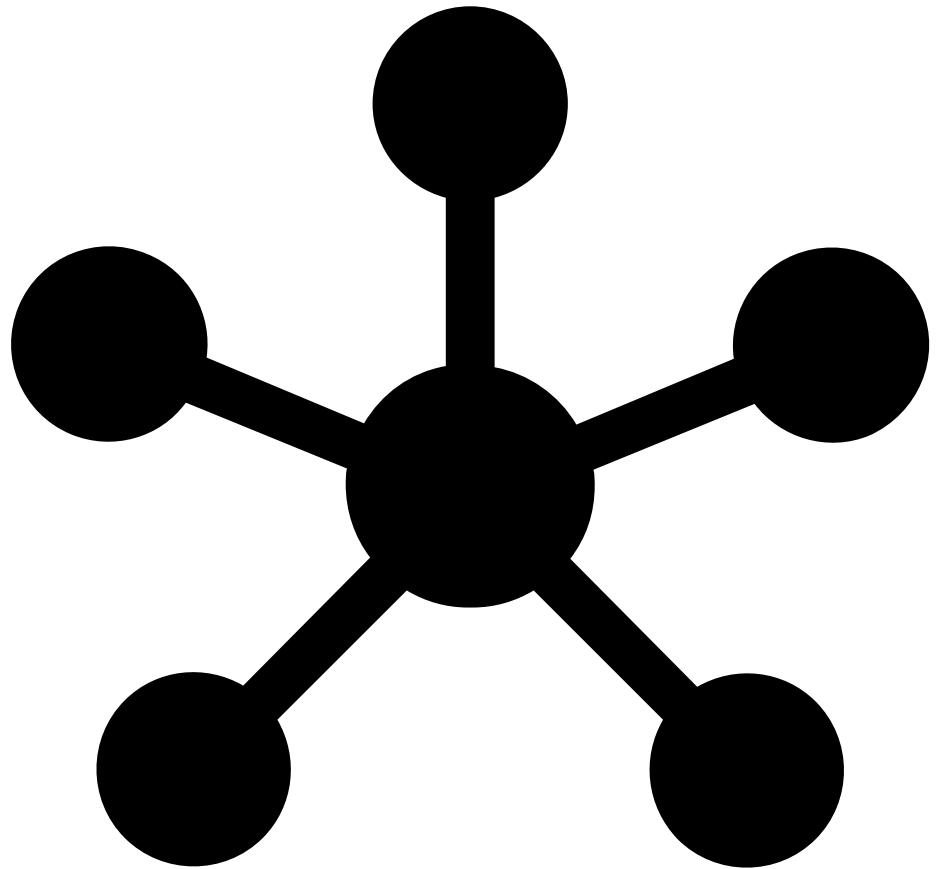


# Teoria dos Grafos

---

GRAFOS E ALGORITMOS



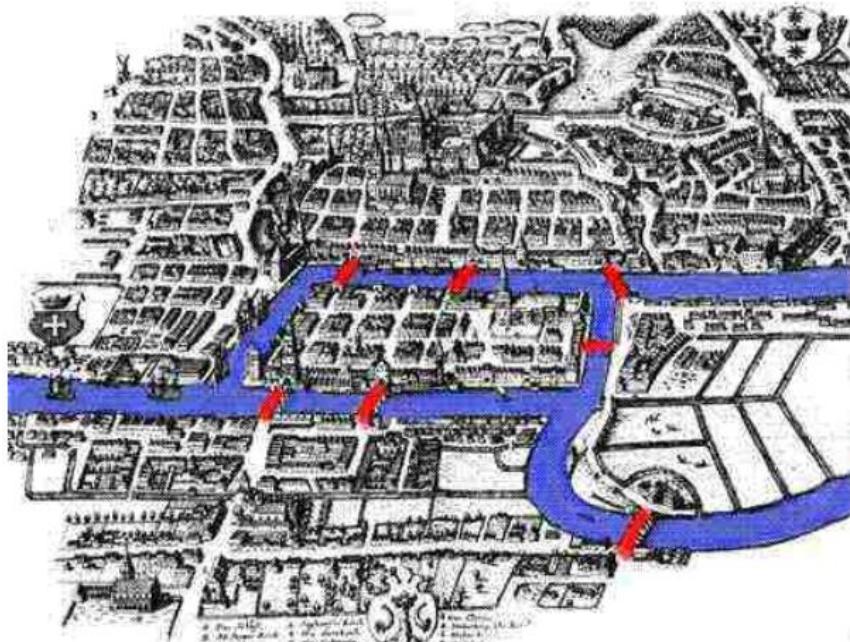
# Parte 1

CONCEITOS E DEFINIÇÕES

# O problema das sete pontes de Königsberg

---

A cidade de Königsberg foi construída numa região onde haviam dois braços do Rio Pregel e uma ilha. Foram construídas sete pontes ligando diferentes partes da cidade, como mostrado na figura:



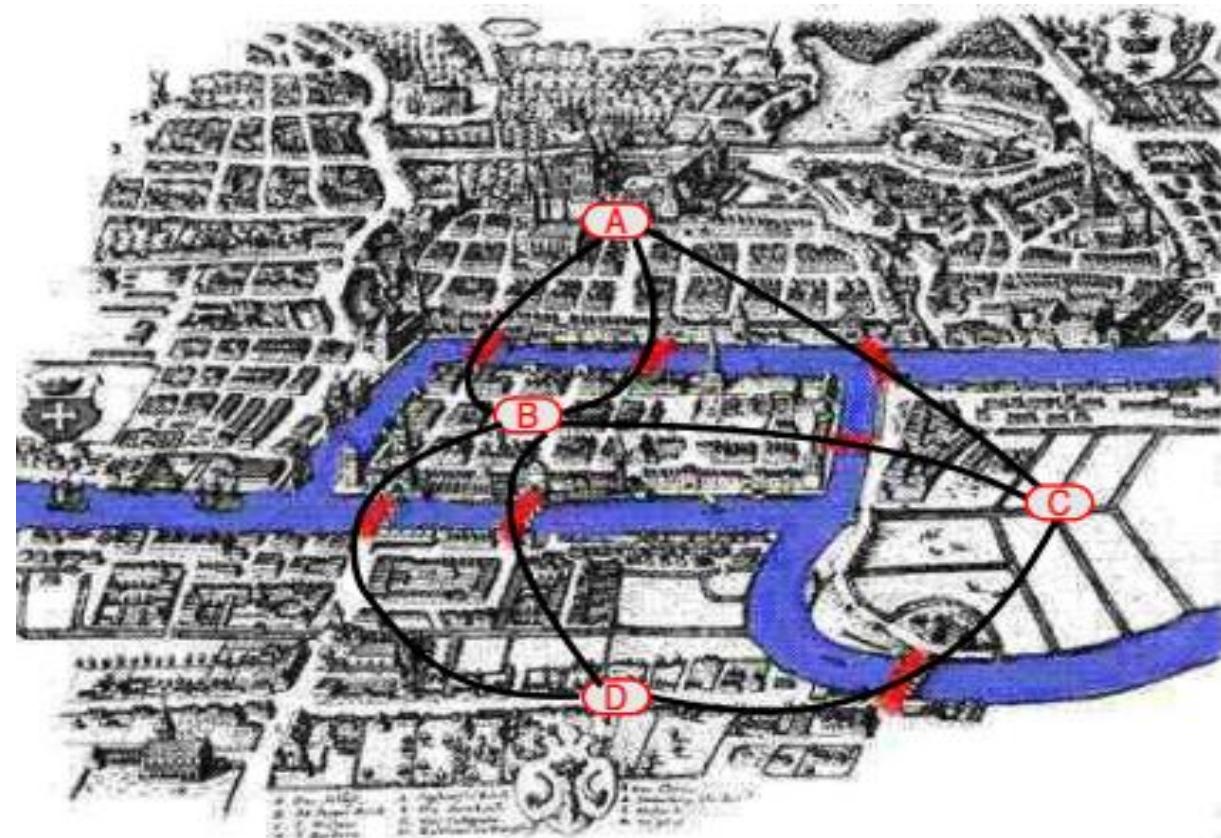
**Problema:** É possível que uma pessoa faça um percurso na cidade de tal forma que inicie e volte a mesma posição passando por todas as pontes somente uma única vez?

# O problema das sete pontes de Königsberg

Euler resolveu este problema dando início à teoria dos grafos.

Modelagem proposta por Euler:

- Todos os “pontos” de uma dada área de terra podem ser representados por um único ponto já que uma pessoa pode andar de um lado para o outro sem atravessar uma ponte.
- Um ponto é conectado a outro se houver uma ponte de um lado para o outro.
- Graficamente, Euler representou o problema como:

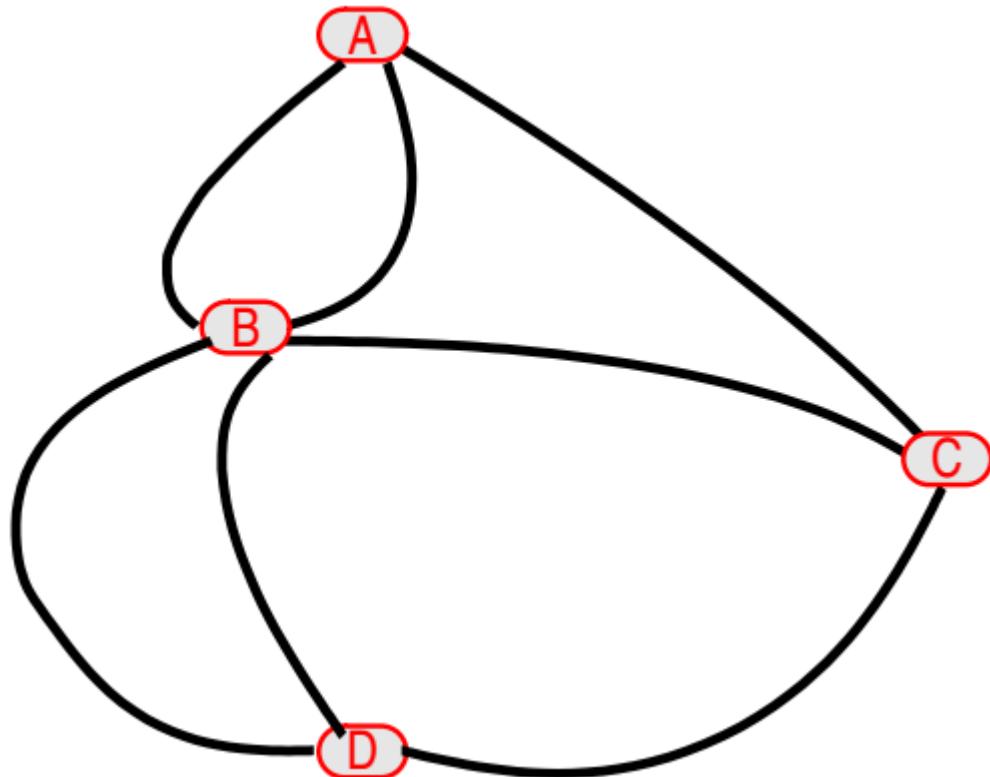


# O problema das sete pontes de Königsberg

---

Problema a ser resolvido:

- É possível achar um caminho que comece e termine num vértice qualquer ( $A$ ,  $B$ ,  $C$ , ou  $D$ ) e passe por cada aresta, exatamente, e uma única vez?, ou ainda,
- É possível desenhar este grafo que comece e termine na mesma posição sem levantar o lápis do papel?



# O problema das sete pontes de Königsberg

---

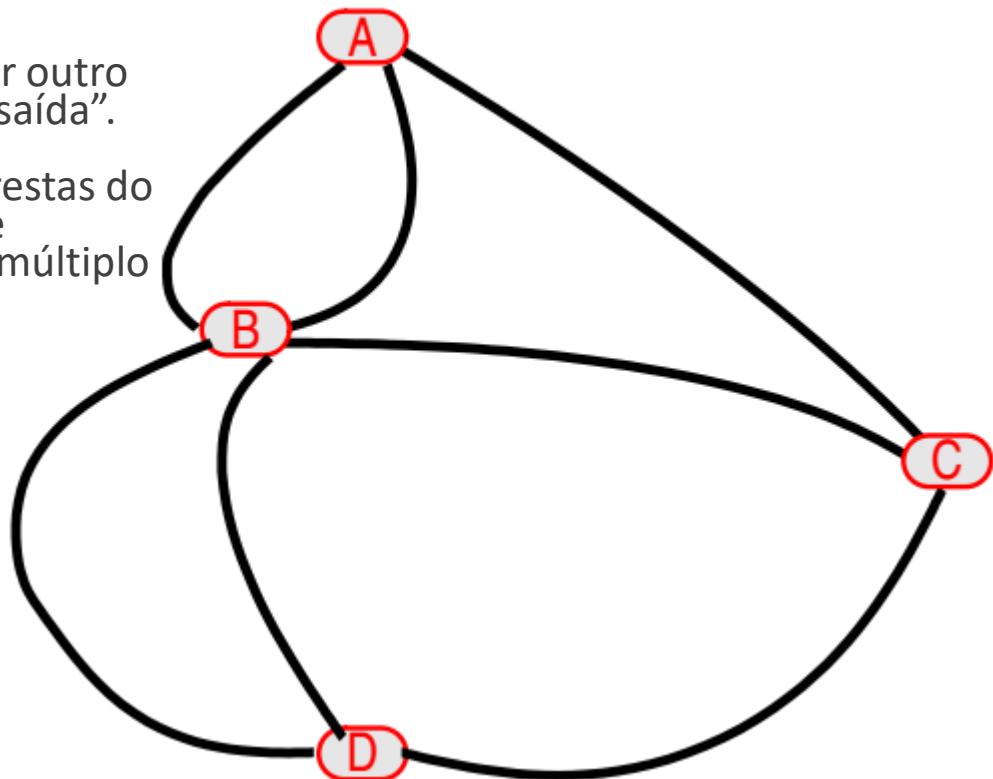
Aparentemente não existe solução!

Partindo do vértice A, toda vez que se passa por qualquer outro vértice, duas arestas são usadas: a de “chegada” e a de “saída”.

Assim, se for possível achar uma rota que usa todas as arestas do grafo e começa e termina em A, então o número total de “chegadas” e “saídas” de cada vértice deve ser um valor múltiplo de 2.

- No entanto, temos:
  - $\text{grau}(A) = \text{grau}(C) = \text{grau}(D) = 3$ ; e
  - $\text{grau}(B) = 5$ .

Assim, por este raciocínio informal não é possível ter uma solução para este problema.



# Caminhamentos em grafos - Caminho

Seja  $G$  um grafo não dirigido,  $n \geq 1$ , e  $v$  e  $w$  vértices de  $G$ .

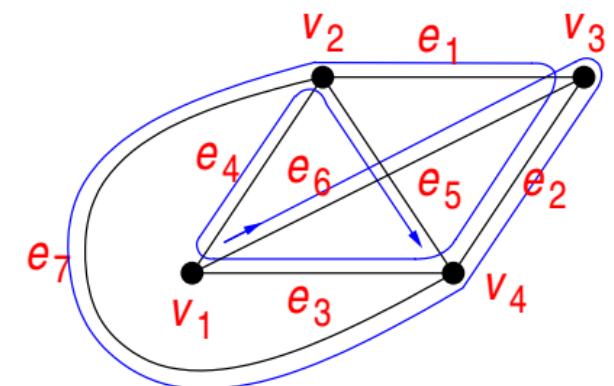
**Caminho (walk):** Um caminho de  $v$  para  $w$  é uma sequência alternada de vértices e arestas adjacentes de  $G$ . Um caminho tem a forma:

$$(v =) v_0 e_1 v_1 e_2 v_2 \dots v_{n-1} e_n v_n (= w)$$

ou ainda

$$v_0[v_0, v_1] v_1[v_1, v_2] v_2 \dots v_{n-1}[v_{n-1}, v_n] v_n$$

onde  $v_0 = v$  e  $v_n = w$ .



Um possível caminho entre  $v_1$  e  $v_4$ :  
 $v_1 e_6 v_3 e_2 v_4 e_7 v_2 e_1 v_3 e_2 v_4 e_3 v_1 e_4 v_2 e_5 v_4$

# Caminhamentos em grafos - Caminho

---

No caso de arestas múltiplas, deve-se indicar qual delas está sendo usada.

Vértices  $v_0$  e  $v_n$  são extremidades do caminho.

Tamanho (comprimento) do caminho: número de arestas do mesmo, ou seja, número de vértices menos um

O **caminho trivial** de  $v$  para  $v$  consiste apenas do vértice  $v$ .

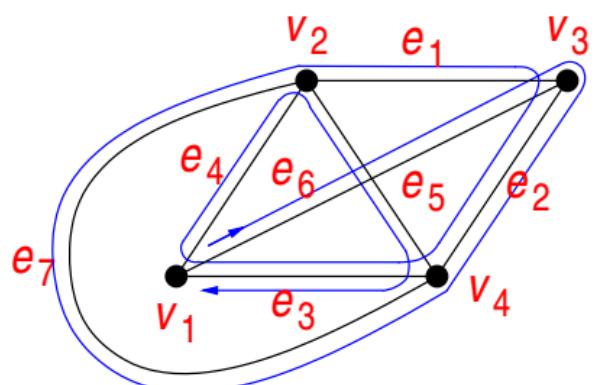
Se existir um caminho  $c$  de  $v$  para  $w$  então  $w$  é alcançável a partir de  $v$  via  $c$

# Caminhamentos em grafos - Caminho

**Caminho fechado (*Closed walk*):** Caminho que começa e termina no mesmo vértice:

$$(v =) v_0 e_1 v_1 e_2 v_3 \dots v_{n-1} e_n v_n (= w)$$

onde  $v = w$ .



Um possível caminho fechado é:

$$v_1 e_6 v_3 e_2 v_4 e_7 v_2 e_1 v_3 e_2 v_4 e_3 v_1 e_4 v_2 e_5 v_4 e_3 v_1$$

Um caminho fechado com pelo menos uma aresta é chamado de **ciclo**

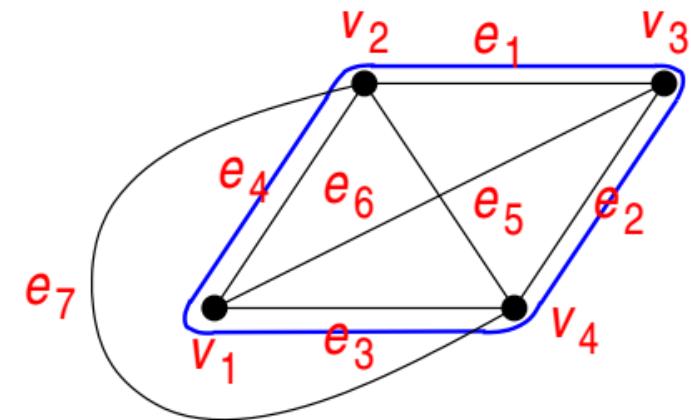
# Caminhamentos em grafos - Caminho

Dois caminhos fechados

$$v_0 v_1 \dots v_n \quad \text{e} \quad v'_0 v'_1 \dots v'_n$$

formam o mesmo ciclo se existir um inteiro  $j$  tal que

$$v'_i = v_{i+j} \bmod n, \text{ para } i = 0, 1, \dots, n-1.$$



O caminho fechado  $v_1 v_2 v_3 v_4 v_1$  forma o mesmo ciclo que os caminhos fechados  $v_2 v_3 v_4 v_1 v_2$ ,  $v_3 v_4 v_1 v_2 v_3$  e  $v_4 v_1 v_2 v_3 v_4$ .

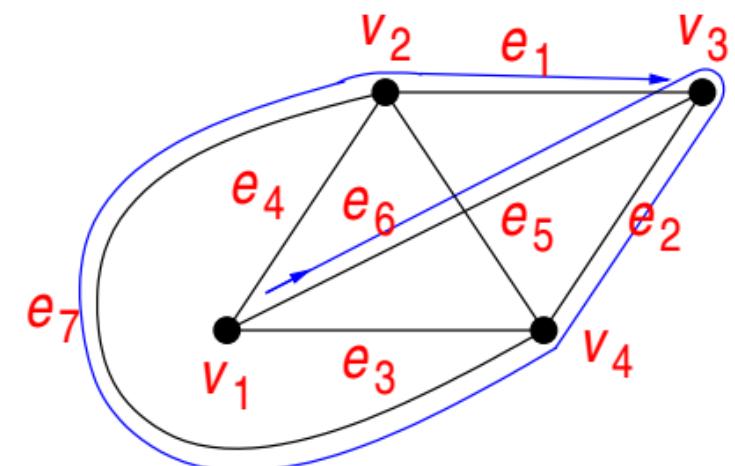
# Caminhamentos em grafos - Caminho

**Trajeto (*Path*):** Caminho de  $v$  para  $w$  sem arestas repetidas:

$$(v =) v_0 e_1 v_1 e_2 v_3 \dots v_{n-1} e_n v_n (= w)$$

onde todas as arestas  $e_i$  são distintas, ou seja,  
 $e_i \neq e_k$ , para qualquer  $i \neq k$ .

