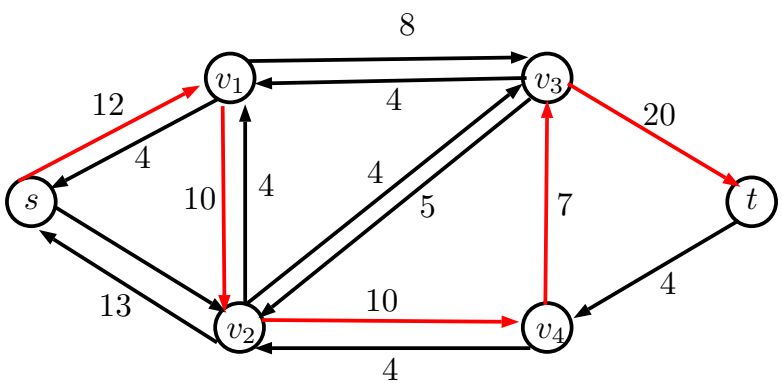
```
 1: procedure FordFulkerson(G, s, t)
 2:     ▷ entrada: grafo G, fonte s e ralo t
 3:     ▷ saída: fluxo máximo
 4:     for each (u, v) ∈ E do
 5:         f[u, v] ← 0
 6:         f[v, u] ← 0
 7:     end for
 8:     while ∃ caminho aumentante p na rede residual G_f do
 9:         calcular c_f(p)
10:         for each (u, v) ∈ E do
11:             f[u, v] ← f[u, v] + c_f(p)
12:             f[v, u] = −f[u, v]
13:         end for
14:     end while
15: end procedure
```

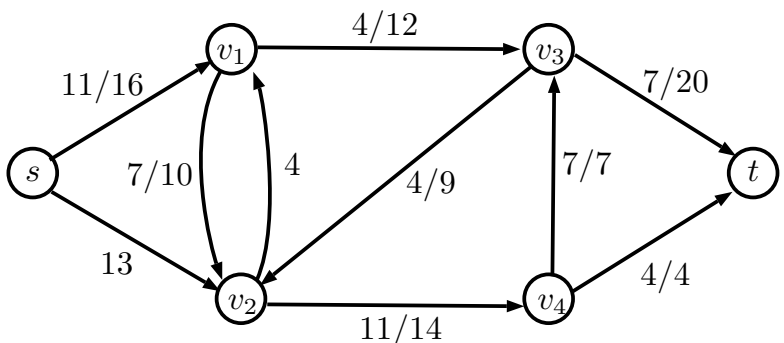
Fluxo em Redes - algoritmo de Ford-Fulkerson

Sequência de execução do algoritmo mostrando a rede residual e o fluxo resultante do caminho aumentante:



