

# Complexidade de Algoritmos

Prof. Rafael Alceste Berri

[rafaelberri@gmail.com](mailto:rafaelberri@gmail.com)

Prof. Diego Buchinger

[diego.buchinger@outlook.com](mailto:diego.buchinger@outlook.com)

Prof. Cristiano Damiani Vasconcellos

[cristiano.vasconcellos@udesc.br](mailto:cristiano.vasconcellos@udesc.br)

---

# Algoritmos de Grafos – Complexidade com múltiplas variáveis

---

Grafos possuem dois elementos principais:

- Vértices – Vertex (V)
- Arestas – Edges (E)

Grafos costumam ser representados de duas formas, que utilizam uma quantidade diferente de **espaço**:

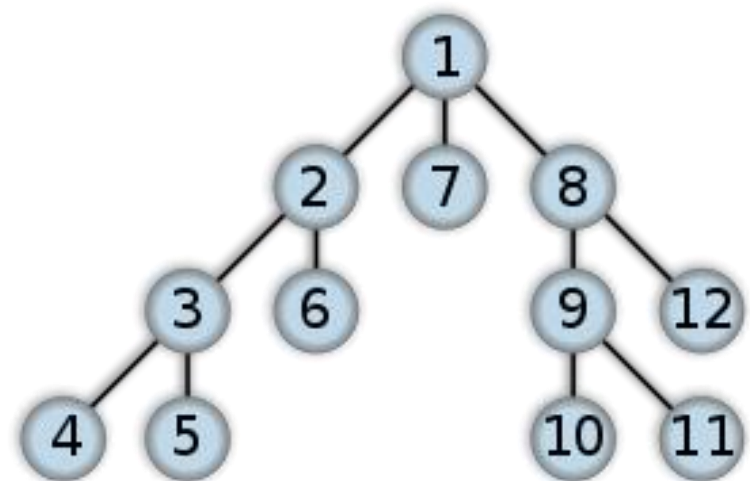
- Matriz de Adjacência  $\Theta( V^2 )$
- Lista de Adjacência  $\Theta( V+E )$

\* Considere apenas o espaço criado pelo algoritmo

# Árvores são grafos

---

- Toda árvore é um grafo, mas nem todo grafo é uma árvore!
- Portanto uma árvore pode ser representada como **Matriz de Adjacência** ou **Lista de Adjacência**.
- Algoritmos grafos operam em árvores

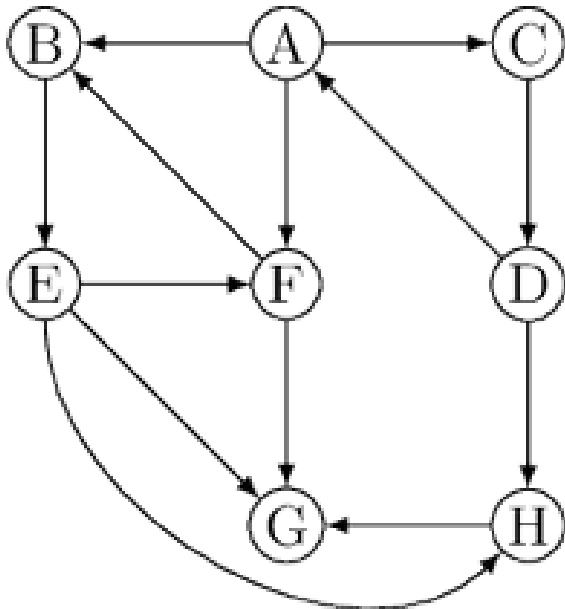


# Depth First-Search

(*Busca em Profundidade*)

---

O **algoritmo de Busca em Profundidade** realiza uma busca ou travessia em um grafo. É utilizado para verificar quais vértices são acessíveis iniciando um caminho a partir de um vértice específico



Quais vértices são acessíveis partindo de A?