Um possível trajeto é:  
 $v_1 e_6 v_3 e_2 v_4 e_7 v_2 e_1 v_3$



# Caminhamentos em grafos - Caminho

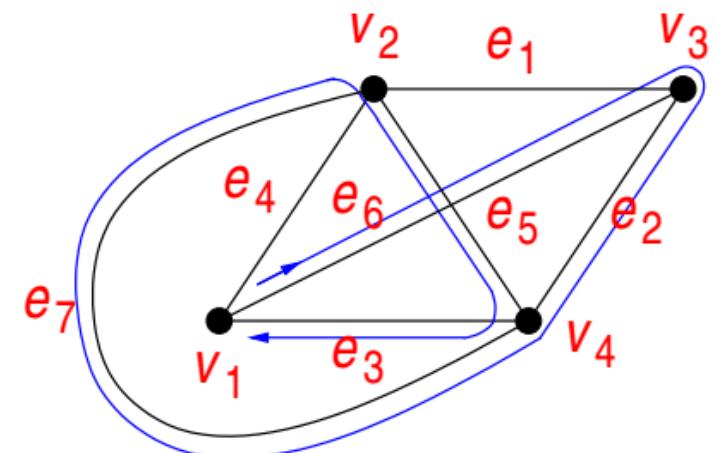
**Círculo (Circuit):** Trajeto fechado, ou seja, um caminho onde não há aresta repetida e os vértices inicial e final são idênticos:

$$(v =) v_0 e_1 v_1 e_2 v_3 \dots v_{n-1} e_n v_n (= w)$$

onde toda aresta  $e_i$ ,  $1 \leq i \leq n$ , é distinta e  $v_0 = v_n$ .

Um possível circuito é:

$$v_1 e_6 v_3 e_2 v_4 e_7 v_2 e_1 v_3$$

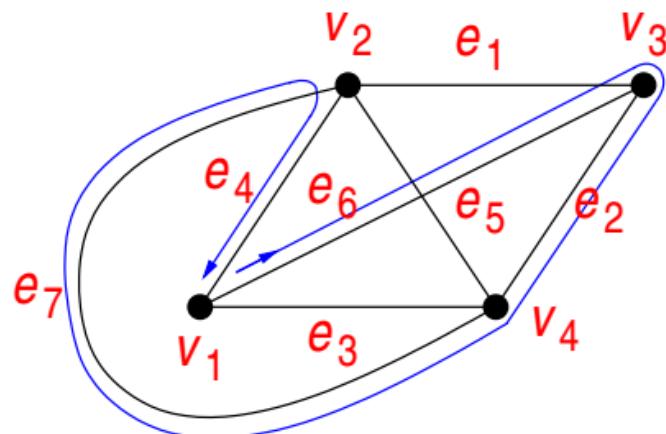


# Caminhamentos em grafos - Caminho

**Círculo simples (*Simple circuit*):** Trajeto fechado, ou seja, um caminho onde não há arestas e vértices repetidos, exceto os vértices inicial e final que são idênticos

Um possível circuito simples é:

$v_1e_6v_3e_2v_4e_7v_2e_4v_1$



Um circuito simples também é chamado de ciclo simples.

# Caminhamentos em grafos - Caminho

---

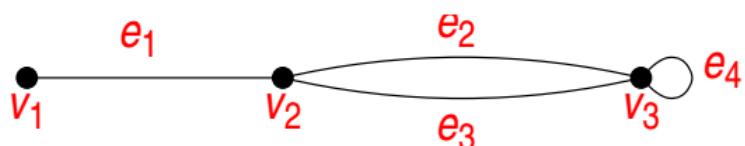
<b>Tipo</b>	<b>Aresta repetida?</b>	<b>Vértice repetido?</b>	<b>Começa e termina no mesmo vértice?</b>
Caminho ( <i>walk</i> )	Pode	Pode	Pode
Caminho fechado ( <i>closed walk</i> )	Pode	Pode	Sim
Trajeto ( <i>path</i> )	Não	Pode	Pode
Trajeto simples ( <i>simple path</i> )	Não	Não	Não
Círculo ( <i>circuit</i> )	Não	Pode	Sim
Círculo simples ( <i>simple circuit</i> )	Não	$v_0 = v_n$	Sim

# Caminhamentos em grafos - Caminho

---

## Notação simplificada

Em geral um caminho pode ser identificado de forma não ambígua através de uma sequência de arestas ou vértices.



O caminho  $e_1e_2e_4e_3$  representa de forma não ambígua o caminho  $v_1e_1v_2e_2v_3e_4v_3e_3v_2$

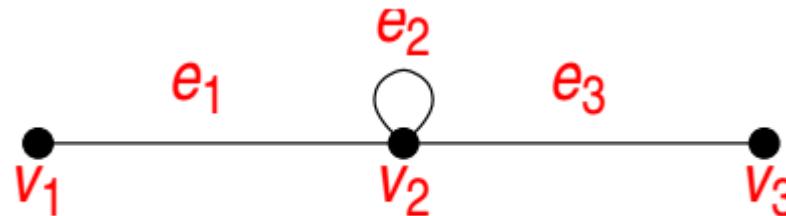
A notação  $v_2v_3$  é ambígua, se usada para referenciar um caminho, pois pode representar duas possibilidades:  $v_2e_2v_3$  ou  $v_2e_3v_3$

# Caminhamentos em grafos - Caminho

---

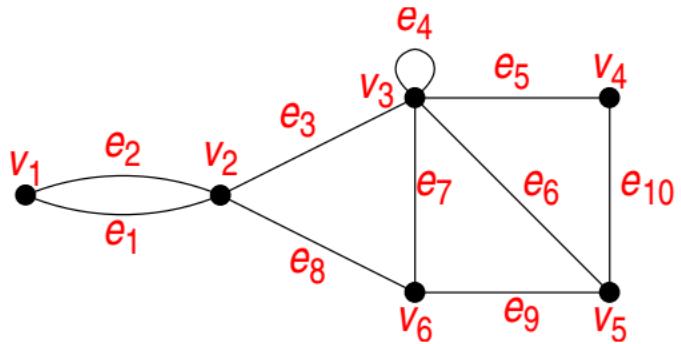
A notação  $v_1v_2v_2v_3$ , se for associada a um caminho, representa de forma não ambígua o caminho  $v_1e_1v_2e_2v_2e_3v_3$

Se um grafo  $G$  não possui arestas paralelas, então qualquer caminho em  $G$  pode ser determinado de forma única por uma sequência de vértices.



# Caminhamentos em grafos - Caminho

## Identificando o caminhamento



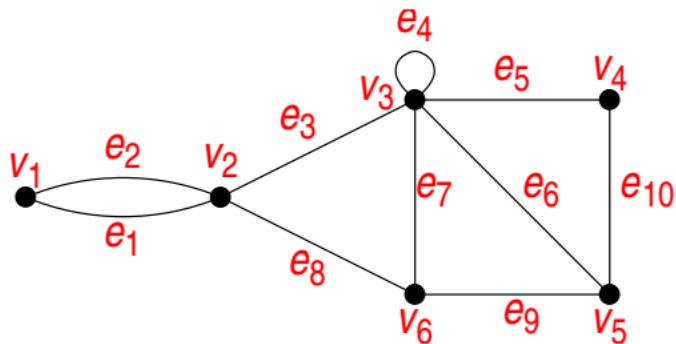
Que tipo de caminhamento é?

- $v_1e_1v_2e_3v_3e_4v_3e_5v_4$ 
  - Aresta repetida? Não.
  - Vértice repetido? Sim –  $v_3$ .
  - Começa e termina no mesmo vértice? Não.
- Trajeto.
- $e_1e_3e_5e_5e_6$ 
  - Aresta repetida? Sim –  $e_5$ .
  - Vértice repetido? Sim –  $v_3$ .
  - Começa e termina no mesmo vértice? Não.
- Caminho.

Tipo	Aresta repetida?	Vértice repetido?	Começa e termina no mesmo vértice?
Caminho ( <i>walk</i> )	Pode	Pode	Pode
Caminho fechado ( <i>closed walk</i> )	Pode	Pode	Sim
Trajeto ( <i>path</i> )	Não	Pode	Pode
Trajeto simples ( <i>simple path</i> )	Não	Não	Não
Círculo ( <i>circuit</i> )	Não	Pode	Sim
Círculo simples ( <i>simple circuit</i> )	Não	$v_0 = v_n$	Sim

# Caminhamentos em grafos - Caminho

## Identificando o caminhamento



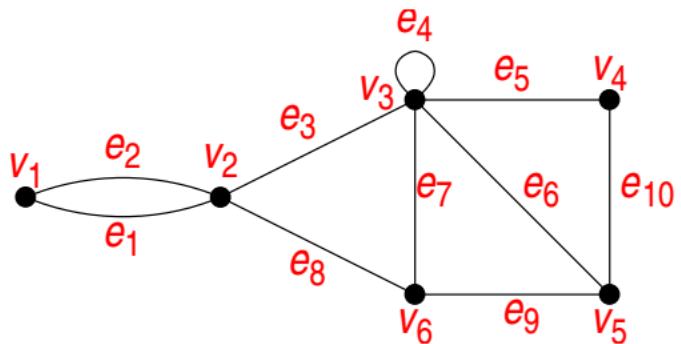
Tipo	Aresta repetida?	Vértice repetido?	Começa e termina no mesmo vértice?
Caminho ( <i>walk</i> )	Pode	Pode	Pode
Caminho fechado ( <i>closed walk</i> )	Pode	Pode	Sim
Trajeto ( <i>path</i> )	Não	Pode	Pode
Trajeto simples ( <i>simple path</i> )	Não	Não	Não
Círculo ( <i>circuit</i> )	Não	Pode	Sim
Círculo simples ( <i>simple circuit</i> )	Não	$v_0 = v_n$	Sim

Que tipo de caminhamento é?

- $v_2v_3v_4v_5v_3v_6v_2$ 
  - Aresta repetida? Não.
  - Vértice repetido? Sim –  $v_2$  e  $v_3$ .
  - Começa e termina no mesmo vértice? Sim –  $v_2$ .
- Circuito.
- $v_2v_3v_4v_5v_6v_2$ 
  - Aresta repetida? Não.
  - Vértice repetido? Sim –  $v_2$ .
  - Começa e termina no mesmo vértice? Sim –  $v_2$ .
- Circuito simples.

# Caminhamentos em grafos - Caminho

## Identificando o caminhamento



Tipo	Aresta repetida?	Vértice repetido?	Começa e termina no mesmo vértice?
Caminho ( <i>walk</i> )	Pode	Pode	Pode
Caminho fechado ( <i>closed walk</i> )	Pode	Pode	Sim
Trajeto ( <i>path</i> )	Não	Pode	Pode
Trajeto simples ( <i>simple path</i> )	Não	Não	Não
Círculo ( <i>circuit</i> )	Não	Pode	Sim
Círculo simples ( <i>simple circuit</i> )	Não	$v_0 = v_n$	Sim

Que tipo de caminhamento é?

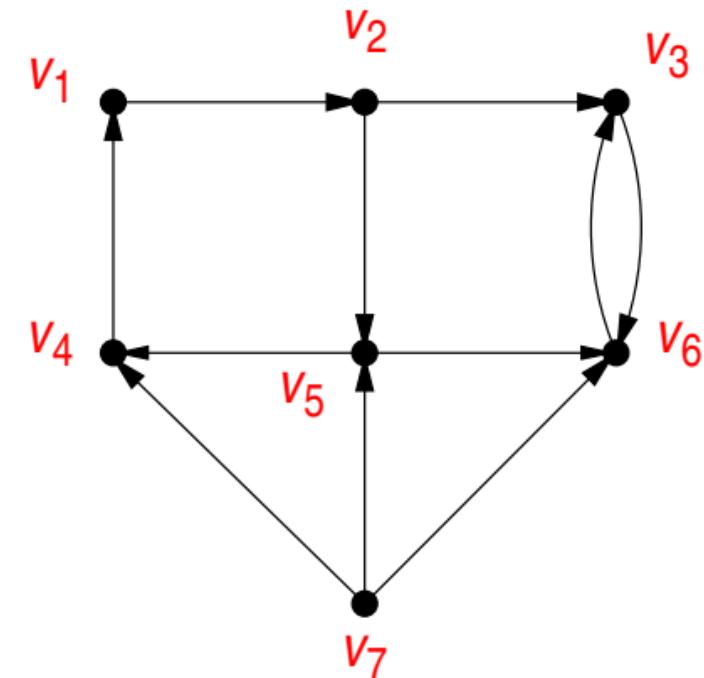
- $v_2v_3v_4v_5v_6v_3v_2$ 
  - Aresta repetida? Sim –  $e_3$ .
  - Vértice repetido? Sim –  $v_2$  e  $v_3$ .
  - Começa e termina no mesmo vértice? Sim –  $v_2$ .
  - ➔ Caminho fechado.
- $v_1$ 
  - Aresta repetida? Não.
  - Vértice repetido? Não.
  - Começa e termina no mesmo vértice? Sim –  $v_1$ .
  - ➔ Caminho (círculo) trivial.

# Caminhamentos em grafos - Caminho

## Fecho transitivo direto

Definição: O fecho transitivo direto (FTD) de um vértice  $v$  é o conjunto de todos os vértices que podem ser atingidos por algum caminho iniciando em  $v$

Exemplo: O FTD do vértice  $v_5$  do grafo ao lado é o conjunto  $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ . Note que o próprio vértice faz parte do FTD já que ele é alcançável partindo-se dele mesmo.

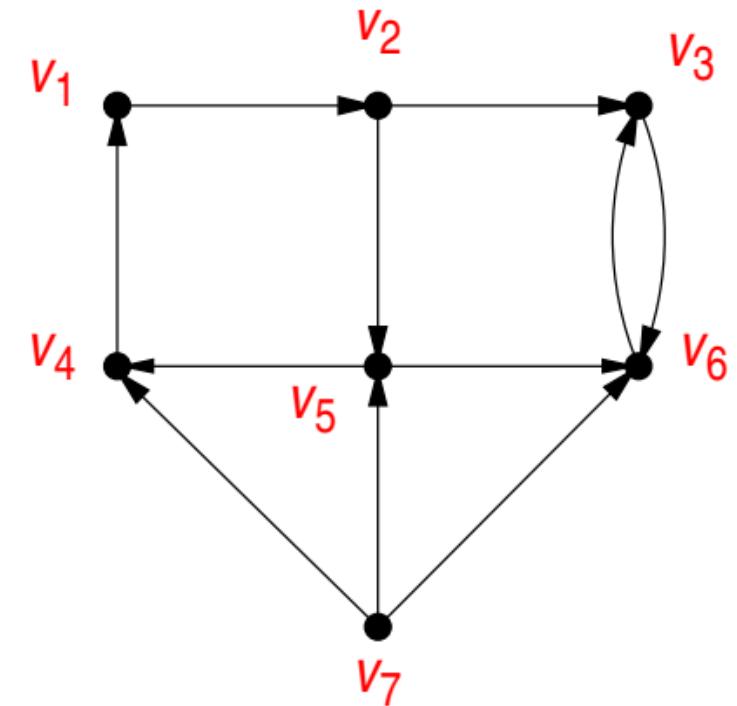


# Caminhamentos em grafos - Caminho

## Fecho transitivo inverso

Definição: O fecho transitivo inverso (FTI) de um vértice  $v$  é o conjunto de todos os vértices a partir dos quais se pode atingir  $v$  por algum caminho.

Exemplo: O FTI do vértice  $v_5$  do grafo abaixo é o conjunto  $\{v_1, v_2, v_4, v_5, v_7\}$ . Note que o próprio vértice faz parte do FTI já que dele pode alcançar ele mesmo.



# Conectividade

---

Informalmente um grafo é **conexo** (conectado) se for possível caminhar de qualquer vértice para qualquer outro vértice através de uma sequência de arestas adjacentes.

**Definição:** Seja  $G$  um grafo. Dois vértices  $v$  e  $w$  de  $G$  estão **conectados** se existe um caminho de  $v$  para  $w$ . Um grafo  $G$  é conexo se dado um par qualquer de vértice  $v$  e  $w$  em  $G$ , existe um caminho de  $v$  para  $w$ . Simbolicamente,

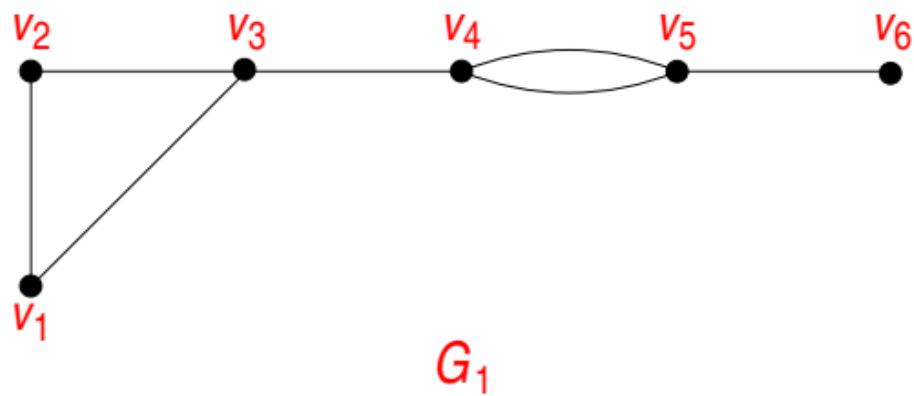
$$G \text{ é conexo} \Leftrightarrow \forall \text{ vértices } v, w \in V(G), \exists \text{ um caminho de } v \text{ para } w.$$

Se a negação desta afirmação for tomada, é possível ver que um grafo não é conexo se existem dois vértices em  $G$  que não estão conectados por qualquer caminho.

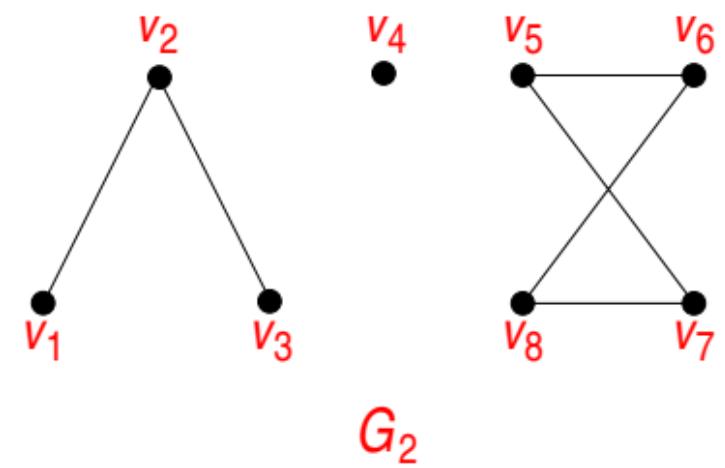
# Conectividade

---

Grafo conexo



Grafos não conexos



# Conectividade

---

## Lemas

Seja  $G$  um grafo

- a) Se  $G$  é conexo, então quaisquer dois vértices distintos de  $G$  podem ser conectados por um trajeto simples (*simple path*).
- b) Se vértices  $v$  e  $w$  são parte de um circuito de  $G$  e uma aresta é removida do circuito, ainda assim existe um trajeto de  $v$  para  $w$  em  $G$
- c) Se  $G$  é conexo e contém um circuito, então uma aresta do circuito pode ser removida sem desconectar  $G$

# Componente conexo

---

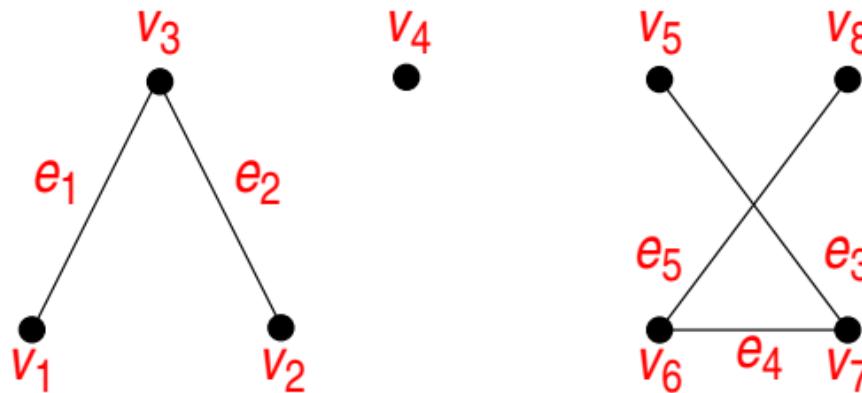
Definição: Um grafo  $H$  é um componente conexo de um grafo  $G$  se:

- 1)  $H$  é um subgrafo de  $G$ ;
  - 2)  $H$  é conexo;
  - 3) Nenhum subgrafo conexo  $I$  de  $G$  tem  $H$  como um subgrafo e  $I$  contém vértices ou arestas que não estão em  $H$ .
- Um grafo pode ser visto como a união de seus componentes conexos.