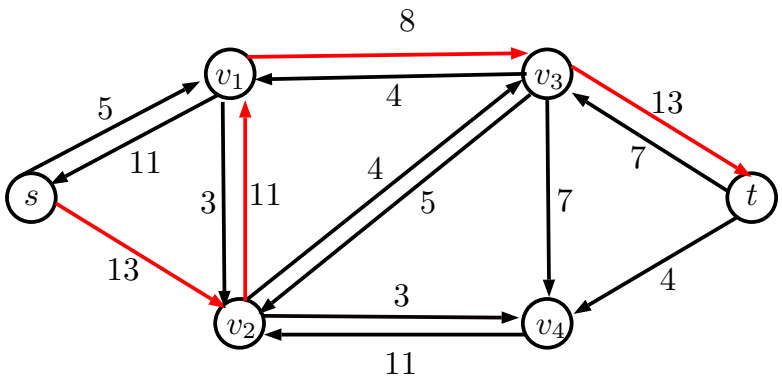


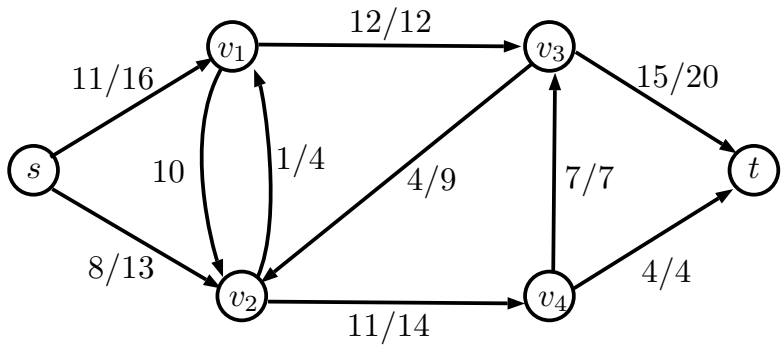


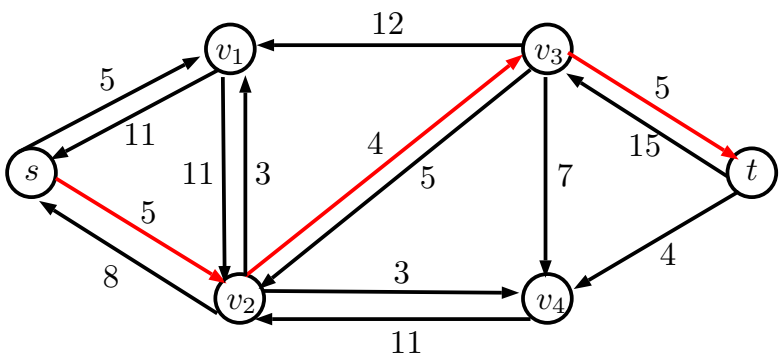


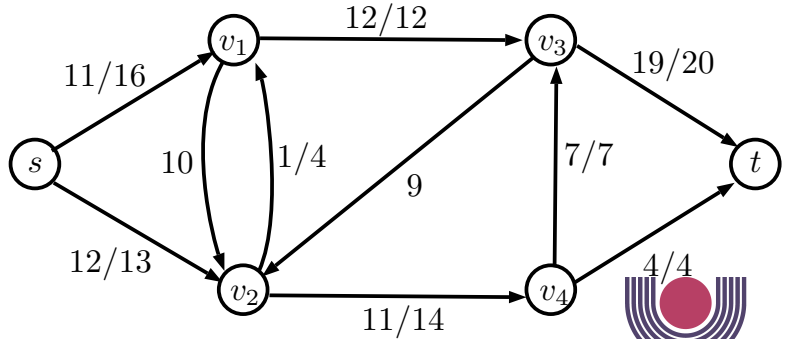
Fluxo em Redes - algoritmo de Ford-Fulkerson

Sequência de execução do algoritmo mostrando a rede residual e o fluxo resultante do caminho aumentante:



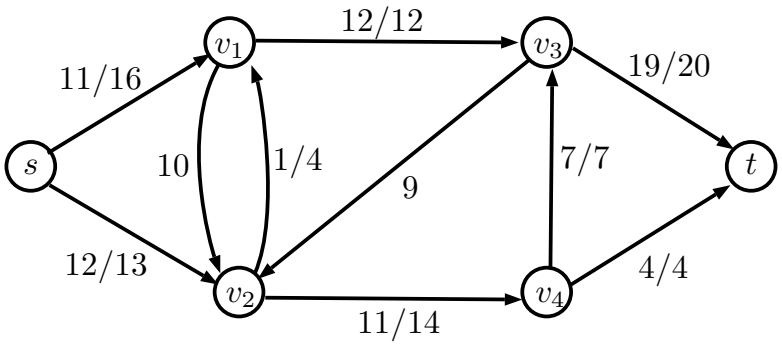


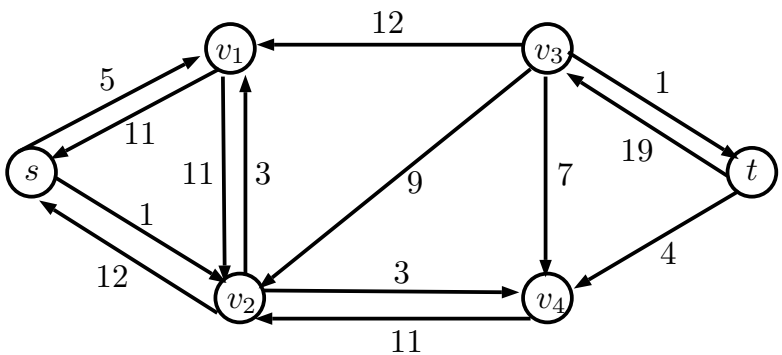




Fluxo em Redes - algoritmo de Ford-Fulkerson

Não há mais caminho aumentante na residual, logo o fluxo já é máximo $f(s,t) = 23$.