Quais vértices são acessíveis partindo de E?

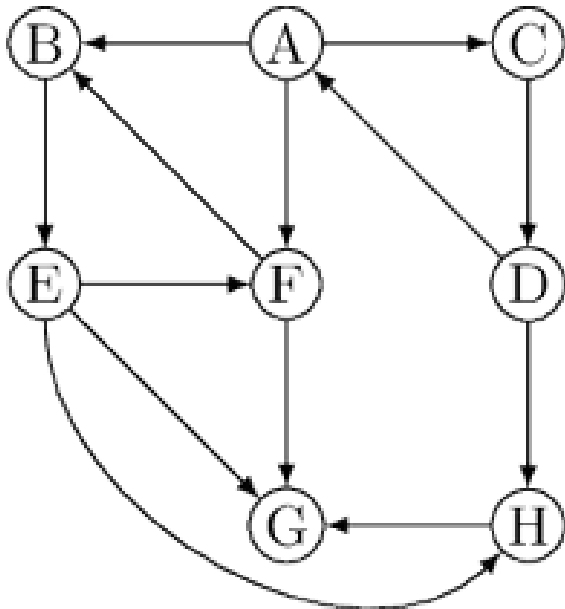
Árvores: Pré-ordem

# Depth First-Search

(*Busca em Profundidade*)

---

O **algoritmo de Busca em Profundidade** realiza uma busca ou travessia em um grafo. É utilizado para verificar quais vértices são acessíveis iniciando um caminho a partir de um vértice específico

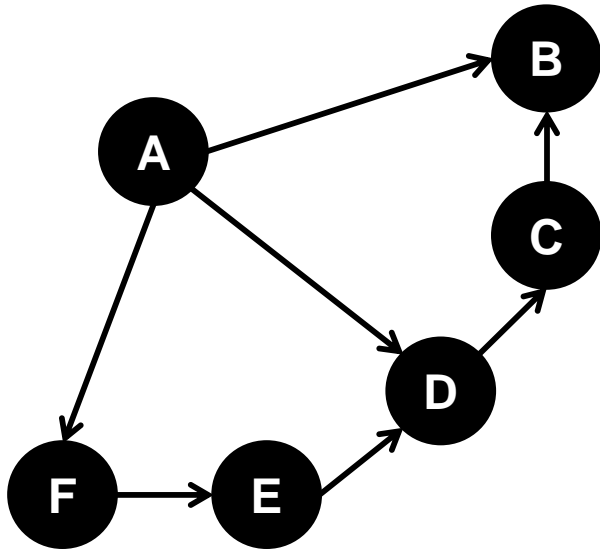


```
// G = grafo, s = vértice inicial,  
// v = vetor de visitação de vértices  
DFS( G, s, v )  
    v[s] = true  
    para cada u ∈ G.adj[s]  
        if( v[u] == false )  
            DFS( G, u, v )
```

# Depth First-Search

(*Busca em Profundidade*)

---



**Complexidade de tempo:**  
**Matriz**

**Melhor caso:**

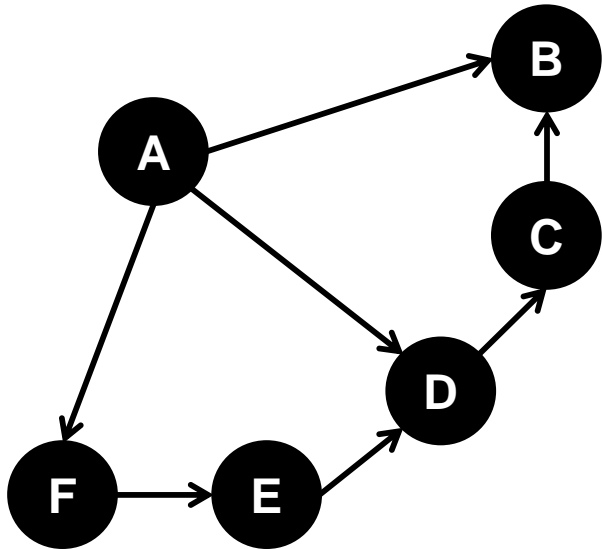
**Pior caso:**

**Complexidade de espaço:**

# Depth First-Search

(*Busca em Profundidade*)