# Componente conexo

---

Os componentes conexos do grafo  $G$  abaixo são:



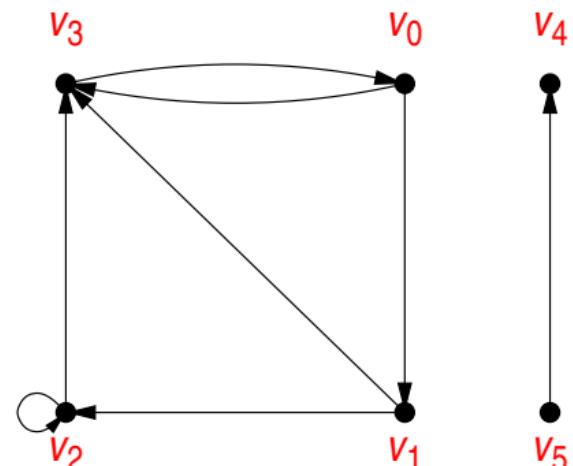
$G$  possui três componentes conexos:

$$\begin{array}{ll} H_1 : V_1 = \{v_1, v_2, v_3\} & E_1 = \{e_1, e_2\} \\ H_2 : V_2 = \{v_4\} & E_2 = \emptyset \\ H_3 : V_3 = \{v_5, v_6, v_7, v_8\} & E_3 = \{e_3, e_4, e_5\} \end{array}$$

# Componente conexo

## Componente fortemente conexo (conectado)

- Um grafo dirigido  $G = (V; E)$  é **fortemente conexo** se cada dois vértices quaisquer são alcançáveis a partir um do outro.
- Os componentes fortemente conexos de um grafo dirigido são conjuntos de vértices sob a relação “são mutuamente alcançáveis”.
- Um **grafo dirigido fortemente conexo** tem apenas um componente fortemente conexo



Os componentes fortemente conexos do grafo ao lado são:

$$H_1 : V_1 = \{v_0, v_1, v_2, v_3\}$$

$$H_2 : V_2 = \{v_4\}$$

$$H_3 : V_3 = \{v_5\}$$

Observe que  $\{v_4, v_5\}$  não é um componente fortemente conexo já que o vértice  $v_5$  não é alcançável a partir do vértice

# Círculo Euleriano

---

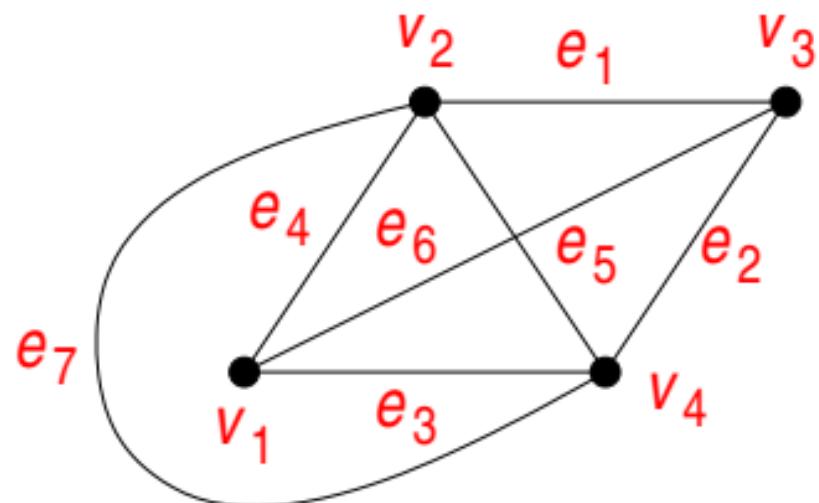
**Definição:** Seja  $G$  um grafo. Um **círculo Euleriano** é um circuito que contém cada vértice e cada aresta de  $G$ . É uma sequência de vértices e arestas adjacentes que começa e termina no mesmo vértice de  $G$ , passando pelo menos uma vez por cada vértice e exatamente uma única vez por cada aresta de  $G$ .

**Teorema:** Se um grafo possui um círculo Euleriano, então cada vértice do grafo tem grau par

**Teorema:** Se algum vértice de um grafo tem grau ímpar, então o grafo não tem um círculo Euleriano.

# Círculo Euleriano

**Teorema:** Se algum vértice de um grafo tem grau ímpar, então o grafo não tem um circuito Euleriano.



Esta versão do teorema é útil para mostrar que um grafo não possui um circuito Euleriano

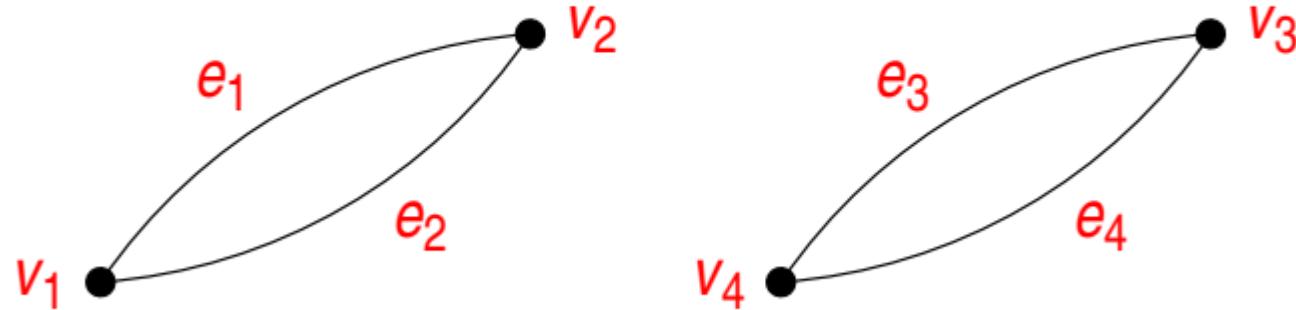
Vértices  $v_1$  e  $v_3$  possuem grau 3 e, assim, não possuem um circuito Euleriano

# Círculo Euleriano

---

No entanto, se cada vértice de um grafo tem grau par, então o grafo tem um circuito Euleriano?

- **Não.** Por exemplo, no grafo abaixo todos os vértices têm grau par, mas como o grafo não é conexo, não possui um circuito Euleriano



# Círculo Euleriano

---

**Teorema:** Se cada vértice de um grafo não vazio tem grau par e o grafo é conexo, então o grafo tem um círculo Euleriano

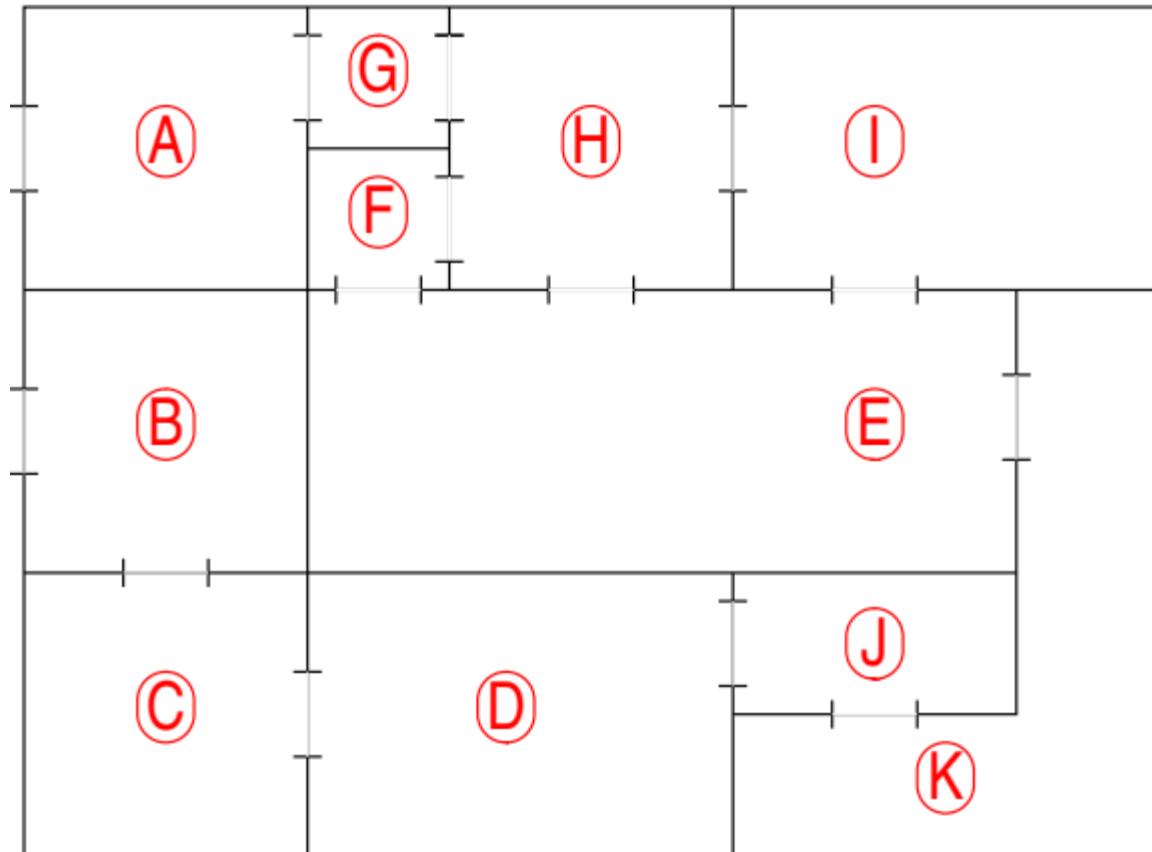
**Teorema:** Um grafo  $G$  tem um círculo Euleriano se  $G$  é conexo e cada vértice de  $G$  tem grau par

**Definição:** Seja  $G$  um grafo e seja  $v$  e  $w$  dois vértices de  $G$ . Um **Trajeto Euleriano** de  $v$  para  $w$  é uma sequência de arestas e vértices adjacentes que começa em  $v$ , termina em  $w$  e passa por cada vértice de  $G$  pelo menos uma vez e passa por cada aresta de  $G$  exatamente uma única vez.

**Corolário:** Seja  $G$  um grafo e dois vértices  $v$  e  $w$  de  $G$ . Existe um trajeto Euleriano de  $v$  para  $w$  se  $G$  é conexo e  $v$  e  $w$  têm grau ímpar e todos os outros vértices de  $G$  têm grau par

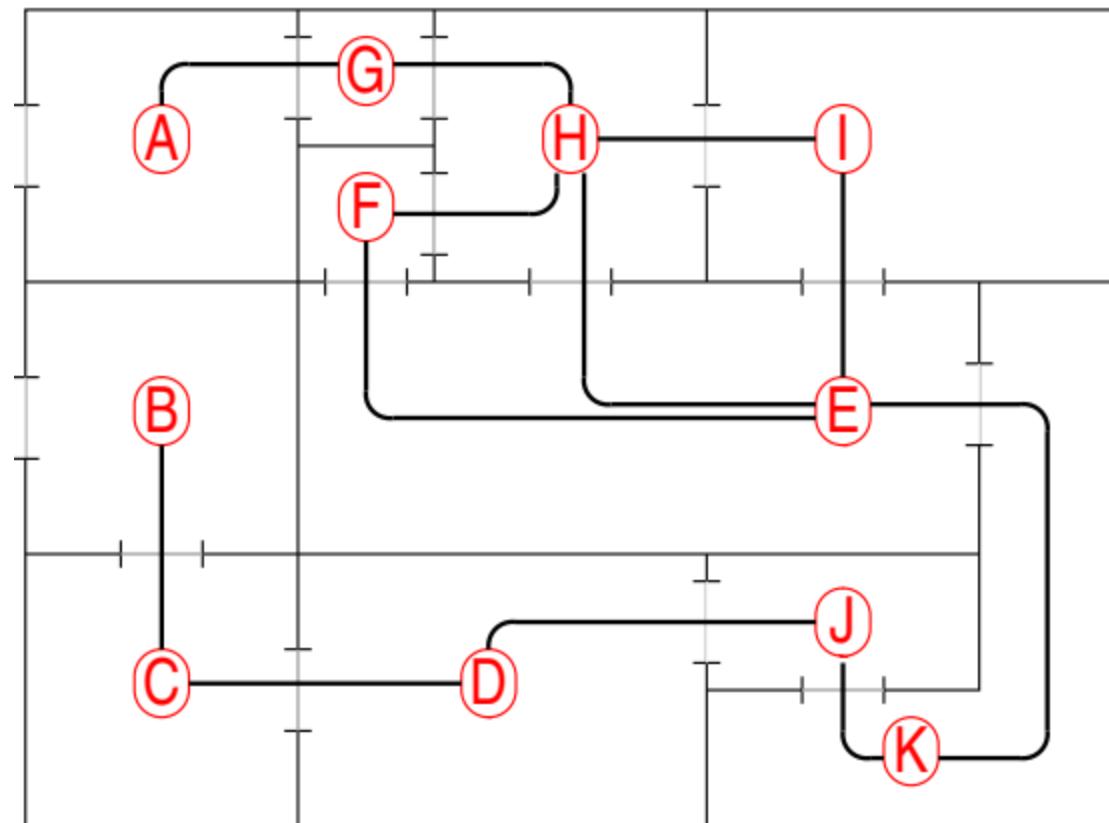
# Trajeto Euleriano

Uma casa possui uma divisão representada pela planta abaixo. É possível uma pessoa sair do cômodo *A*, terminar no cômodo *B* e passar por todas as portas da casa exatamente uma única vez? Se sim, apresente um possível trajeto.



# Trajeto Euleriano

A planta da casa pode ser representada pelo grafo :



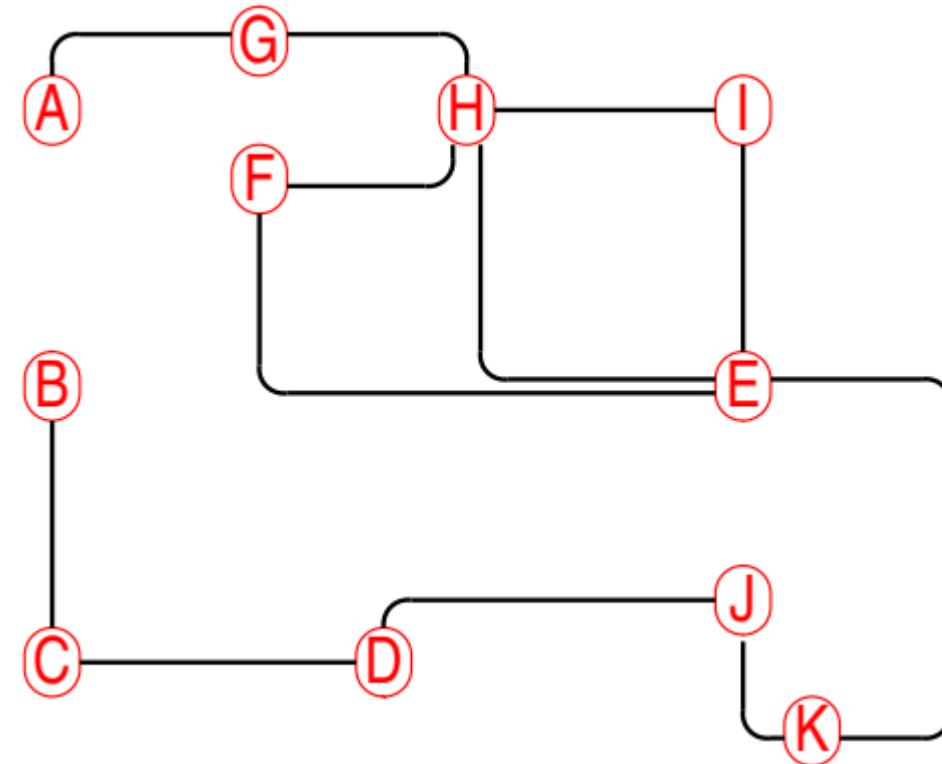
# Trajeto Euleriano

---

Cada vértice deste grafo tem um grau par, exceto os vértices  $A$  e  $B$  que têm grau 1.

Assim, pelo corolário anterior, existe um trajeto Euleriano de  $A$  para  $B$ .

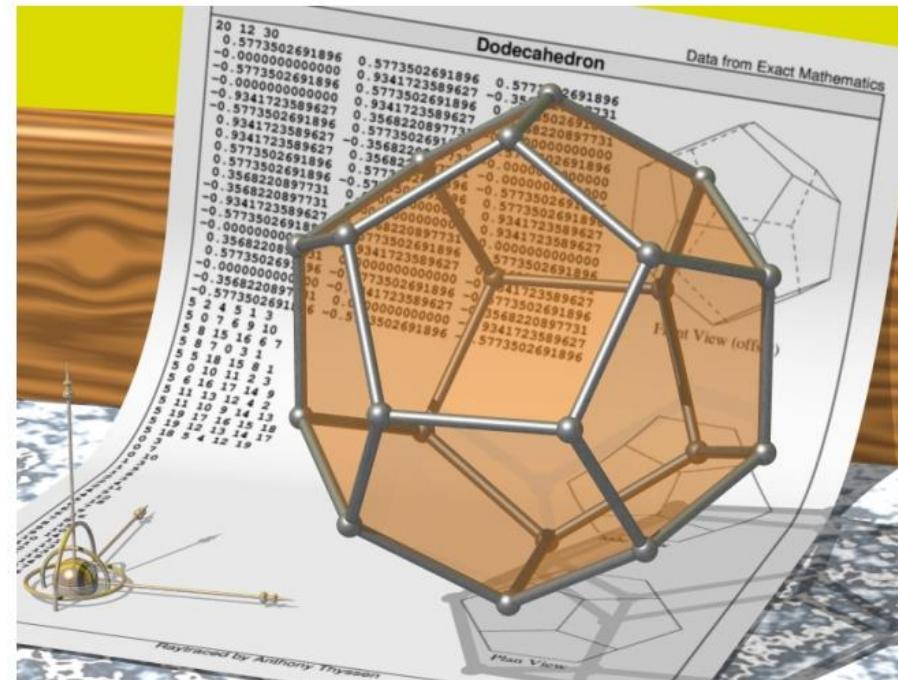
→AGHFEIHEKJDCB



# Círcuito Hamiltoniano

William Hamilton (1805–1865), matemático irlandês. Contribuiu para o desenvolvimento da óptica, dinâmica e álgebra. Em particular, descobriu a álgebra dos *quaternions*. Seu trabalho provou ser significante para o desenvolvimento da mecânica quântica.

Em 1859, propôs um jogo na forma de um dodecaedro (sólido de 12 faces)

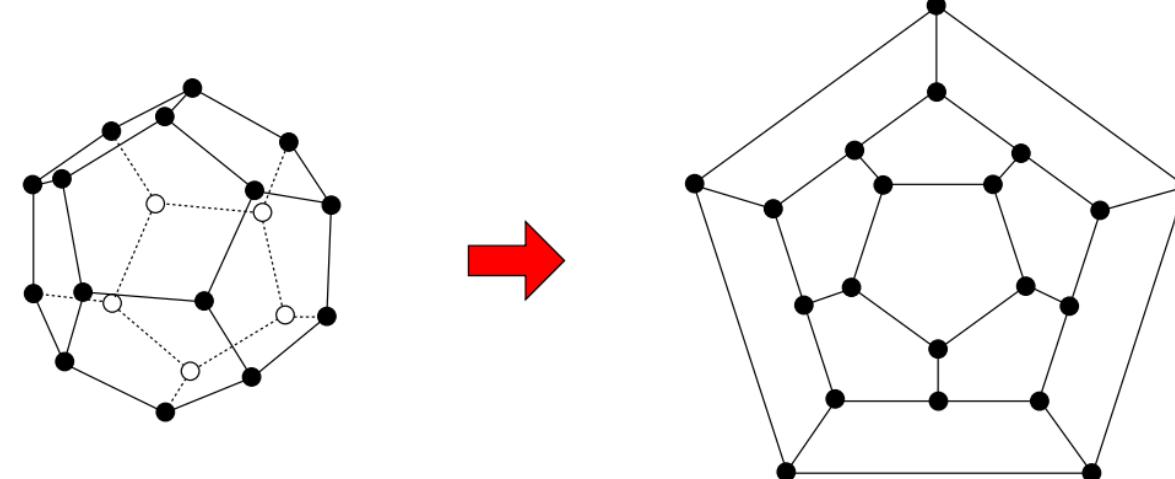


# Círculo Hamiltoniano

## Jogo proposto por Hamilton

Cada vértice recebeu o nome de uma cidade: Londres, Paris, Hong Kong, New York, etc. O problema era: É possível começar em uma cidade e visitar todas as outras cidades exatamente uma única vez e retornar à cidade de partida?