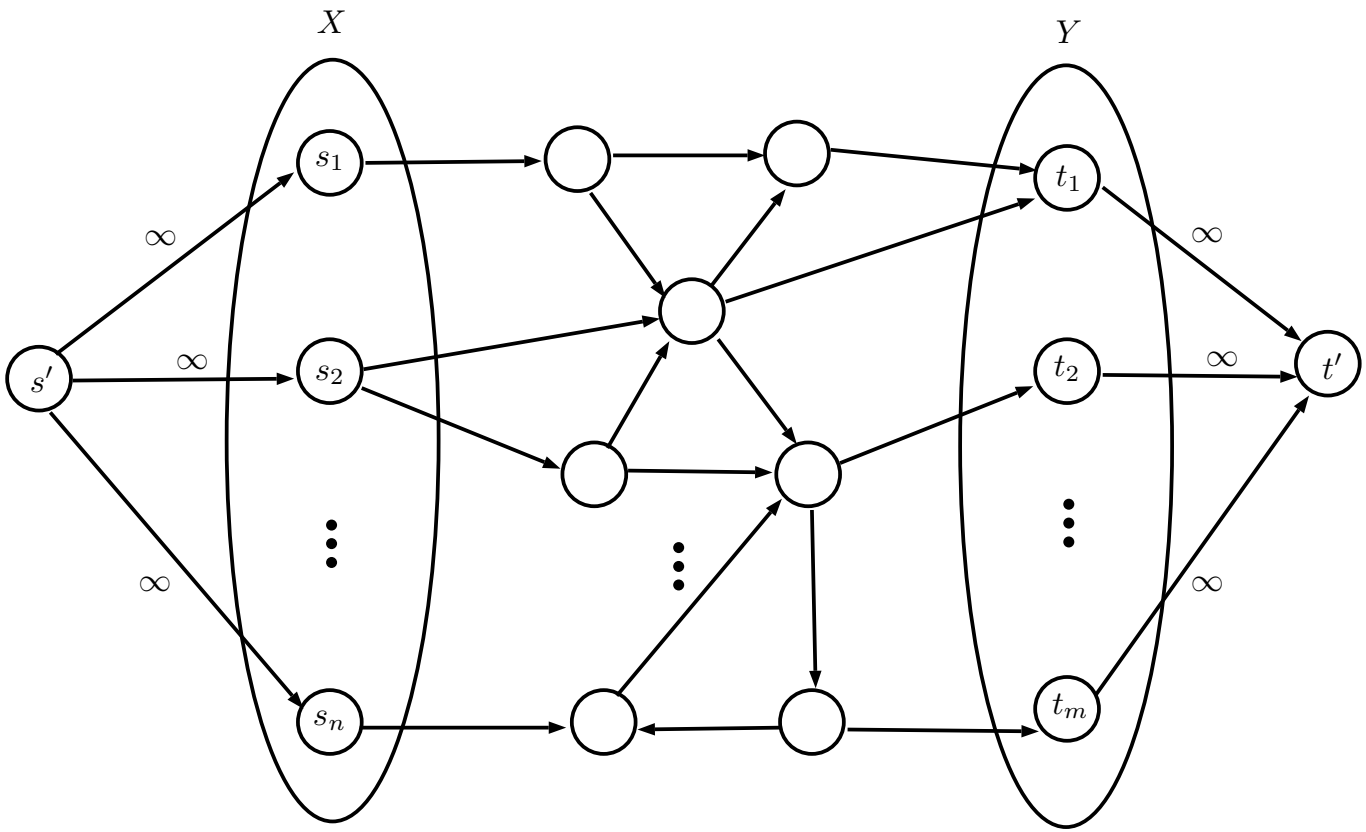


Fluxo em Redes - algoritmo de Ford-Fulkerson

- capacidades devem ser inteiras
- se capacidades forem racionais, é possível fazer uma transformação de escala para utilizar somente inteiros, aplicar o algoritmo e depois transformar o resultado para a escala anterior
- se existirem capacidades irracionais, o algoritmo pode não terminar, pois pode não convergir para um fluxo máximo
- complexidade: $O(E|f^*|)$, onde f^* é o fluxo máximo

Fluxo em Redes - algoritmo de Ford-Fulkerson

- Redes com múltiplas fontes e múltiplos ralos:
  - seja X o conjunto das fontes e Y o conjunto dos ralos
  - criar superfonte (s') e superralo (t') com capacidade ∞ conforme esquema abaixo



Emparelhamento Máximo em Grafos Bipartidos

Emparelhamento em Grafos

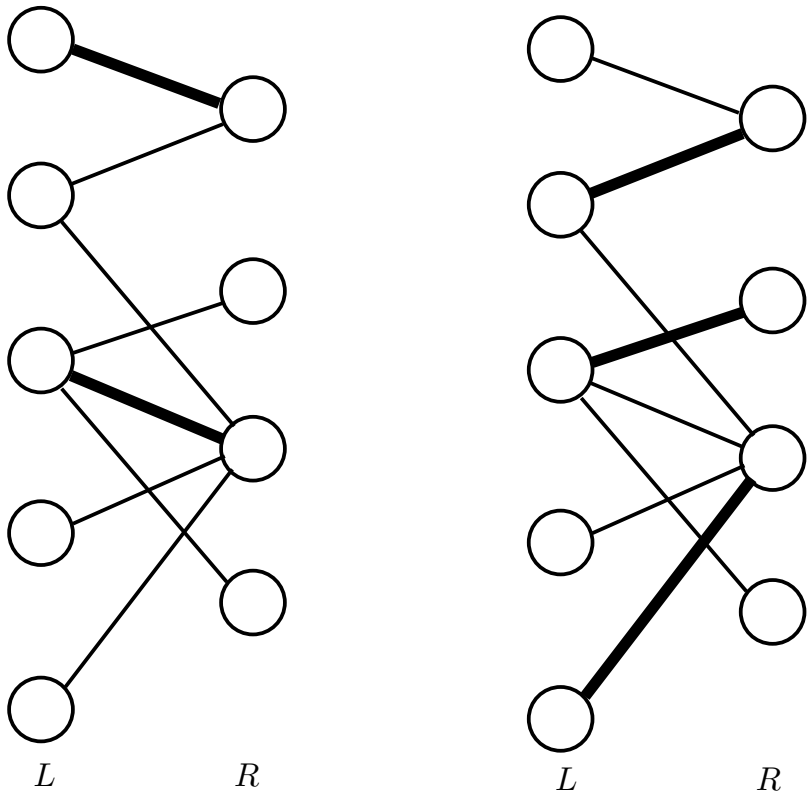
Dado um grafo $G = (V, E)$, um **emparelhamento** $M \in E$ é tal que quaisquer 2 arestas em M não são adjacentes.

Emparelhamento Máximo em Grafos Bipartidos

Dado um grafo $G = (V, E)$ bipartido, um emparelhamento $M \in E$ é máximo se qualquer outro emparelhamento M', tem-se que $|M'| \leq |M|$.