---



## Complexidade de tempo:

### Matriz

**Melhor caso:**  $\Omega ( V )$

Nenhum nó está acessível a partir de um determinado nó

**Pior caso:**  $O ( V^2 )$

Todos os outros nós estão acessíveis a partir de um nó

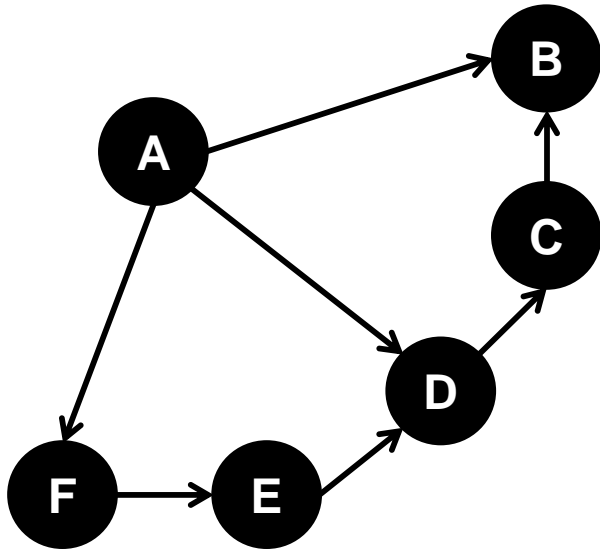
**Complexidade de espaço:** precisamos manter o vetor de nós visitados  $\Theta ( V )$



# Depth First-Search

(*Busca em Profundidade*)

---



## Complexidade de tempo:

**Lista**

**Melhor caso:**  $\Omega ( V )$

Necessidade de inicializar  
vértices como não visitados

**Pior caso:**  $O ( V + E )$

Todos os outros nós estão  
acessíveis a partir de um nó

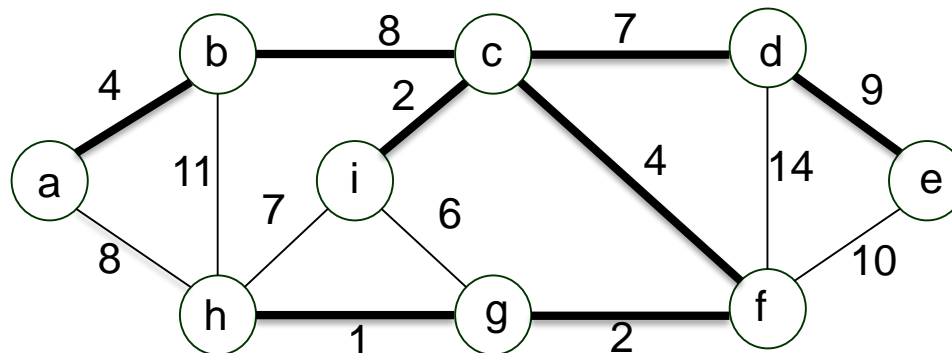
**Complexidade de espaço:** precisamos manter o vetor de  
nós visitados  $\Theta ( V )$

# Prim

(Árvore Geradora Mínima)

---

Uma **árvore geradora** para um grafo conexo é uma árvore que conecta todos os vértices do grafo e que o conjunto de arestas é um subconjunto das arestas desse grafo. A árvore geradora é mínima se o somatório dos custos associados cada arestas for o menor possível.