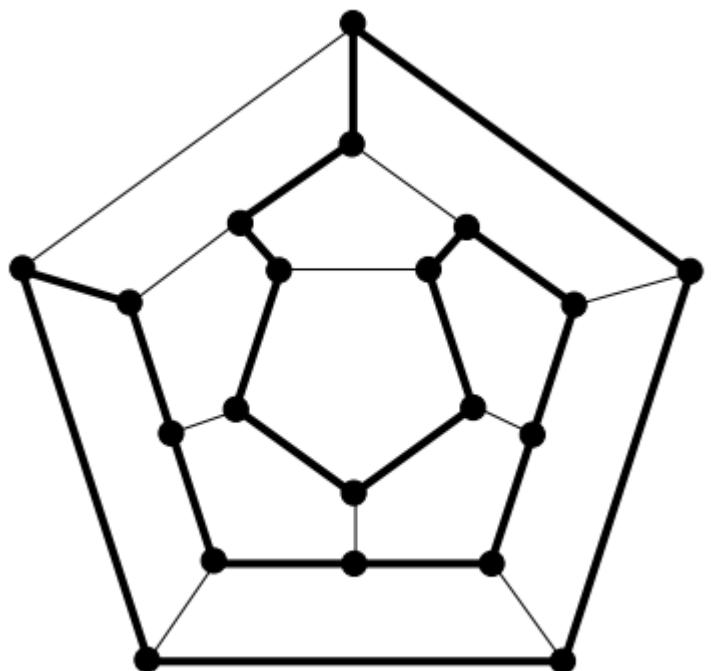
O jogo é mais fácil de ser imaginado projetando o dodecaedro no plano:



# Círculo Hamiltoniano

---

Uma possível solução para este grafo é:



Definição: Dado um grafo  $G$ , um **Círculo Hamiltoniano** para  $G$  é um circuito simples que inclui cada vértice de  $G$ , ou seja, uma sequência de vértices adjacentes e arestas distintas tal que cada vértice de  $G$  aparece exatamente uma única vez

# Comentários sobre circuitos Euleriano e Hamiltoniano

---

Círculo Euleriano:

- Inclui todas as arestas uma única vez.
- Inclui todos os vértices, mas que podem ser repetidos, ou seja, pode não gerar um circuito Hamiltoniano.

Círculo Hamiltoniano

- Inclui todos os vértices uma única vez (exceto o inicial = final).
- Pode não incluir todas as arestas, ou seja, pode não gerar um circuito Euleriano

# Comentários sobre circuitos Euleriano e Hamiltoniano

---

É possível determinar a priori se um grafo  $G$  possui um circuito Euleriano.

Não existe um teorema que indique se um grafo possui um circuito Hamiltoniano nem se conhece um algoritmo eficiente (polinomial) para achar um circuito Hamiltoniano

No entanto, existe uma técnica simples que pode ser usada em muitos casos para mostrar que um grafo não possui um circuito Hamiltoniano

# Comentários sobre circuitos Euleriano e Hamiltoniano

---

É possível determinar a priori se um grafo  $G$  possui um circuito Euleriano.

Não existe um teorema que indique se um grafo possui um circuito Hamiltoniano nem se conhece um algoritmo eficiente (polinomial) para achar um circuito Hamiltoniano

No entanto, existe uma técnica simples que pode ser usada em muitos casos para mostrar que um grafo não possui um circuito Hamiltoniano

# Determinando se um grafo não possui um circuito Hamiltoniano

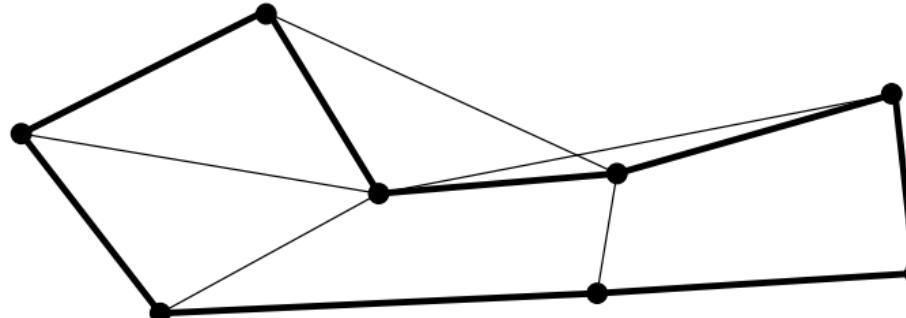
---

Suponha que um grafo  $G$  tenha um circuito Hamiltoniano  $C$  dado por:

$$C : v_0e_1v_1e_2 \dots v_{n-1}e_nv_n$$

Como  $C$  é um circuito simples, todas as arestas  $e_i$  são distintas e todos os vértices são distintos, exceto  $v_0 = v_n$ .

Seja  $H$  um subgrafo de  $G$  que é formado pelos vértices e arestas de  $C$ , como mostrado na figura abaixo ( $H$  é o subgrafo com as linhas grossas)

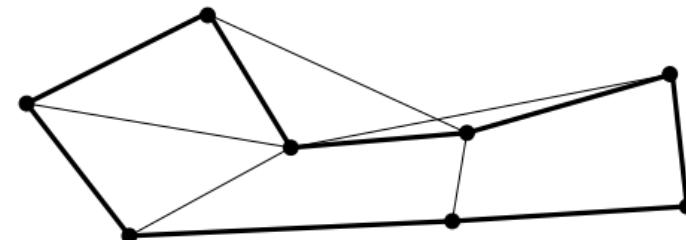


# Determinando se um grafo não possui um circuito Hamiltoniano

---

Se um grafo  $G$  tem um circuito Hamiltoniano então  $G$  tem um subgrafo  $H$  com as seguintes propriedades:

1.  $H$  contém cada vértice de  $G$ ;
2.  $H$  é conexo (existe um caminho entre qualquer par de vértices)
3.  $H$  tem o mesmo número de arestas e de vértices;
4. Cada vértice de  $H$  tem grau 2.



Contrapositivo desta afirmação:

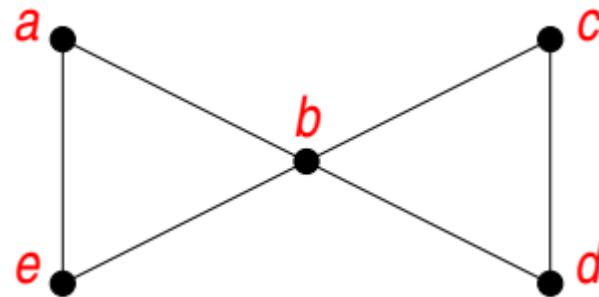
→ Se um grafo  $G$  não tem um subgrafo  $H$  com propriedades (1)–(4) então  $G$  não possui um circuito Hamiltoniano

# Determinando se um grafo não possui um circuito Hamiltoniano

Prove que o grafo  $G$  não tem um circuito Hamiltoniano.

Se  $G$  tem um circuito Hamiltoniano, então  $G$  tem um subgrafo  $H$  que:

1.  $H$  contém cada vértice de  $G$ ;
2.  $H$  é conexo;
3.  $H$  tem o mesmo número de arestas e de vértices;
4. Cada vértice de  $H$  tem grau 2.



- Em  $G$ ,  $\text{grau}(b) = 4$  e cada vértice de  $H$  tem grau 2;
  - Duas arestas incidentes a  $b$  devem ser removidas de  $G$  para criar  $H$ ;
  - Qualquer aresta incidente a  $b$  que seja removida fará com que os outros vértices restantes tenham grau menor que 2;
- Consequentemente, não existe um subgrafo  $H$  com as quatro propriedades acima e, assim,  $G$  não possui um circuito Hamiltoniano.

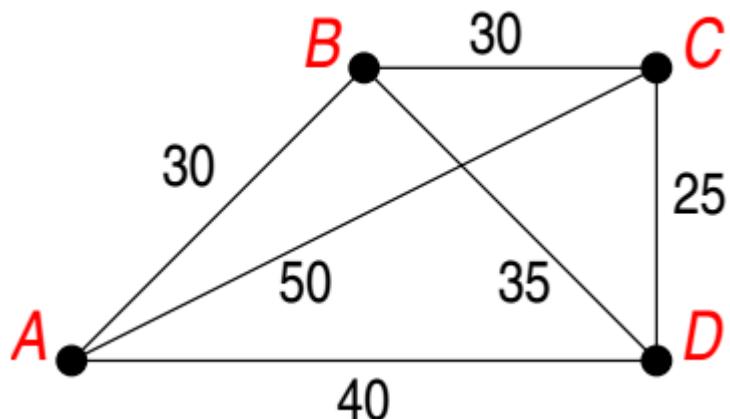
# O Problema do Caixeiro Viajante

Em inglês, *Traveling Salesman Problem*, ou TSP

Suponha o mapa mostrando quatro cidades ( $A; B; C; D$ ) e as distâncias em km entre elas.

Um caixeiro viajante deve percorrer um circuito Hamiltoniano, ou seja, visitar cada cidade exatamente uma única vez e voltar a cidade inicial.

Que rota deve ser escolhida para minimizar o total da distância percorrida?

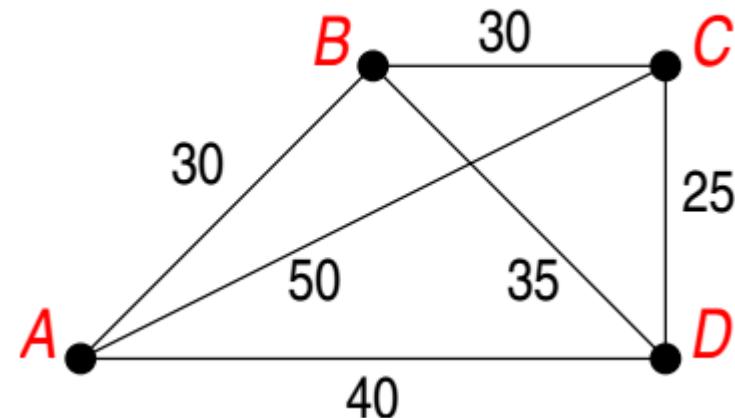


# O Problema do Caixeiro Viajante

Possível solução:

- Enumere todos os possíveis circuitos Hamiltonianos começando e terminando em A;
- Calcule a distância de cada um deles;
- Determine o menor deles.

Rota	Distância (km)
ABCDA	$30 + 30 + 25 + 40 = 125$
ABDCA	$30 + 35 + 25 + 50 = 140$
ACBDA	$50 + 30 + 35 + 40 = 155$
ACDBA	$50 + 25 + 35 + 30 = 140$
ADBCA	$40 + 35 + 30 + 50 = 155$
ADCBA	$40 + 25 + 30 + 30 = 125$



→ Tanto a rota ABCDA ou ADCBA tem uma distância total de 125 km

# O Problema do Caixeiro Viajante

---

A solução do TSP é um circuito Hamiltoniano que minimiza a distância total percorrida para um grafo valorado arbitrário  $G$  com  $n$  vértices, onde uma distância é atribuída a cada aresta.

Algoritmo para resolver o TSP:

- Atualmente, força bruta, como feito no exemplo anterior.
- Problema da classe NP-Completo

Exemplo: para o grafo  $K_{30}$  existem:

$$29! \approx 8,84 \times 10^{30}$$

Circuitos Hamiltonianos diferentes começando e terminando num determinado vértice

Mesmo se cada circuito puder ser achado e calculado em apenas  $1\mu s$ , seria necessário aproximadamente  $2,8 \times 10^{17}$  anos para terminar a computação nesse computador

# Representação de um grafo

---

Dado um grafo  $G = (V, E)$ :

- $V$  = conjunto de vértices.
- $E$  = conjunto de arestas, que pode ser representado pelo subconjunto de  $V \times V$ .

O tamanho da entrada de dados é medido em termos do:

- Número de vértices  $|V|$
- Número de arestas  $|E|$

**Se  $G$  é conexo então  $|E| \geq |V| - 1$**

# Representação de um grafo

---

## Convenção I (Notação)

- Dentro e somente dentro da notação assintótica os símbolos  $V$  e  $E$  significam respectivamente  $|V|$  e  $|E|$ .
- Se um algoritmo “executa em tempo  $O(V + E)$ ” é equivalente a dizer que “executa em tempo  $O(|V| + |E|)$ ”.

## Convenção II (Em pseudo-código):

- O conjunto  $V$  de vértices de  $G$  é representado por  $V[G]$ .
- O conjunto  $E$  de arestas de  $G$  é representado por  $E[G]$ .
- Os conjuntos  $V$  e  $E$  são vistos como atributos de  $G$ .

# Representação de um grafo

---

Matriz de adjacência:

- Forma preferida de representar grafos densos ( $|E| \approx |V|^2$ ).

Lista de adjacência:

- Representação normalmente preferida.
- Provê uma forma compacta de representar grafos esparsos ( $|E| \ll |V|^2$ )

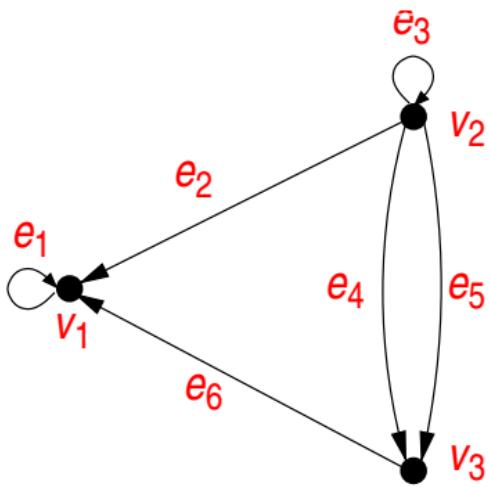
Matriz de incidência:

- Representação que inclui vértice e aresta.

As duas primeiras formas acima são as principais formas de representar um grafo

# Representação de um grafo

## Matriz de adjacência e grafo dirigido



Este grafo pode ser representado por uma matriz  $A = (a_{ij})$ , onde  $(a_{ij})$  representa o número de arestas de  $v_i$  para  $v_j$ .

$$A = \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & 1 & 0 & 0 \\ v_2 & 1 & 1 & 2 \\ v_3 & 1 & 0 & 0 \end{bmatrix}$$

**Matriz de adjacência**

Valor diferente de zero na diagonal principal: laço.

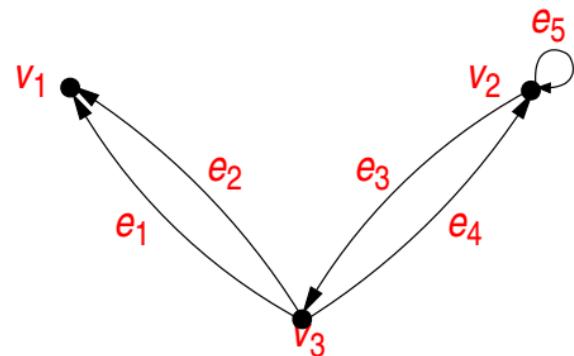
- Valor igual a 1 na entrada  $(a_{ij})$ : uma única aresta de  $v_i$  a  $v_j$ .
- Valores maiores que 1 na entrada  $(a_{ij})$ : arestas paralelas de  $v_i$  a  $v_j$

# Representação de um grafo

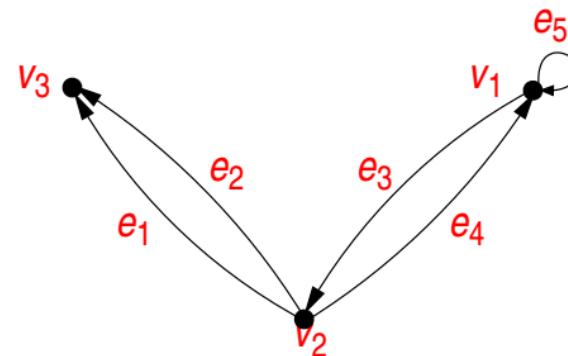
## Matriz de adjacência e grafo dirigido

Valor diferente de zero na diagonal principal: laço.

- Valor igual a 1 na entrada ( $a_{ij}$ ): uma única aresta de  $v_i$  a  $v_j$ .
- Valores maiores que 1 na entrada ( $a_{ij}$ ): arestas paralelas de  $v_i$  a  $v_j$



$$A = \begin{matrix} & \begin{matrix} v_1 & v_2 & v_3 \end{matrix} \\ \begin{matrix} v_1 \\ v_2 \\ v_3 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 2 & 1 & 0 \end{bmatrix} \end{matrix}$$



$$A = \begin{matrix} & \begin{matrix} v_1 & v_2 & v_3 \end{matrix} \\ \begin{matrix} v_1 \\ v_2 \\ v_3 \end{matrix} & \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

# Representação de um grafo

## Matriz de adjacência e grafo dirigido

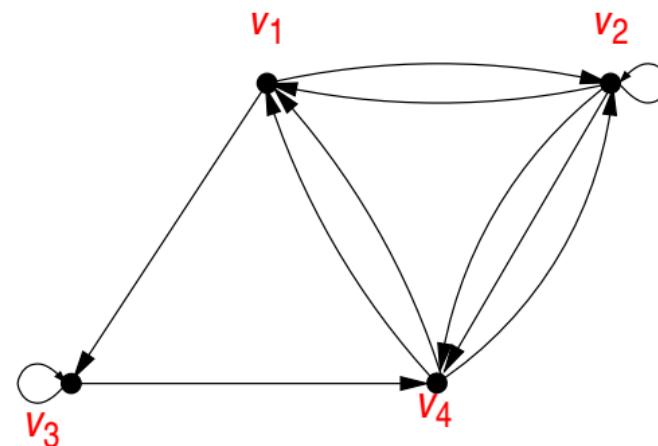
Valor diferente de zero na diagonal principal: laço.

- Valor igual a 1 na entrada ( $a_{ij}$ ): uma única aresta de  $v_i$  a  $v_j$ .
- Valores maiores que 1 na entrada ( $a_{ij}$ ): arestas paralelas de  $v_i$  a  $v_j$

Dada a matriz de adjacência de um grafo:

$$A = \begin{matrix} & v_1 & v_2 & v_3 & v_4 \\ v_1 & \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} \\ v_2 & \begin{bmatrix} 1 & 1 & 0 & 2 \end{bmatrix} \\ v_3 & \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} \\ v_4 & \begin{bmatrix} 2 & 1 & 0 & 0 \end{bmatrix} \end{matrix}$$

Um possível desenho deste grafo é:



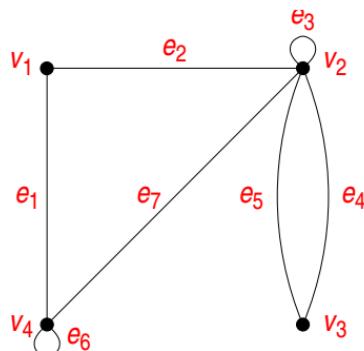
# Representação de um grafo

## Matriz de adjacência e grafo não dirigido

Definição: Seja  $G$  um grafo não dirigido com vértices  $v_1, v_2, \dots, v_n$ . A matriz de adjacência de  $G$  é a matriz  $A = (a_{ij})$  sobre o conjunto dos inteiros não negativos tal que

$$a_{ij} = \# \text{ de arestas conectando } v_i \text{ a } v_j, \forall i, j = 1, 2, \dots, n.$$

Dado o grafo:



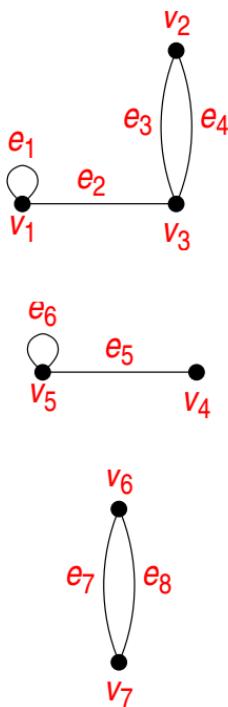
A matriz de adjacência correspondente é:

$$A = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 1 & 0 & 1 \\ v_2 & 1 & 1 & 2 & 1 \\ v_3 & 0 & 2 & 0 & 0 \\ v_4 & 1 & 1 & 0 & 1 \end{bmatrix}$$

# Representação de um grafo

## Matriz de adjacência e componentes conexos

Dado o grafo:



A matriz de adjacência correspondente é:

$$A = \left[ \begin{array}{ccc|ccc} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \end{array} \right]$$

A matriz  $A$  consiste de “blocos” de diferentes tamanhos ao longo da diagonal principal, já que o conjunto de vértices é disjunto.