- pode ser reduzido ao problema do fluxo máximo em redes

Emparelhamento Máximo em Grafos Bipartidos

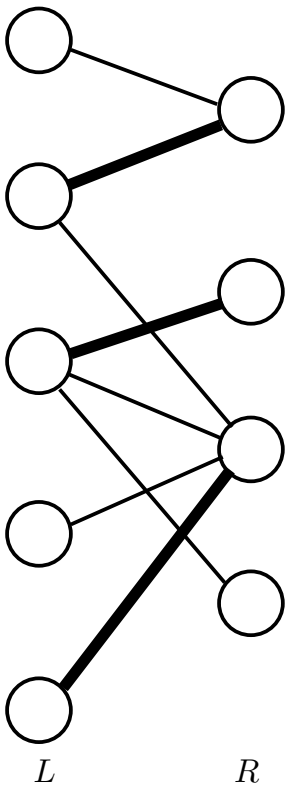


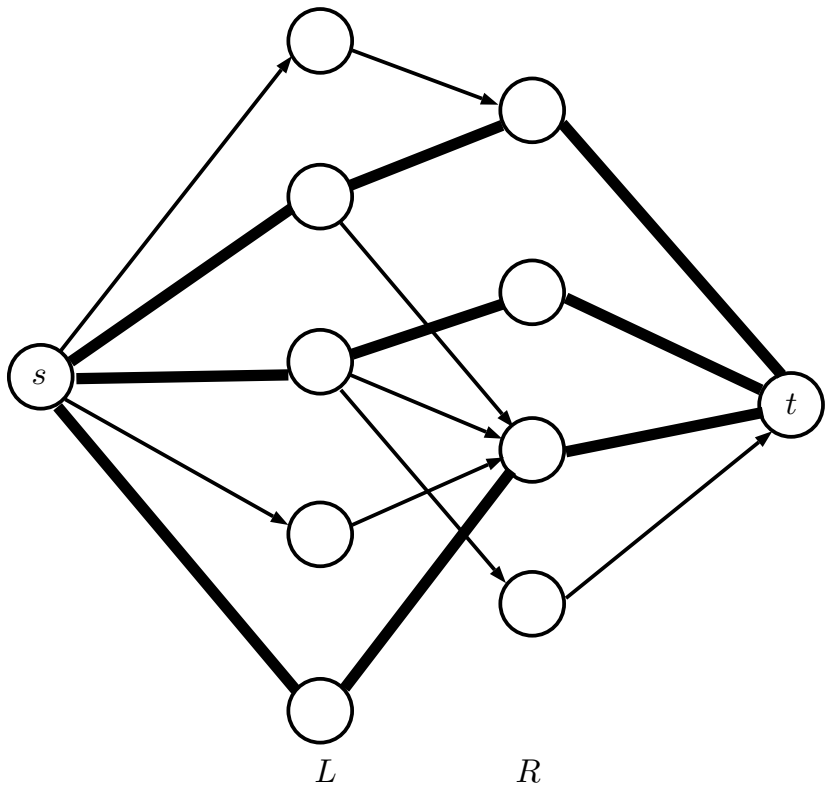
- (a): emparelhamento de tamanho 2
- (b): emparelhamento de tamanho 3 (máximo)

Emparelhamento Máximo em Grafos Bipartidos

- Seja $G = (V, E)$ um grafo bipartido com partições L e R:
  1. acrescentar vértice fonte s e criar arestas ligando s a todos os vértices de L
  2. acrescentar vértice falo t e criar arestas ligando todos os vértice R a t
  3. atribuir capacidade unitária a todas as arestas
  4. aplicar algoritmo de Ford-Fulkerson para obtenção do fluxo máximo, que irá definir o emparelhamento máximo de maneira subjacente

Emparelhamento Máximo em Grafos Bipartidos





Bibliografia I

[Bondy 1982] Bondy, J. A.; Murty, U.S.R.
Graph Theory with Applications. Elsivier, 1982.

[Netto 1996] P.O. Boaventura Netto.
Grafos: Teoria, Modelos, Algoritmos. Edgard Blucher, São Paulo, 1996.

[Diestel 1997] R. Diestel.
Graph Theory. Springer, New York, 1997.

[Cormen 1997] Cormen,T.; Leiserson, C.; Rivest, R.
Introduction to Algorithms. McGrawHill, New York, 1997.