# Prim

(Árvore Geradora Mínima)

// G = grafo, s = vértice inicial, w = vetor de custo

**MST-PRIM( G, s, w )**

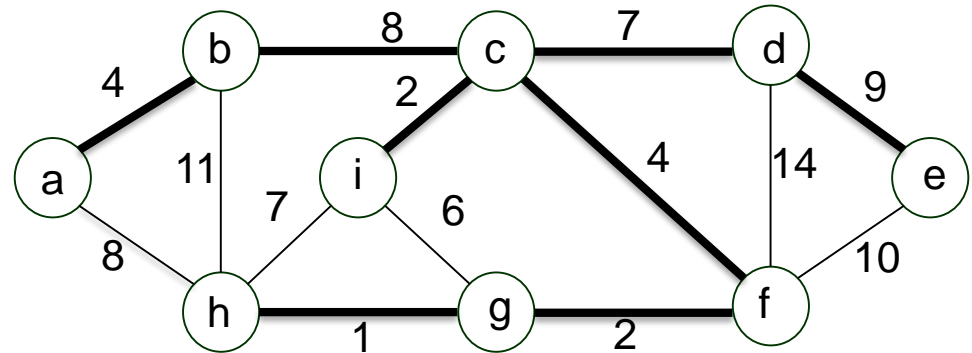
para cada u  $\in$  G.V

w[u] =  $\infty$

u.pai = null

w[s] = 0

q.add( {0,s} )



enquanto q  $\neq \emptyset$

u = extrai-minimo( q )

para cada z  $\in$  G.adj[u]

se z.custo < w[z.dest]

q.add( {z.custo, z.dest} ) // Qual a estrutura??

w[z.dest] = z.custo

z.dest.pai = u

// |V| vezes

// Qual a estrutura usada??

// Qual a estrutura usada??

# Prim

(Árvore Geradora Mínima)

---

// G = grafo, s = vértice inicial, w = vetor de custo

MST-PRIM( G, s, w )

para cada u  $\in$  G.V

w[u] =  $\infty$

u.pai = null

w[s] = 0

q.add( {0,s} )

**Usando G=>Matriz e q=>Lista**

Tempo:

$O(|V|^2 + |V|^2)$

$O(|V|^2)$

enquanto q  $\neq \emptyset$

// |V| vezes

u = extraí-minimo( q )

// |V| verificações

para cada z  $\in$  G.adj[u]

// |V| vezes

se z.custo < w[z.dest]

q.add( {z.custo, z.dest} ) // O(1)

w[z.dest] = z.custo

z.dest.pai = u

# Prim

(Árvore Geradora Mínima)

---

// G = grafo, s = vértice inicial, w = vetor de custo

MST-PRIM( G, s, w )

para cada u  $\in$  G.V

w[u] =  $\infty$

u.pai = null

w[s] = 0

q.add( {0,s} )

**Usando G=>Lista e q=>Lista**

Tempo:

$O(|V|^2 + |E|)$

\* lembrando que  $E = O(V^2)$

enquanto q  $\neq \emptyset$

// \* {  $O(V)$  }

u = extrai-minimo( q )

// \*  $\Rightarrow O(V^2)$  verificações

para cada z  $\in$  G.adj[u]

// \*  $\Rightarrow O(E)$

se z.custo < w[z.dest]

q.add( {z.custo, z.dest} ) //  $O(1)$

w[z.dest] = z.custo

z.dest.pai = u

# Prim

(Árvore Geradora Mínima)

---

// G = grafo, s = vértice inicial, w = vetor de custo

**MST-PRIM( G, s, w )**

  para cada u  $\in$  G.V

    w[u] =  $\infty$

    u.pai = null

  w[s] = 0

  q.add( {0,s} )

**Usando G=>Lista e q=>Árvore**

Tempo:

$O( |V \log V + E \log V| )$

$O( (V+E) \log V )$

**enquanto** q  $\neq \emptyset$

// \* {  $O(V)$  }

  u = extrai-minimo( q )

// \*  $\Rightarrow O( V \log V )$

  para cada z  $\in$  G.adj[u]