# Representação de um grafo

---

## Matriz de adjacência: Análise

Deve ser utilizada para grafos densos, onde  $|E|$  é próximo de  $|V|^2$  ( $|E| \approx |V|^2$ ).

O tempo necessário para acessar um elemento é independente de  $|V|$  ou  $|E|$

É muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices

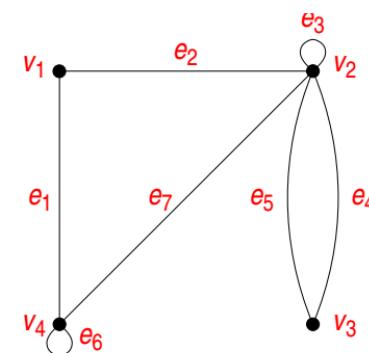
# Representação de um grafo

## Matriz de incidência

Definição: Seja  $G$  um grafo não dirigido com vértices  $v_1, v_2, \dots, v_n$  e arestas  $e_1, e_2, \dots, e_m$ . A matriz de incidência de  $G$  é a matriz  $M = (m_{ij})$  de tamanho  $n \times m$  sobre o conjunto dos inteiros não negativos tal que:

$$m_{ij} = \begin{cases} 1 & \text{quando a aresta } e_j \text{ é incidente a } v_i. \\ 0 & \text{caso contrário.} \end{cases}$$

Dado o grafo:



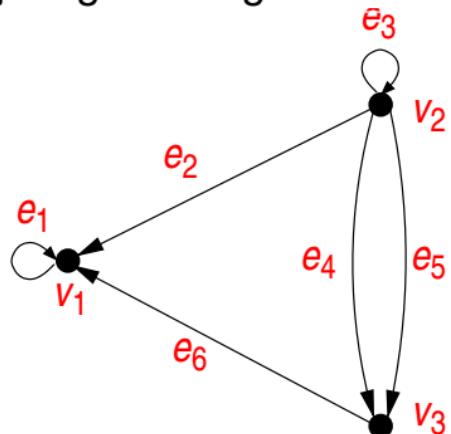
A matriz de incidência correspondente é:

$$M = \begin{bmatrix} v_1 & \left[ \begin{array}{ccccccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} \right] \\ v_2 & \left[ \begin{array}{ccccccc} 0 & 1 & 1 & 1 & 1 & 0 & 1 \end{array} \right] \\ v_3 & \left[ \begin{array}{ccccccc} 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{array} \right] \\ v_4 & \left[ \begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{array} \right] \end{bmatrix}$$

# Representação de um grafo

**Lista de adjacência e grafo dirigido**

Seja o grafo dirigido abaixo:

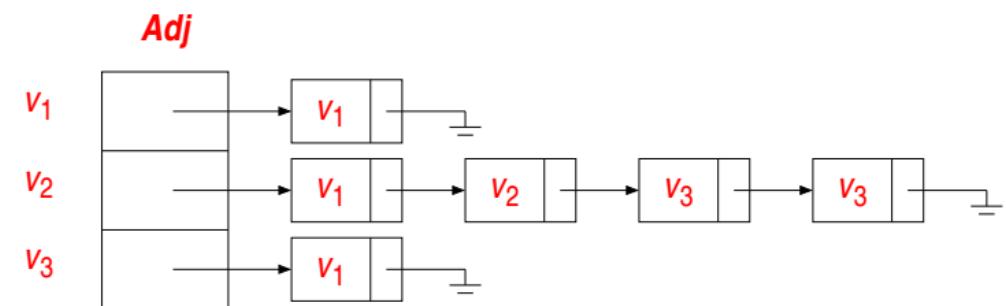


Este grafo pode ser representado por uma lista de adjacência  $Adj$ :

$$Adj[v_1] = [v_1]$$

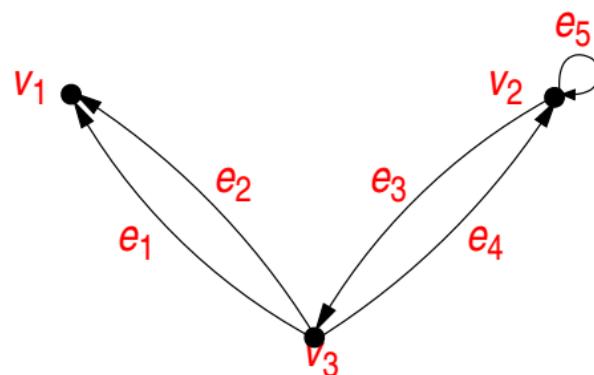
$$Adj[v_2] = [v_1, v_2, v_3, v_3]$$

$$Adj[v_3] = [v_1]$$



# Representação de um grafo

## Lista de adjacência e grafo dirigido

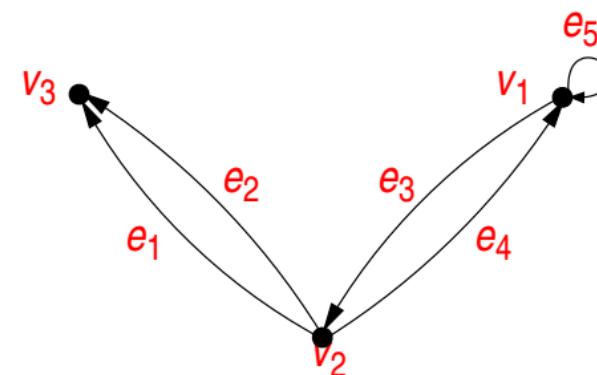


Este grafo pode ser representado pela lista de adjacência:

$$Adj[v_1] = []$$

$$Adj[v_2] = [v_3, v_3]$$

$$Adj[v_3] = [v_1, v_1, v_2]$$



Este grafo pode ser representado pela lista de adjacência:

$$Adj[v_1] = [v_1, v_2]$$

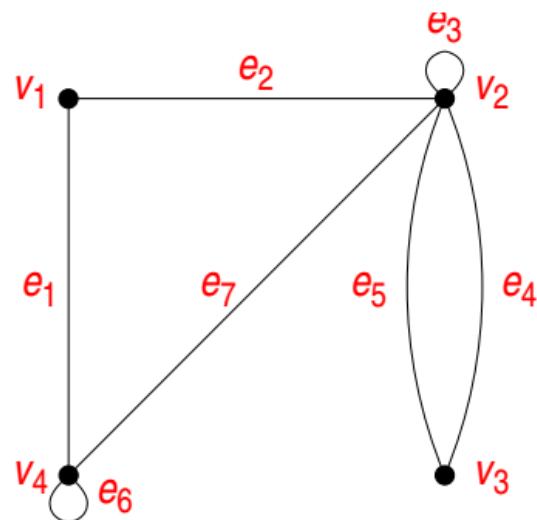
$$Adj[v_2] = [v_1, v_3, v_3]$$

$$Adj[v_3] = []$$

# Representação de um grafo

**Lista de adjacência e grafo não dirigido**

Dado o grafo:



Uma lista de adjacência correspondente é:

$$Adj[v_1] = [v_2, v_4]$$

$$Adj[v_2] = [v_1, v_2, v_3, v_3, v_4]$$

$$Adj[v_3] = [v_2, v_2]$$

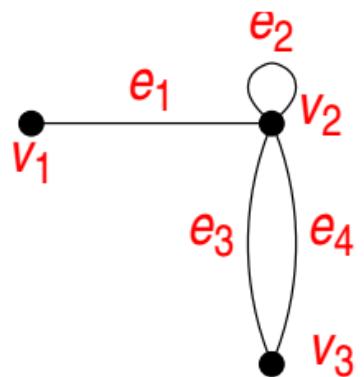
$$Adj[v_4] = [v_1, v_2, v_4]$$

# Representação de um grafo

**Contando caminhos de tamanho  $n$**

O tamanho (comprimento) de um caminho é o número de arestas do mesmo, ou seja, número de vértices menos um.

Dado o grafo



O caminho

$v_2e_3v_3e_4v_2e_2v_2e_3v_3$

tem tamanho 4.

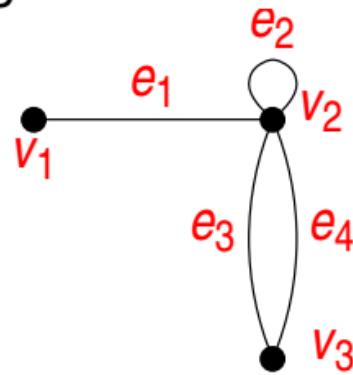
# Representação de um grafo

## Contando caminhos de tamanho $n$

Quantos caminhos distintos de tamanho  $n$  existem conectando dois vértices de um dado grafo  $G$ ?

→ Este valor pode ser computado usando multiplicação de matrizes.

Seja o grafo:



A matriz de adjacência correspondente é:

$$A = \begin{matrix} & v_1 & v_2 & v_3 \\ v_1 & 0 & 1 & 0 \\ v_2 & 1 & 1 & 2 \\ v_3 & 0 & 2 & 0 \end{matrix}$$

# Representação de um grafo

Contando caminhos de tamanho  $n$

O valor de  $A^2$  é dado por:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 2 \\ 0 & 2 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 2 \\ 0 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 6 & 2 \\ 2 & 2 & 4 \end{bmatrix}$$

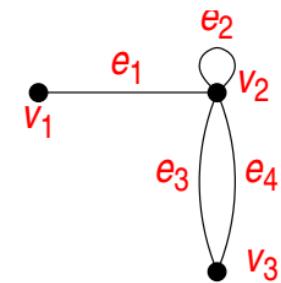
$$A_{m \times n} \cdot B_{n \times p} = C_{m \times p} \rightarrow \begin{pmatrix} 2 & 3 & \dots & 8 \\ a_{21} & a_{22} & \dots & a_{2j} \\ \vdots & & & \vdots \\ a_{i1} & a_{i2} & \dots & a_{ij} \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 6 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} 2.5 + 3.6 + \dots + 8.1 \end{pmatrix}$$

# Representação de um grafo

Contando caminhos de tamanho  $n$

$$A^2 = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 6 & 2 \\ 2 & 2 & 4 \end{bmatrix}$$

Observe que  $a_{22} = 6$ , que é o número de caminhos de tamanho 2 de  $v_2$  para  $v_2$ .



- Se  $A$  é a matriz de adjacência de um grafo  $G$ , a entrada  $a_{ij}$  da matriz  $A^2$  indica a quantidade de caminhos de tamanho 2 conectando  $v_i$  a  $v_j$  no grafo  $G$ .
- Este resultado é válido para caminhos de tamanho  $n$  calculando  $A^n$ .

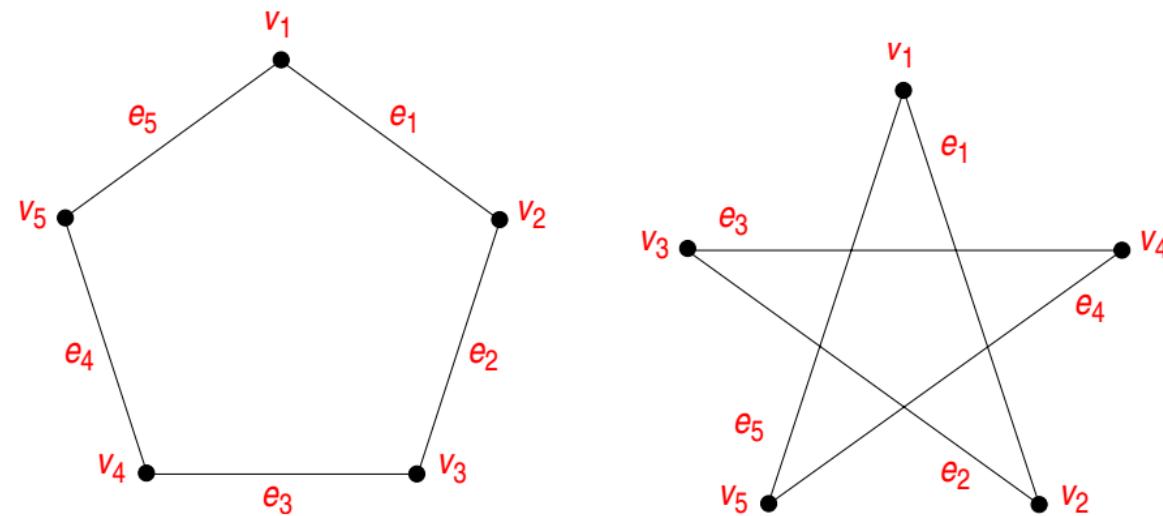
# Isomorfismo de grafos

Representam o mesmo grafo  $G$ :

Conjuntos de vértices e arestas são idênticos;

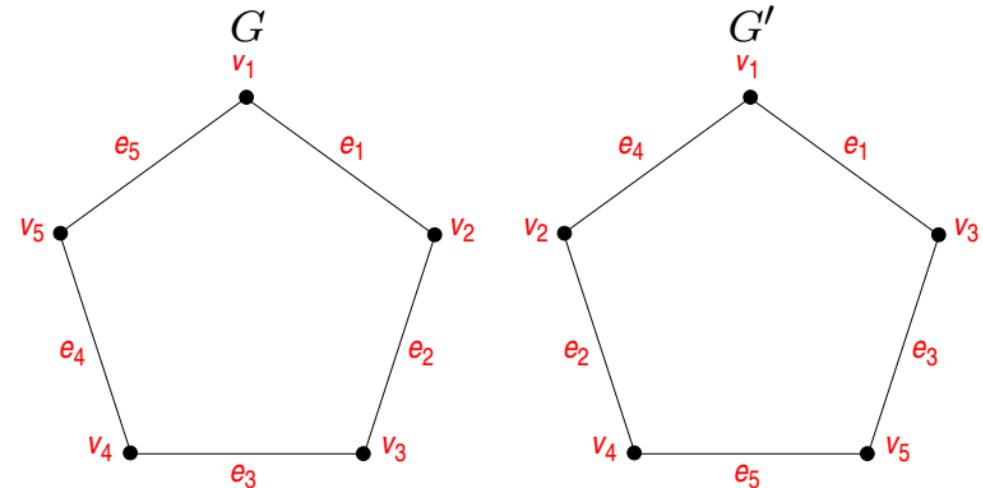
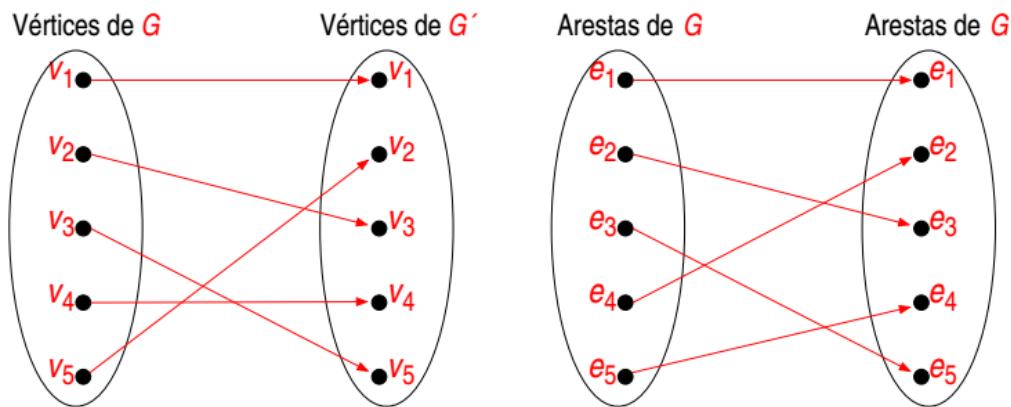
Funções aresta–vértice são as mesmas.

Grafos isomorfos (do grego, o que significa a mesma forma).



# Isomorfismo de grafos

Estes grafos são diferentes apesar de possuírem os mesmos conjuntos de vértices e arestas.  
As funções aresta–vértice não são as mesmas.



# Isomorfismo de grafos

---

Definição: Sejam os grafos  $G$  e  $G'$  com conjuntos de vértices  $V(G)$  e  $V(G')$  e com conjuntos de arestas  $E(G)$  e  $E(G')$ , respectivamente. O grafo  $G$  é isomorfo ao grafo  $G'$  se existem correspondências um-para-um:

$$g : V(G) \rightarrow V(G')$$

$$h : E(G) \rightarrow E(G')$$

que preservam as funções aresta-vértice de  $G$  e  $G'$  no sentido que:

$$\forall v \in V(G) \wedge e \in E(G)$$

$v$  é um nó terminal de  $e \Leftrightarrow g(v)$  é um nó terminal de  $h(e)$ .

# Isomorfismo de grafos

Para resolver este problema,  
devemos achar funções

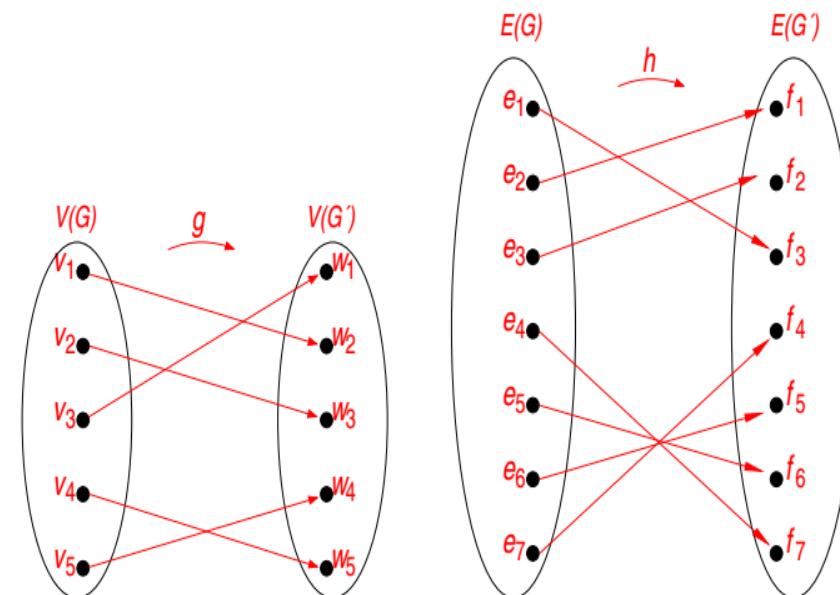
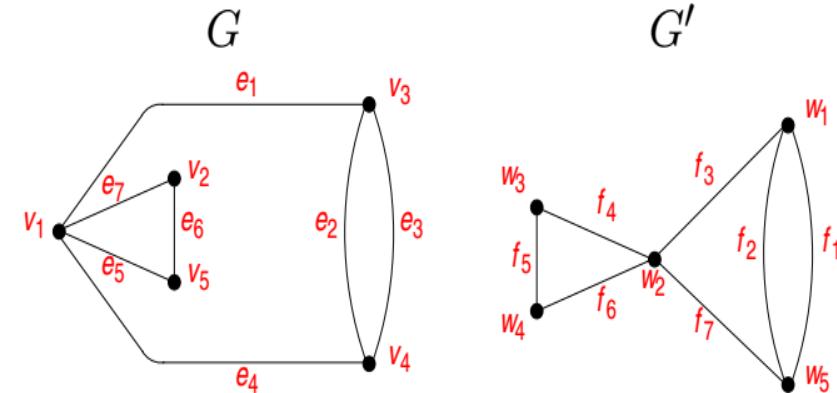
$$g : V(G) \rightarrow V(G')$$

e

$$h : E(G) \rightarrow E(G')$$

tal que exista a correspondência como mencionado anteriormente.

→ Grafos  $G$  e  $G'$  são isomorfos.