// \*  $\Rightarrow O( E )$

**se** z.custo < w[z.dest]

      q.add( {z.custo, z.dest} ) // (log V) [Se não  
      w[z.dest] = z.custo                   manter duplicatas]

      z.dest.pai = u

# Prim

(Árvore Geradora Mínima)

---

// G = grafo, s = vértice inicial, w = vetor de custo

MST-PRIM( G, s, w )

para cada u  $\in$  G.V

w[u] =  $\infty$

u.pai = null

w[s] = 0

q.add( {0,s} )

**Usando G=>Lista e q=>Fib. Heap**

Tempo:

$O( E + V \log V )$

enquanto q  $\neq \emptyset$

// \* {  $O(V)$  }

u = extrai-minimo( q )

// \*  $\Rightarrow O( V \log V )$

para cada z  $\in$  G.adj[u]

// \*  $\Rightarrow O( E )$

se z.custo < w[z.dest]

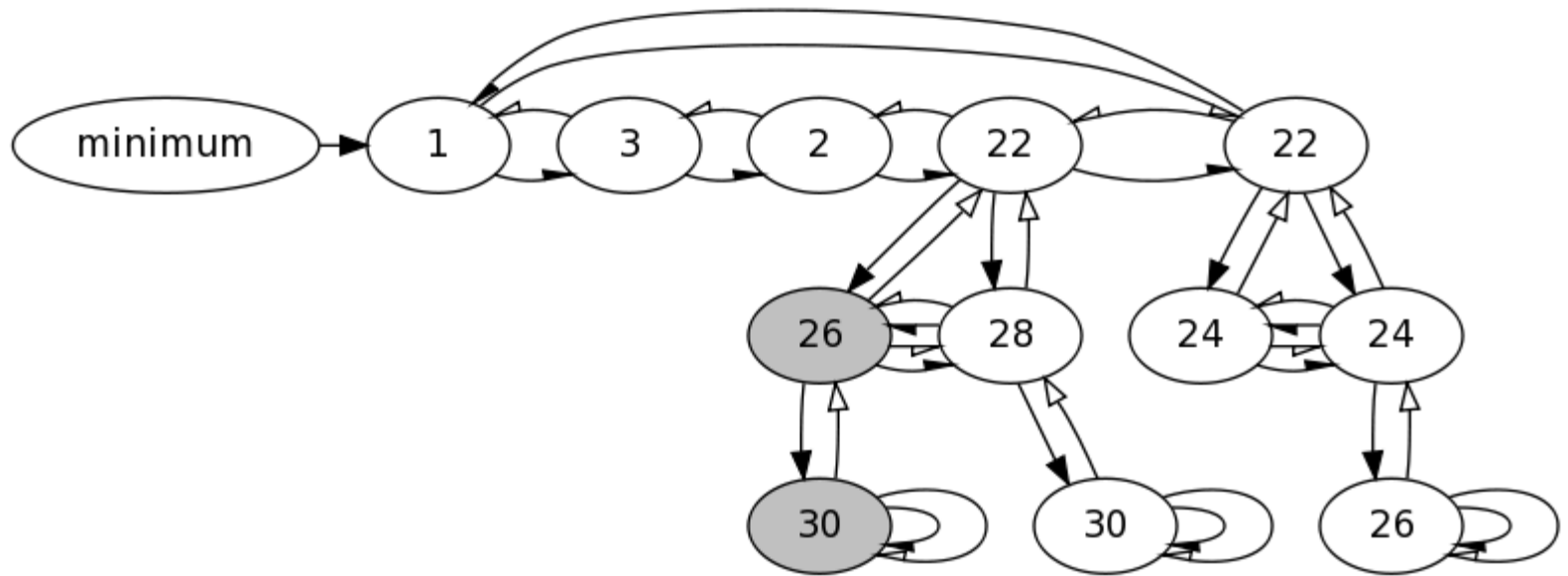
q.add( {z.custo, z.dest} ) // (1)

w[z.dest] = z.custo

z.dest.pai = u

# Fibonacci Heap

Exemplo:





# Fibonacci Heap

- Tabela comparativa para fila de prioridade:

Operações	Lista Enc.	Heaps	
		Binária	Fibonacci
make-heap	1	1	1
insert	1	$\log N$	1
find-min	N	1	1
delete-min	N	$\log N$	$\log N$
union	1	N	1
decrease-key	1	$\log N$	1
delete	N	$\log N$	$\log N$
is-empty	1	1	1

## Atividade Extra

---

Crie um grafo com no Mínimo 5 vértices e 5 arestas (com pesos distintos). Execute o algoritmo de PRIM utilizando Lista e Fibonacci (ver exemplo de implementação de Fibonacci Heap em fibonacci-heap.zip). Como resposta, envie um esboço do seu grafo, a árvore geradora mínima e o código fonte utilizado para gerar a árvore.

Envie os exercícios para [rafaelberri@gmail.com](mailto:rafaelberri@gmail.com) Assunto: “TC-CALEX”  
Anexar: resultados/código fonte

# Dijkstra

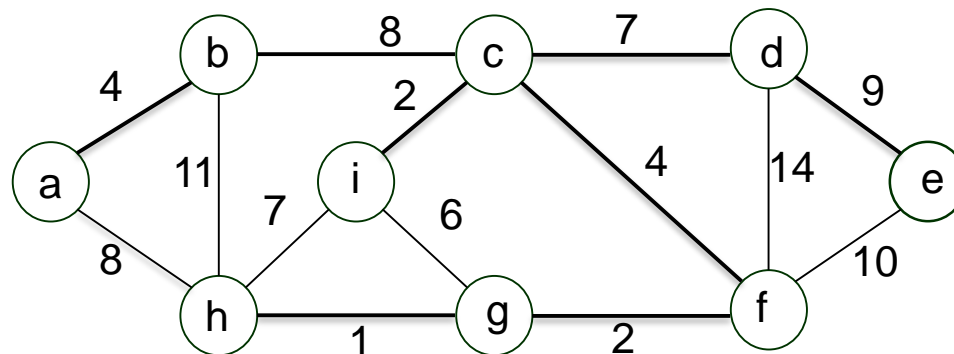
*(menor caminho entre dois nós)*

---

Dado um grafo ponderado – sem arestas com peso negativo – e dois nós, o algoritmo de Dijkstra encontra o menor caminho entre estes dois nós.

Muito semelhante ao algoritmo de Prim

Menor Caminho entre b | g



# Dijkstra

*(menor caminho entre dois nós)*

---

// G = grafo, s = vértice inicial, w = vetor de custo

**DIJKSTRA**( G, s, w )

  para cada u  $\in$  G.V

    w[u] =  $\infty$

    u.pai = null

  w[s] = 0

  q.add( {0,s} )

**Usando G=>Lista e q=>Fib. Heap**

Tempo:

$O( E + V \log V )$

**enquanto** q  $\neq \emptyset$

// \* {  $O(V)$  }

  u = extrai-minimo( q )

// \*  $\Rightarrow O( V \log V )$

  para cada z  $\in$  G.adj[u]

// \*  $\Rightarrow O( E )$

  se w[u] + z.custo < w[z.dest]

    q.add( {w[u] + z.custo, z.dest} ) // (1)