# Isomorfismo de grafos

Para resolver este problema,  
devemos achar funções

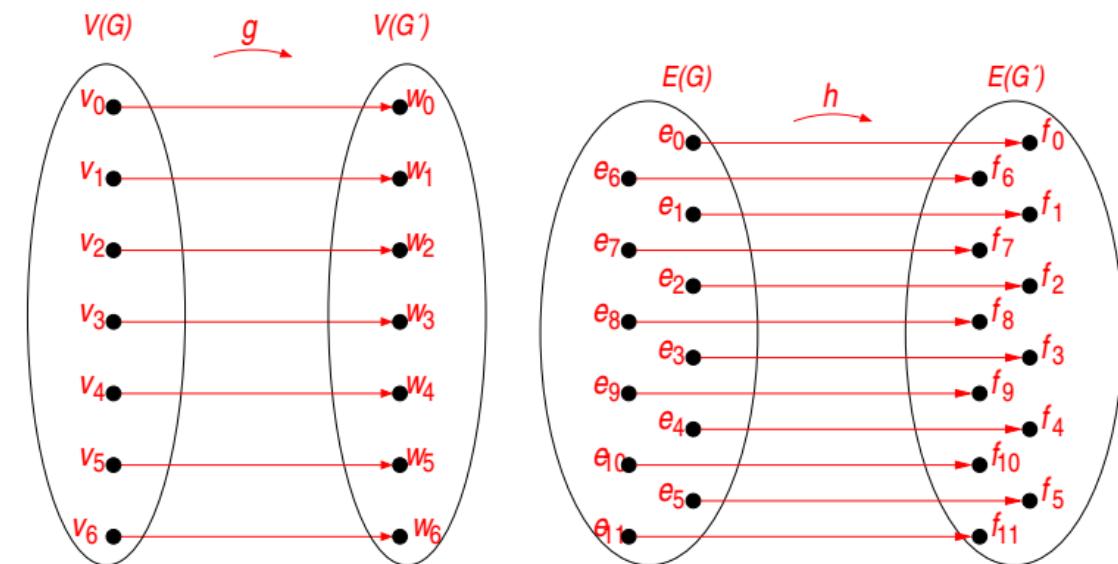
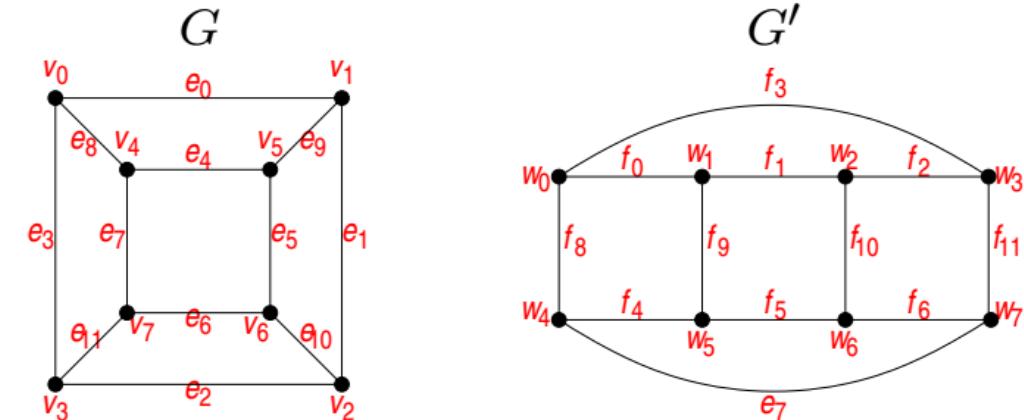
$$g : V(G) \rightarrow V(G')$$

e

$$h : E(G) \rightarrow E(G')$$

tal que exista a correspondência como mencionado anteriormente.

→ Grafos  $G$  e  $G'$  são isomorfos.



# Isomorfismo de grafos

---

Isomorfismo de grafos é uma relação de equivalência no conjunto de grafos

Informalmente, temos que esta propriedade é:

- Reflexiva: Um grafo é isomorfo a si próprio.
- Simétrica: Se um grafo  $G$  é isomorfo a um grafo  $G_0$  então  $G_0$  é isomorfo a  $G$ .
- Transitiva: Se um grafo  $G$  é isomorfo a um grafo  $G'$  e  $G'$  é isomorfo a  $G''$  então  $G$  é isomorfo a  $G''$ .

# Isomorfismo de grafos

---

## Invariante para isomorfismo de grafos

Teorema: Cada uma das seguintes propriedades é uma invariante para isomorfismo de dois grafos  $G$  e  $G'$ , onde  $n$ ,  $m$  e  $k$  são inteiros não negativos:

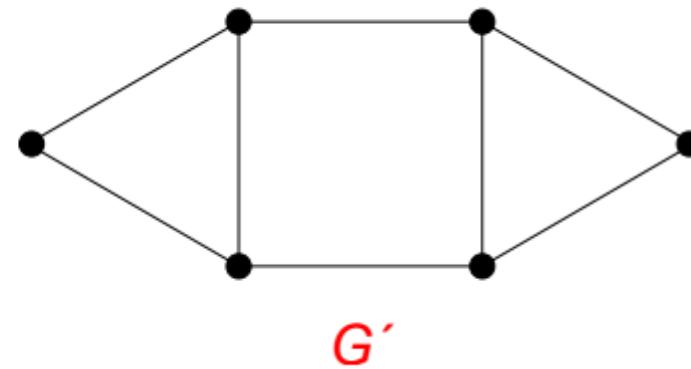
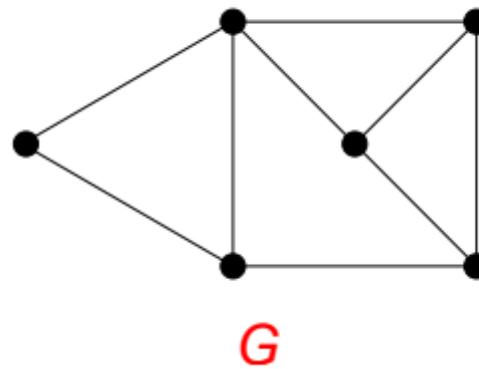
Isto significa que se  $G$  é isomorfo a  $G'$  então se  $G$  tem uma destas propriedades  $G'$  também tem.

1. Tem  $n$  vértices;
2. Tem  $m$  arestas;
3. Tem um vértice de grau  $k$ ;
4. Tem  $m$  vértices de grau  $k$ ;
5. Tem um circuito de tamanho  $k$ ;
6. Tem um circuito simples de tamanho  $k$ ;
7. Tem  $m$  circuitos simples de tamanho  $k$ ;
8. É conexo;
9. Tem um circuito Euleriano;
10. Tem um circuito Hamiltoniano.

# Isomorfismo de grafos

---

São Isomorfos?

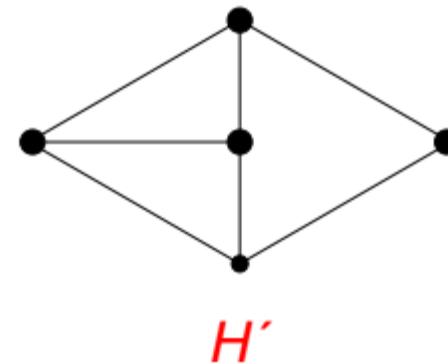
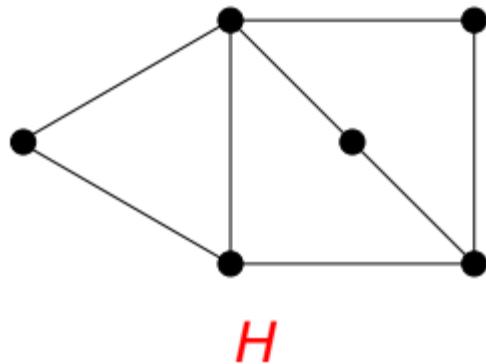


Não.  $G$  tem nove arestas e  $G'$  tem oito arestas.

# Isomorfismo de grafos

---

São Isomorfos?



Não.  $H$  tem um vértice de grau 4 e  $H'$  não tem.

# Isomorfismo de grafos

---

## Isomorfismo de grafos simples

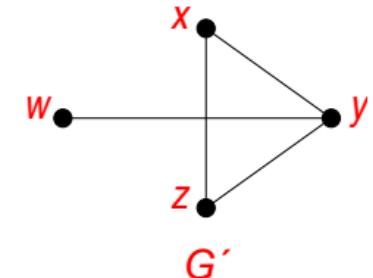
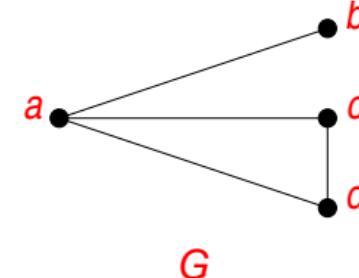
Definição: Se  $G$  e  $G'$  são grafos simples (sem arestas paralelas e sem laços) então  $G$  é isomorfo a  $G'$  sse existe uma correspondência  $g$  um-para-um do conjunto de vértices  $V(G)$  de  $G$  para o conjunto de vértices  $V(G')$  de  $G'$  que preserva a função aresta–vértice de  $G$  e de  $G'$  no sentido que

$$\forall \text{ vértices } u, v \in G$$

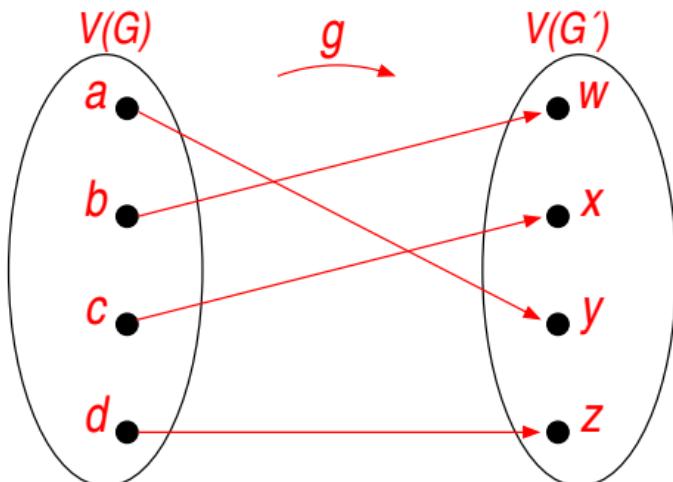
$$uv \text{ é uma aresta de } G \Leftrightarrow \{g(u), g(v)\} \text{ é uma aresta de } G'.$$

# Isomorfismo de grafos

## Isomorfismo de grafos simples



Sim, são isomorfos.



A função  $g$  preserva a função aresta–vértice de  $G$  e de  $G'$ :

Arestas de $G$	Arestas de $G'$
$ab$	$yw = \{g(a), g(b)\}$
$ac$	$yx = \{g(a), g(c)\}$
$ad$	$yz = \{g(a), g(d)\}$
$cd$	$xz = \{g(c), g(d)\}$

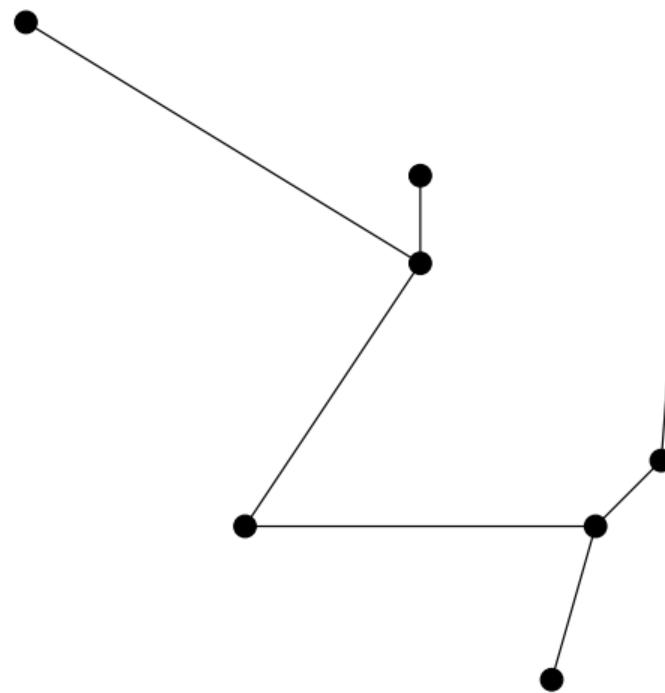
# Árvore

---

Definição: Uma árvore (também chamada de árvore livre) é um grafo não dirigido acíclico e conexo.



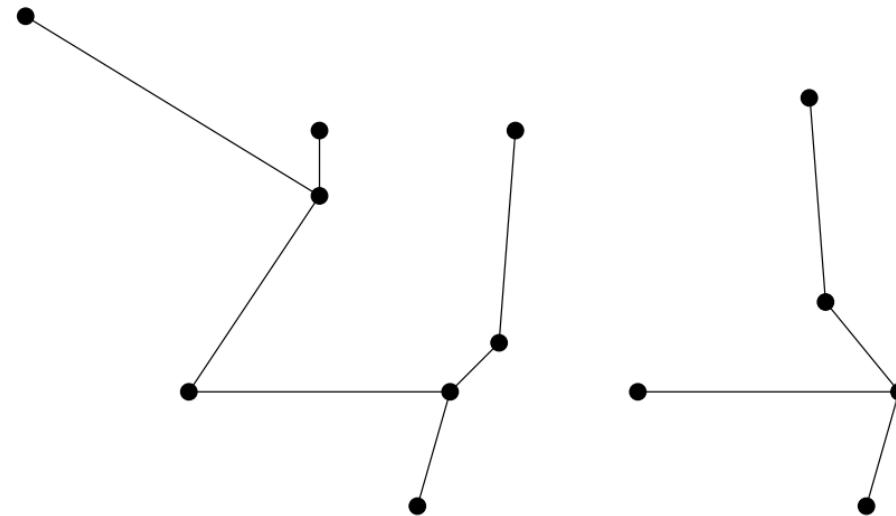
Ciclo em um grafo é um **caminho fechado** sem vértices repetidos



# Árvore

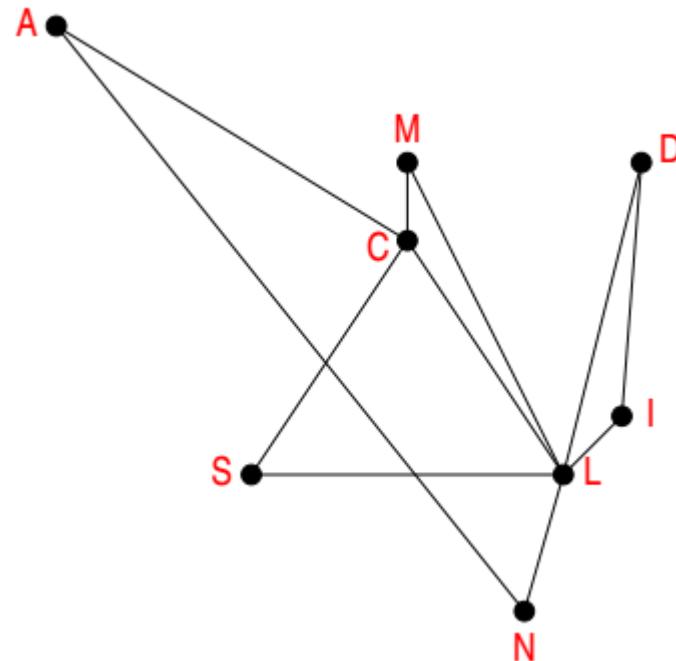
---

Definição: Uma floresta é um grafo não dirigido acíclico podendo ou não ser conexo.

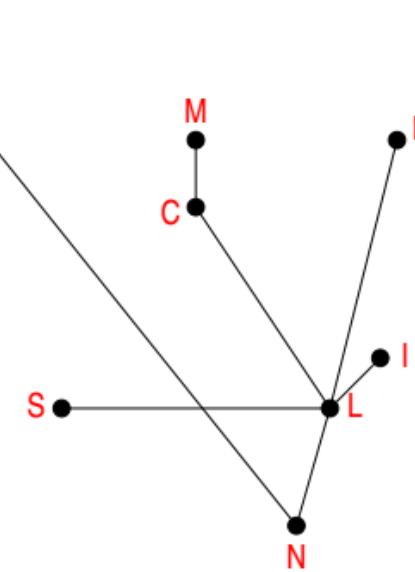


# Árvore geradora

Suponha que uma companhia aérea recebeu permissão para voar nas seguintes rotas:



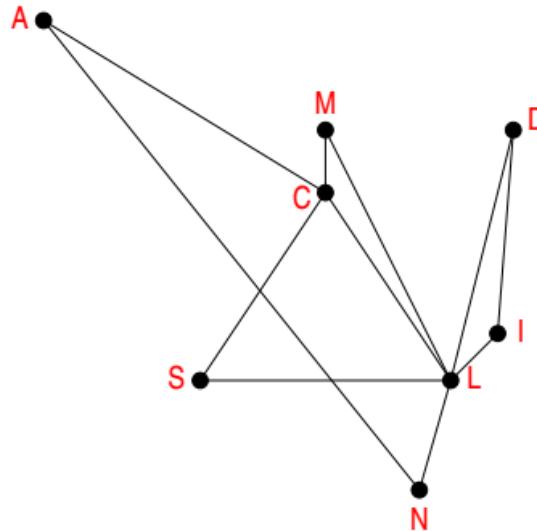
No entanto, por questões de economia, esta empresa irá operar apenas nas seguintes rotas:



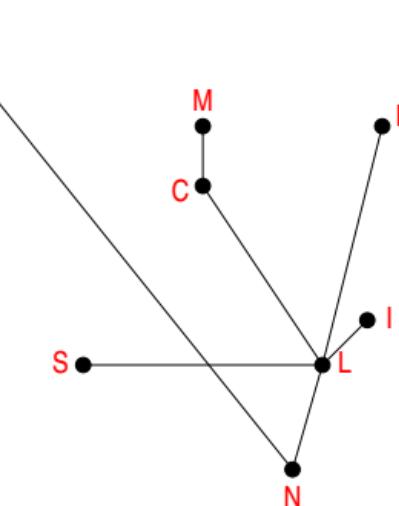
Este conjunto de rotas interconecta todas as cidades

# Árvore geradora

Suponha que uma companhia aérea recebeu permissão para voar nas seguintes rotas:



No entanto, por questões de economia, esta empresa irá operar apenas nas seguintes rotas:



Este conjunto de rotas interconecta todas as cidades

Este conjunto de rotas é mínimo?

– Sim! Qualquer árvore deste grafo possui oito vértices e sete arestas.

# Árvore geradora

---

Definição: Uma **árvore geradora** de um grafo  $G$  é um grafo que contém cada vértice de  $G$  e é uma árvore.

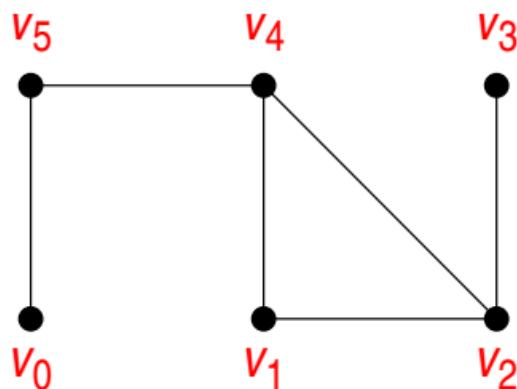
Proposição:

- Cada grafo conexo tem uma árvore geradora.
- Duas árvores geradoras quaisquer de um grafo têm a mesma quantidade de arestas

# Árvore geradora

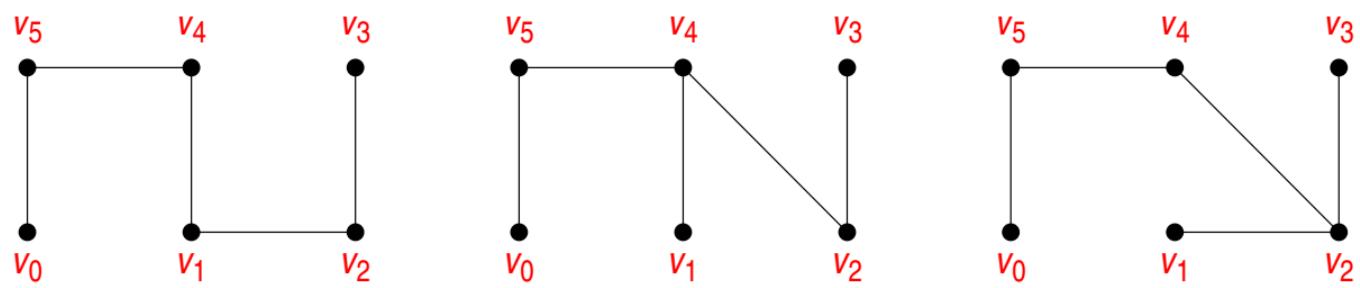
---

Seja o grafo  $G$  abaixo:



Este grafo possui o circuito  $v_2v_1v_4v_2$ .  
A remoção de qualquer uma das três arestas do circuito leva a uma árvore.

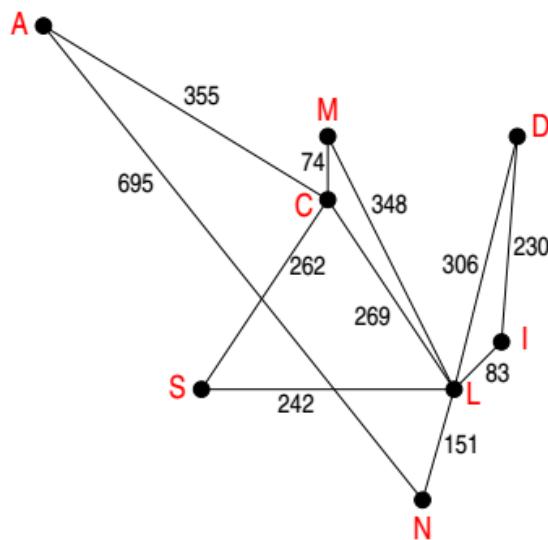
Assim, todas as três árvores geradoras são:



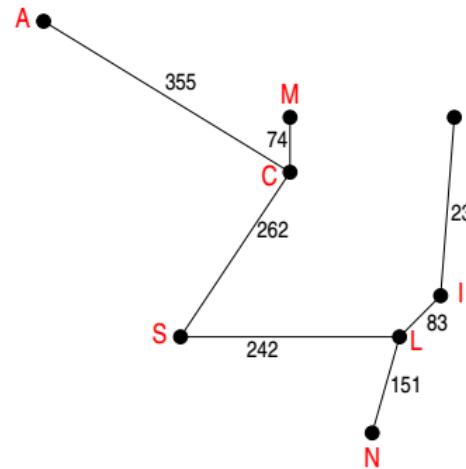
# Árvore geradora

## Árvore geradora mínima ou Minimal Spanning Tree

O grafo de rotas da companhia aérea que recebeu permissão para voar pode ser “rotulado” com as distâncias entre as cidades:



Suponha que a companhia deseja voar para todas as cidades mas usando um conjunto de rotas que minimiza o total de distâncias percorridas:



Este conjunto de rotas interconecta todas as cidades.

# Árvore geradora

---

## Árvore geradora mínima ou Minimal Spanning Tree

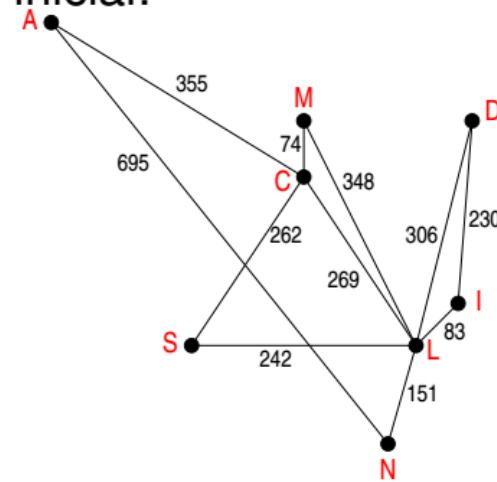
- Definição: Um **grafo com peso** é um grafo onde cada aresta possui um peso representado por um número real. A soma de todos os pesos de todas as arestas é o peso total do grafo.
- Uma **árvore geradora mínima** para um grafo com peso é uma árvore geradora que tem o menor peso total possível dentre todas as possíveis árvores geradoras do grafo

# Árvore geradora

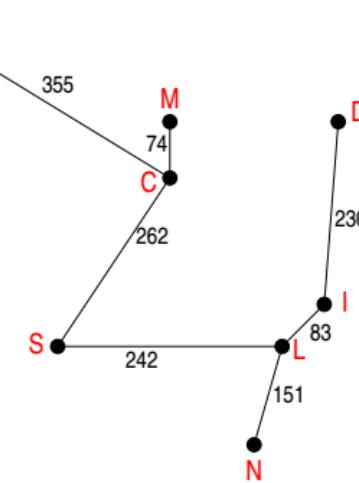
Algoritmos para obter a árvore geradora mínima

- Algoritmo de Kruskal
- Algoritmo de Prim

Grafo inicial:



Árvore geradora mínima:



# Árvore geradora

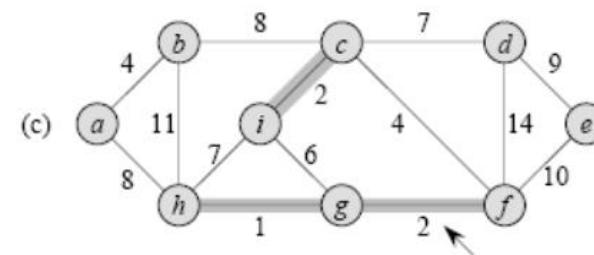
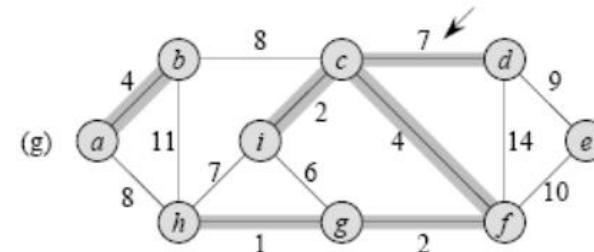
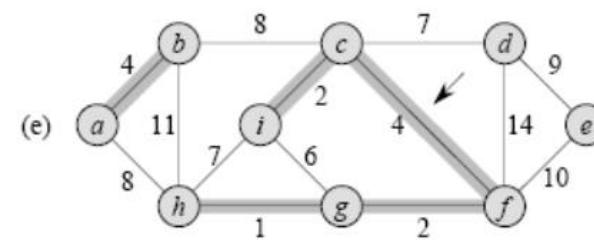
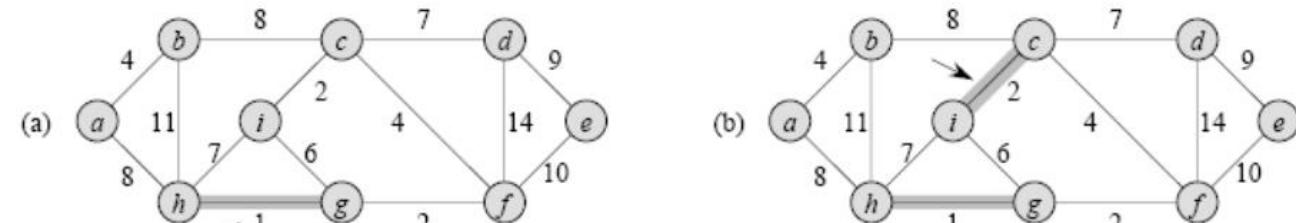
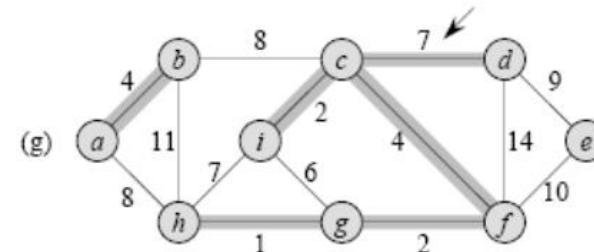
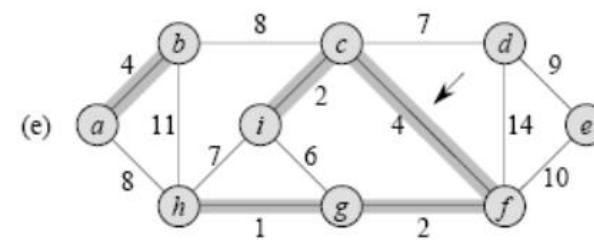
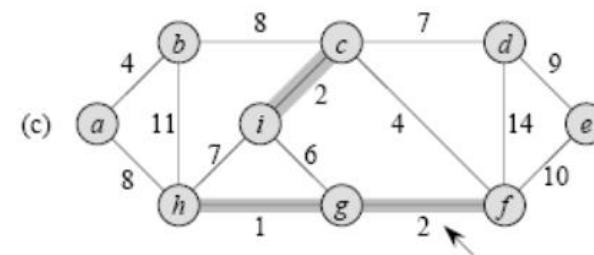
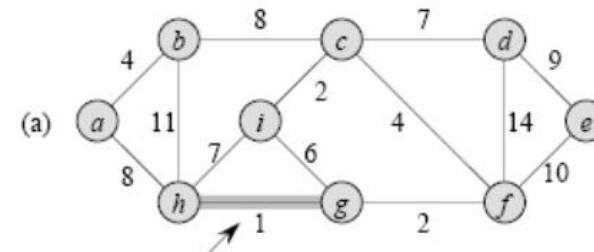
---

## Algoritmo de Kruskal

- Seleciona a aresta de menor peso que conecta duas árvores de uma floresta
- Repita o processo até que todos os vértices estejam conectados sempre preservando a invariante de se ter uma árvore.

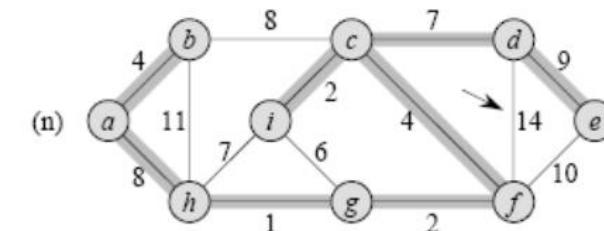
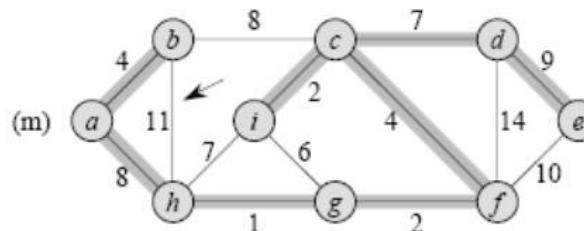
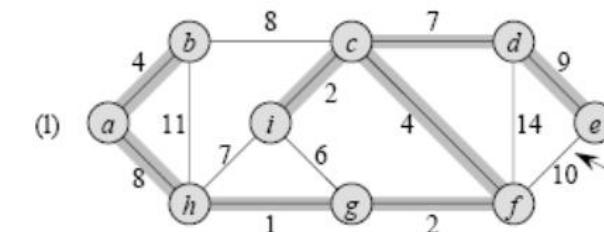
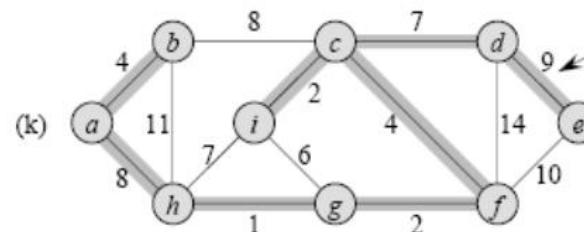
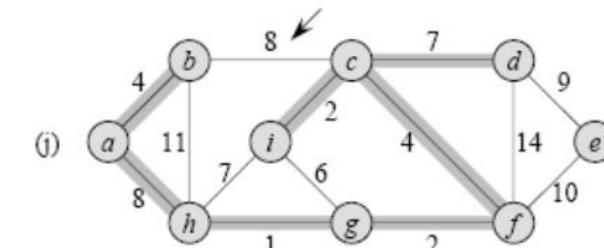
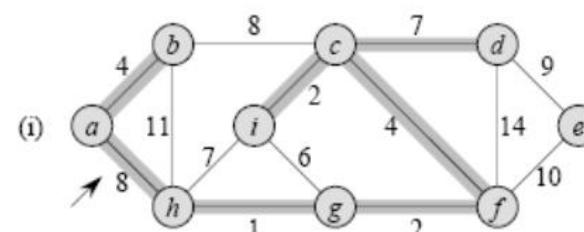
# Árvore geradora

## Algoritmo de Kruskal



# Árvore geradora

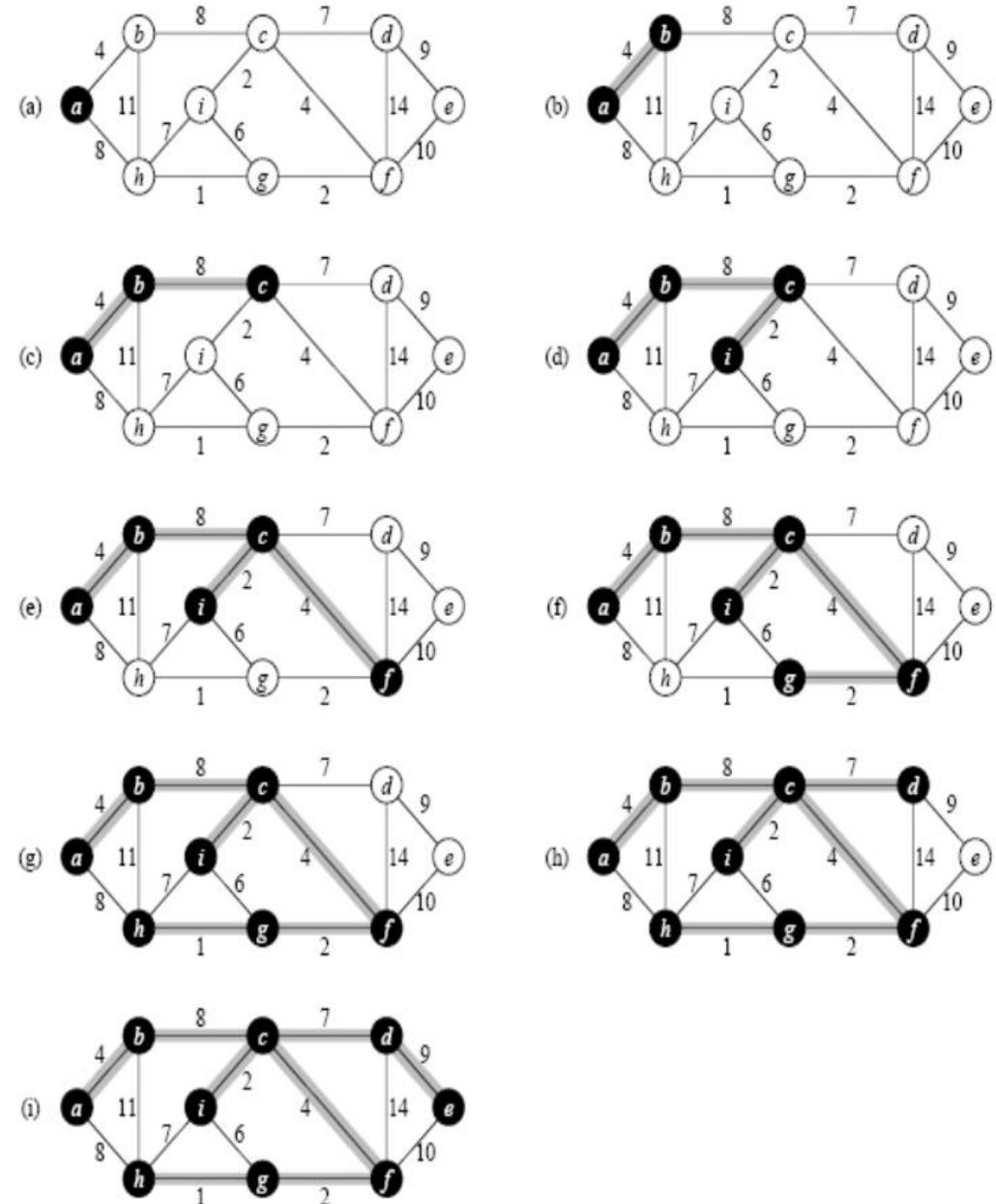
## Algoritmo de Kruskal



# Árvore geradora

## Algoritmo de Prim

- Tomando como vértice inicial  $A$ , crie uma fila de prioridades classificada pelos pesos das arestas conectando  $A$ .
- Repita o processo até que todos os vértices tenham sido visitados.



# Referências

---

Boa Ventura,Paulo, **Grafos: Teoria, Modelos e Algoritmos**, 5<sup>a</sup> edição, Editora Blucher

Loureiro, Antonio Alfredo Ferreira, <http://www.dcc.ufmg.br/~loureiro>

# Árvores binárias

---

# Árvores binária

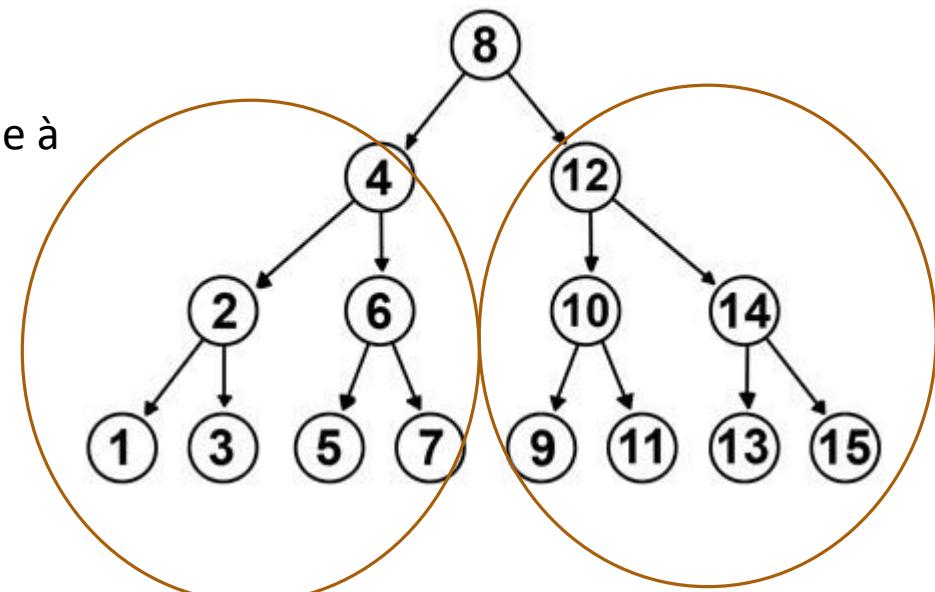
---

Uma árvore binária T é definida como um conjunto finito de elementos, chamados de nós, tal que:

- (1) T é vazio (chamado de árvore nula ou árvore vazia), ou
- (2) T contém um nó notável R, chamado de raiz de T, e os demais nós de T formam um par ordenado de árvores binárias disjuntas T1 e T2.

Se T contém uma raiz R, então as duas árvores T1 e T2 são chamadas, respectivamente, de subárvores de R à esquerda e à direita. Se T1 for não vazia, então sua raiz é chamada de sucessor à esquerda de R; analogamente, se T2 é não vazio, então sua raiz é chamada de sucessor à direita de R.

Lipschutz, Seymour; Lipson, Marc. Matemática Discreta (Coleção Schaum) (Página 235). . Edição do Kindle.



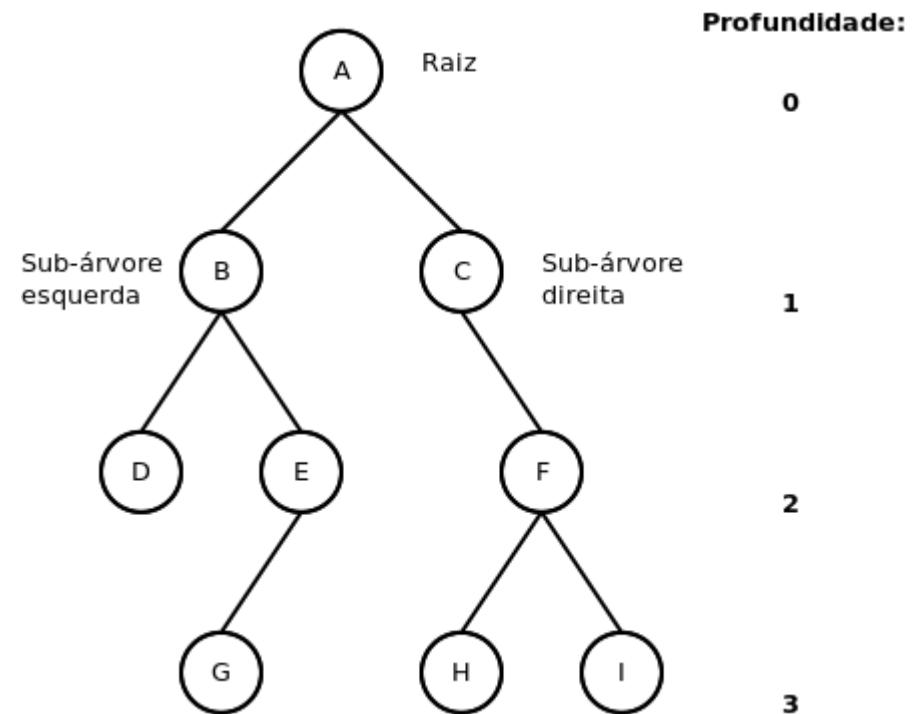
# Árvores binárias

A definição dada de uma árvore binária T é recursiva, uma vez que T é definida em termos de subárvores binárias T1 e T2.

Isso significa, em especial, que todo nó N de T contém uma subárvore à esquerda e à direita, e qualquer uma delas pode ser vazia, inclusive ambas.

Assim, todo nó N de T admite 0, 1 ou 2 sucessores.

Um nó sem sucessores é conhecido como nó terminal. Portanto, ambas as subárvores de um nó terminal são vazias.