    w[z.dest] = w[u] + z.custo

    z.dest.pai = u

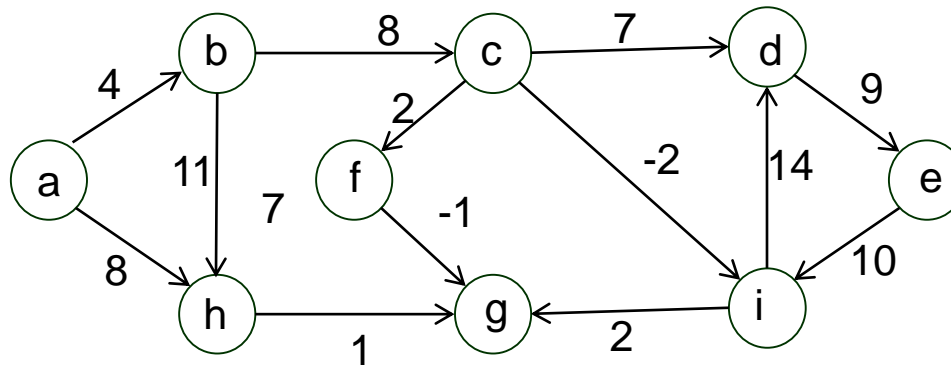
# Bellman Ford

(*menor caminho entre dois nós*)

---

Dado um grafo ponderado – possivelmente com arestas de peso negativo, mas sem ciclo – e dois nós, o algoritmo de Belmann Ford encontra o menor caminho entre os dois nós.

Menor Caminho entre b | g



# Bellman Ford

*(menor caminho entre dois nós)*

---

// G = grafo, s = vértice inicial, w = vetor de custo

**BELLMAN( G, s, w )**

para cada u  $\in$  G.V

w[u] =  $\infty$

u.pai = null

w[s] = 0;

**Usando G=>Lista**

Tempo:

$O( V + V \cdot E )$

$O( VE )$

para cada u  $\in$  G.V:

para cada a  $\in$  G.adj[u]:

se ( w[u] + a.custo < w[a.dest] )

w[a.dest] = w[u] + a.custo

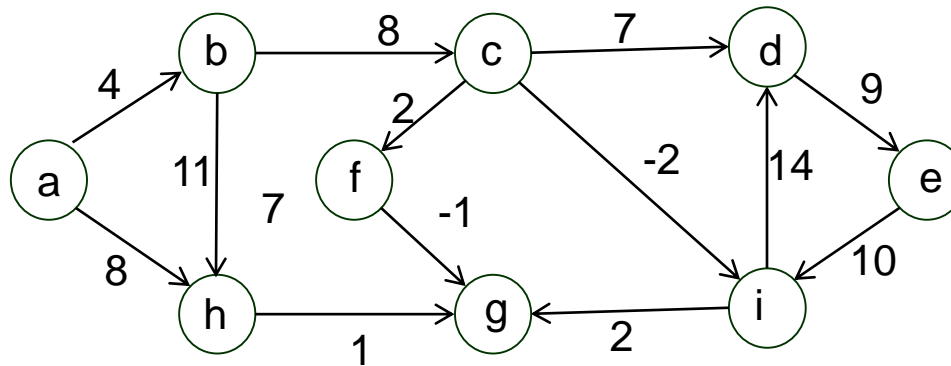
a.dest.pai = u

# Floyd Warshall

*(menor caminho entre todos os nós)*

---

Dado um grafo ponderado – possivelmente com arestas de peso negativo, mas sem ciclo – o algoritmo de Floyd Warshall encontra o menor caminho entre todos os nós (de todos para todos).



# Floyd Warshall

*(menor caminho entre todos os nós)*

---

```
// G = grafo
FLOYD( G )
  dist[G.v][G.v]
  para cada u ∈ G.V
    dist[u][u] = 0
  para cada z(u,v) ∈ G.E
    dist[u][v] = z.custo
```

```
para k de 0 a G.v:
  para i de 0 a G.v:
    para j de 0 a G.v:
      se ( dist[i][j] > dist[i][k] + dist[k][j] )
        dist[i][j] = dist[i][k] + dist[k][j]
```

**Usando  $G \Rightarrow$  Lista / Matriz**

Tempo:  
 $\Theta ( V^3 )$

Uso de programação dinâmica