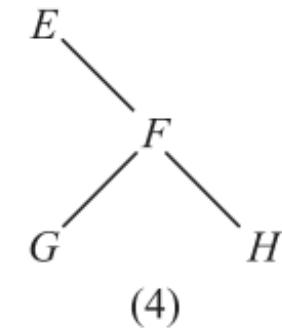
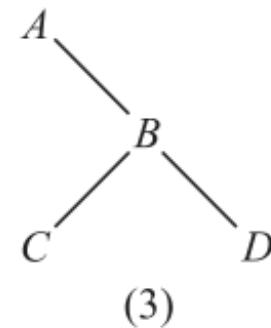
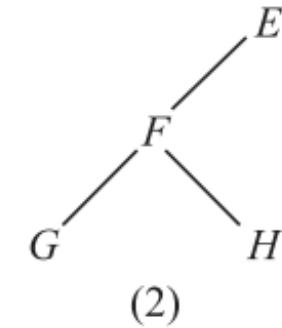
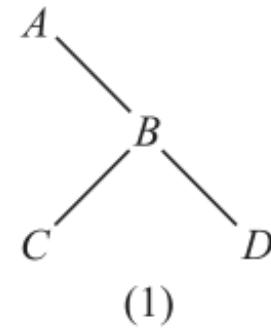


# Árvores binárias

---

## Árvores binárias semelhantes

- As árvores binárias  $T$  e  $T'$  são ditas semelhantes se têm a mesma estrutura ou, em outras palavras, se compartilham a mesma forma.
- As árvores são ditas cópias se forem semelhantes e se tiverem os mesmos conteúdos nos nós correspondentes.



(b)

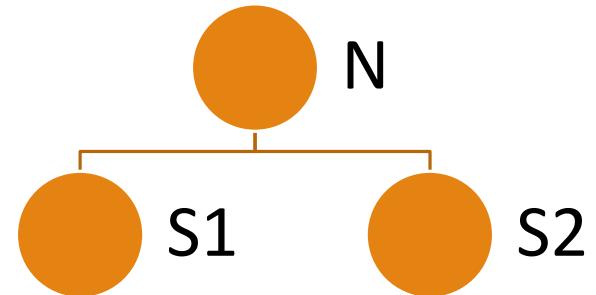
# Árvores binárias

---

## Terminologia

- Especificamente, suponha que N é um nó em T, com sucessor à esquerda  $S_1$  e sucessor à direita  $S_2$ .
- N é chamado de pai de  $S_1$  e  $S_2$ .
- Analogamente,  $S_1$  é chamado de filho à esquerda de N e  $S_2$ , de filho à direita de N.
- Além disso,  $S_1$  e  $S_2$  são ditos irmãos.
- Um nó L é dito descendente de um nó N (e N é chamado de ancestral de L) se existe uma sucessão de filhos de N a L.
- Em especial, L é chamado de descendente à esquerda ou à direita de N se L pertence à subárvore à esquerda ou à direita de N.

Todo nó N em uma árvore binária T, exceto a raiz, tem um único pai, chamado de predecessor de N.

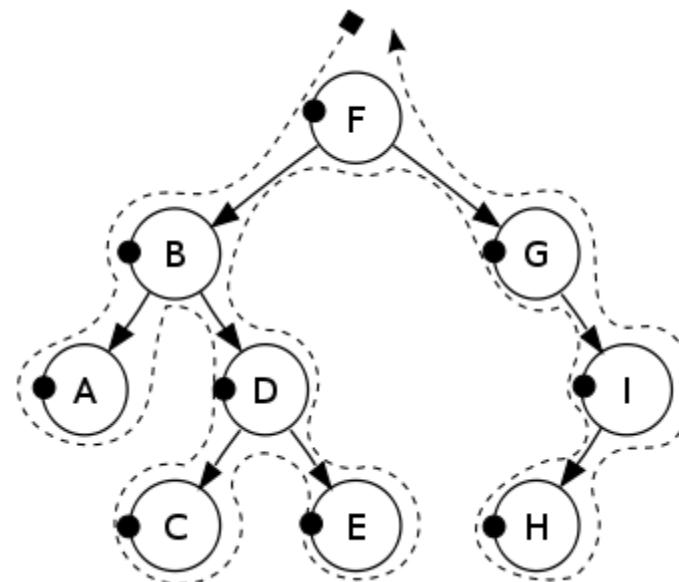


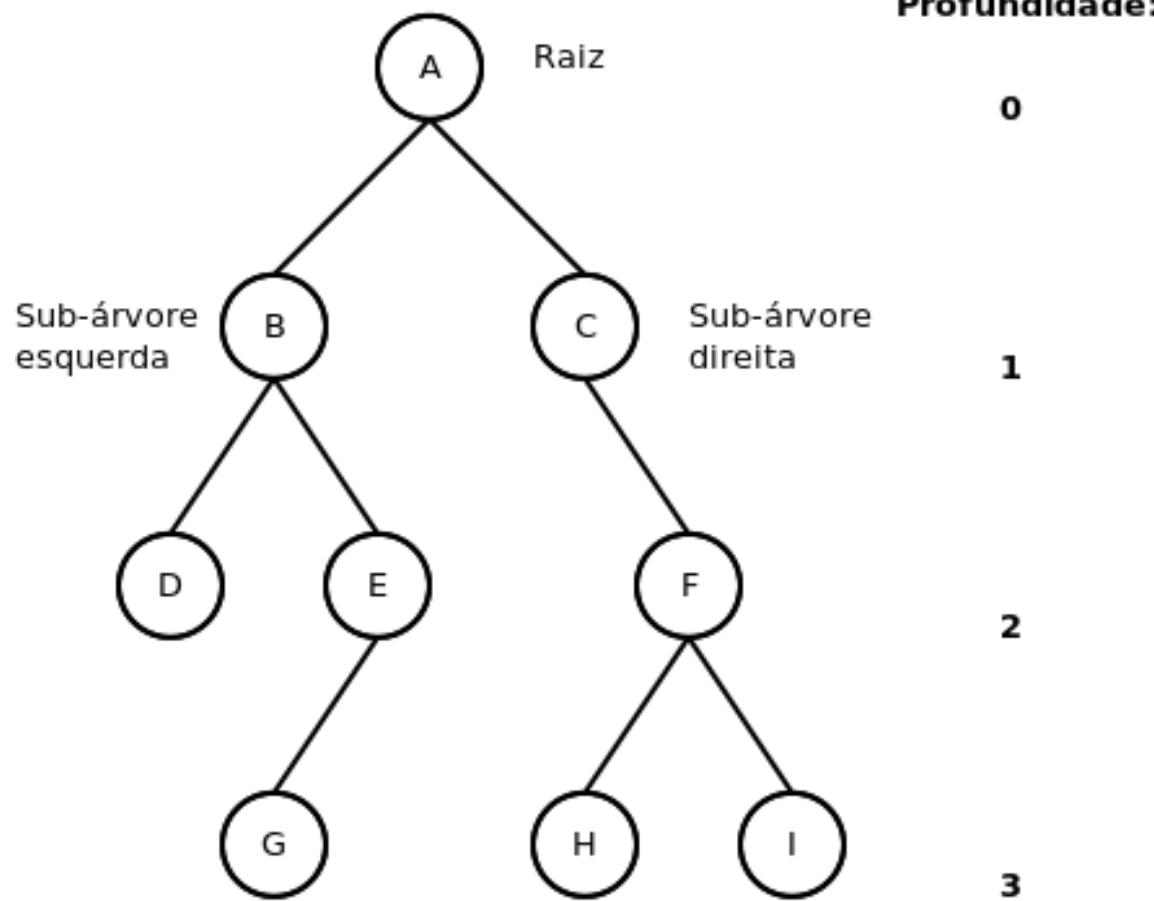
# Árvores binárias

---

## Terminologia

- Uma linha desenhada a partir de um nó N de T até um sucessor é chamada de aresta,
- E uma sequência de arestas consecutivas é conhecida como um caminho.
- Um nó terminal é chamado de folha
- Um caminho terminando em uma folha se chama ramo.





# Árvores binárias

---

## Terminologia

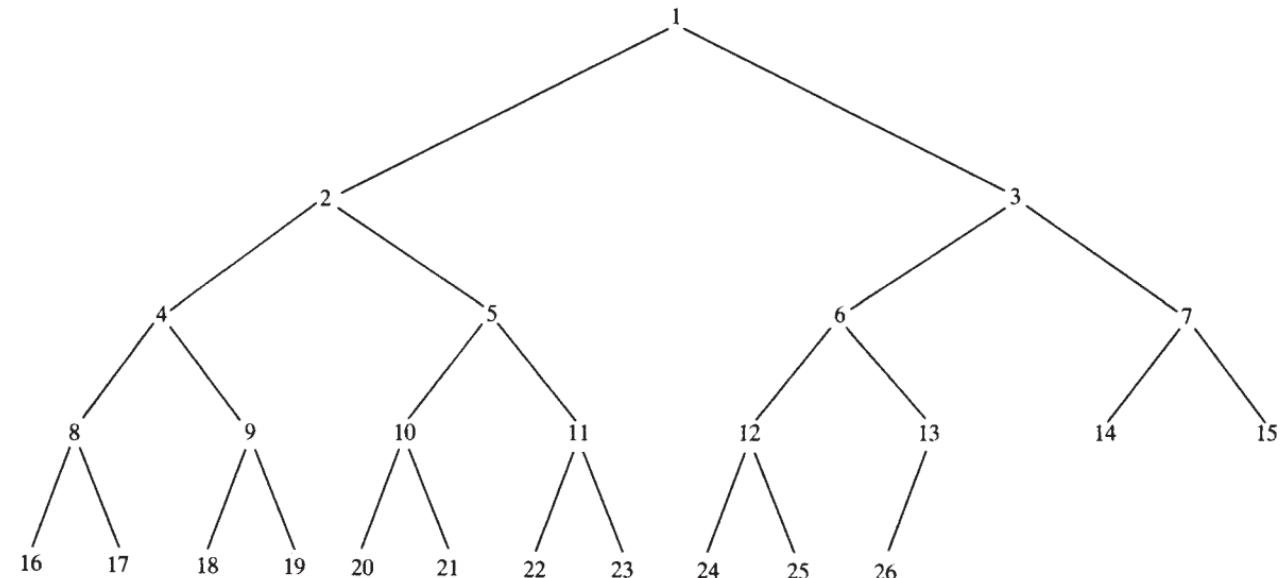
- Cada nó em uma árvore binária T é assinalado com um número de nível
- A raiz R da árvore T é assinalada ao número de nível 0
- E, todos os demais nós são designados com um número de nível que é 1 a mais do que o número de nível de seu pai.
- Além disso, os nós com o mesmo número de nível são ditos pertencerem à mesma geração.
- A profundidade (ou altura) de uma árvore T é o número máximo de nós de um ramo de T.

# Árvores binárias

---

## Árvores binárias completas

- Considere qualquer árvore binária T.
- Cada nó de T pode ter no máximo dois filhos.
- Consequentemente, o nível  $r$  de T pode ter no máximo  $2^r$  nós.
- A árvore T é dita completa se todos os seus níveis, exceto, possivelmente o último, têm o número máximo de possíveis nós e se todos os nós no último nível aparecem o mais à esquerda possível.



# Árvores binárias

## Árvores binárias completas

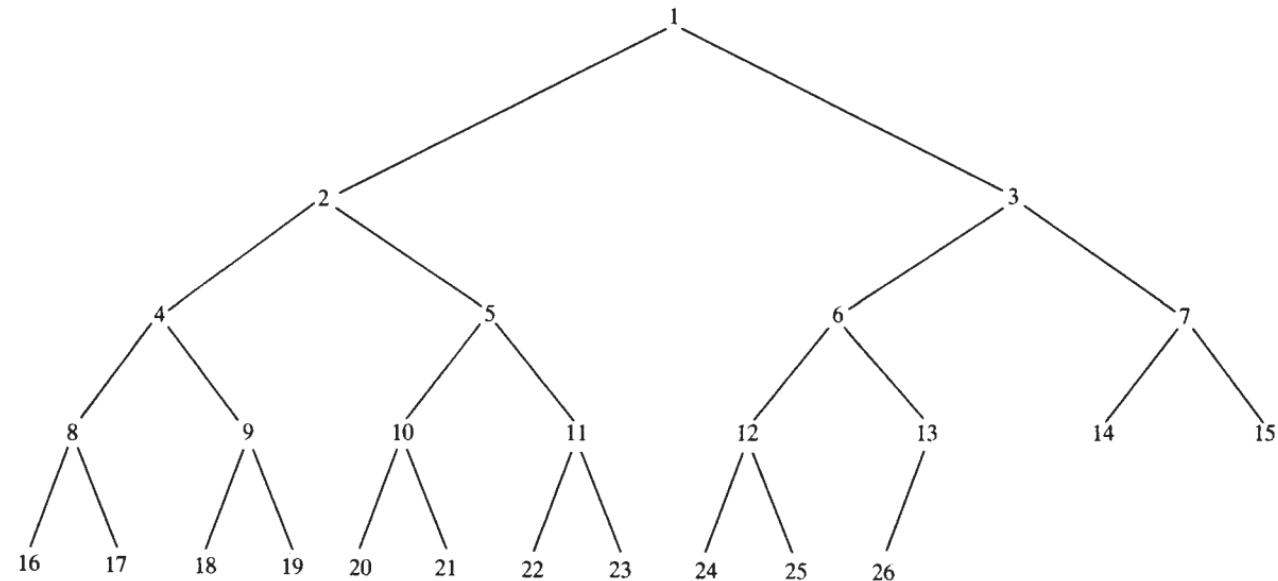
Os nós da árvore binária completa  $T_{26}$  foram intencionalmente rotulados pelos números 1, 2, ..., 26, da esquerda para a direita, geração por geração.

Com esses rótulos, pode-se facilmente determinar os filhos e o pai de cada nó K em qualquer árvore completa  $T_n$ .

De modo específico, os filhos à esquerda e à direita do nó K são, respectivamente,  $2*K$  e  $2*K + 1$ , e o pai de K é o nó  $[K/2]$ .

Por exemplo, os filhos do nó 9 são os nós 18 e 19, e seu pai é o nó  $[9/2] = 4$ .

A profundidade  $d_n$  da árvore completa  $T_n$  com n nós é dada por (aproximadamente):



$$d_n = \lfloor \log_2 n + 1 \rfloor$$

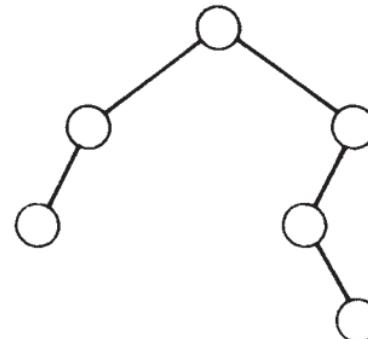
# Árvores binárias

## Árvore binárias estendidas: 2-árvores

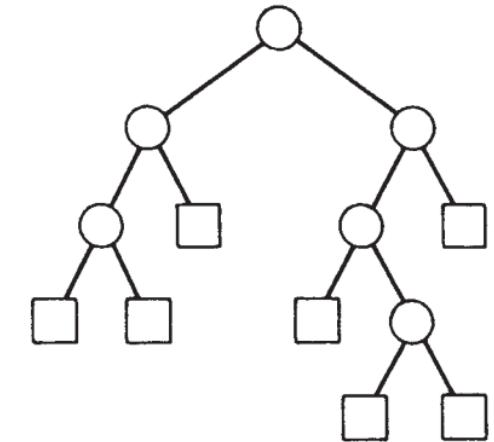
Uma árvore binária T é dita uma 2-árvore ou uma árvore binária estendida se cada nó N admite 0 ou 2 filhos.

Em tal caso, os nós com dois filhos são chamados de internos, e aqueles com 0 filhos são os externos.

Às vezes os nós são distinguidos em diagramas, usando círculos para os internos e quadrados para os externos.



(a) Árvore binária  $T$ .



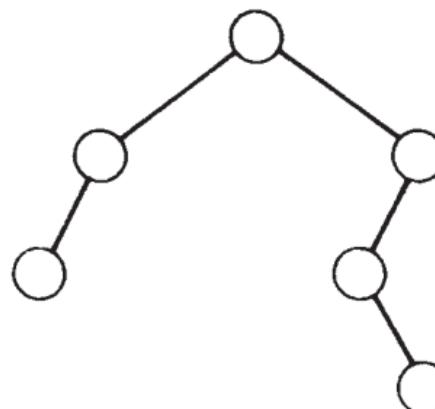
(b) 2-Árvore estendida

# Árvores binárias

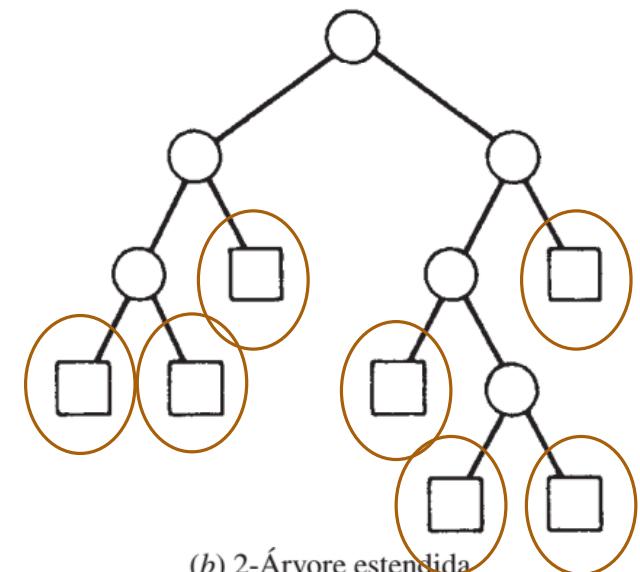
## Árvore binárias estendidas: 2-árvores

O termo “árvore binária estendida” surge da seguinte operação:

- Considere qualquer árvore binária T
- Então T pode ser “convertida” em uma 2-árvore, substituindo cada subárvore vazia por um novo nó
- Observe que a nova árvore é, de fato, uma 2-árvore.
- Além disso, os nós na árvore original T são agora os nós internos na árvore estendida, e os novos nós são os externos.
- Notamos que se uma 2-árvore tem n nós internos, então ela terá n + 1 nós externos.



(a) Árvore binária T.



(b) 2-Árvore estendida

# Árvores binárias

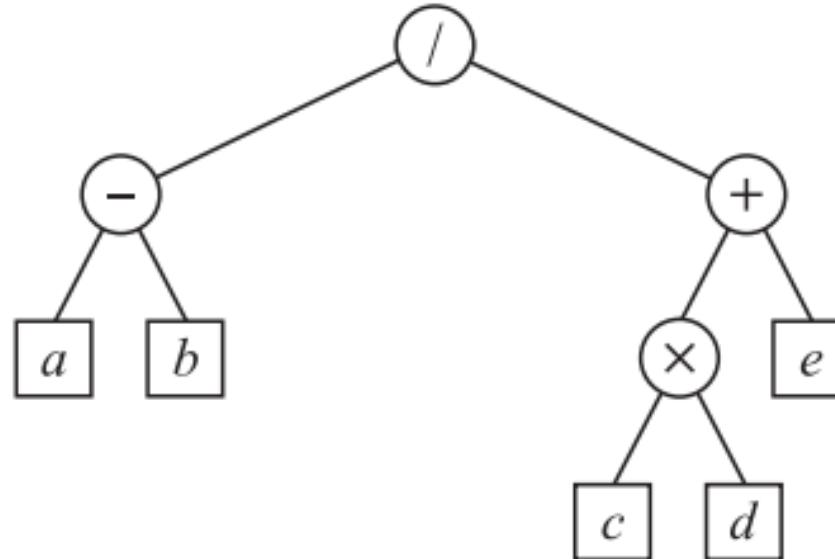
---

Expressões algébricas

Seja E uma expressão algébrica qualquer que emprega somente operações binárias, como:

$$E = (a - b) / ((c \times d) + e)$$

Pode ser representada por uma 2-árvore,



# Árvores binárias

---

## Notação Polonesa

O matemático polonês Lukasiewicz observou que, colocando a operação binária antes de seus argumentos, por exemplo:

$+ab$  no lugar de  $a + b$  e  $/cd$  no lugar de  $c/d$

não há necessidade de usar quaisquer parênteses.

Essa é a chamada de notação polonesa na forma prefixa. (Analogamente, podemos colocar o símbolo após seus argumentos, o que corresponde à notação polonesa na forma pós-fixa.)

Reescrevendo E na forma prefixa, obtemos:  $E = / - a b + \times c d e$

# Árvores binárias

---

Notação Polonesa

$$E = (a - b) / ((c \times d) + e)$$

$$E = / - a b + \times c d e$$

# Árvores binárias

---

## Representação ligada de árvores binárias

Considere uma árvore binária T. A menos que seja dito ou sugerido o contrário, T é mantida na memória por meio de uma representação ligada que emprega três arrays paralelos, INFO, LEFT, RIGHT, e uma variável apontadora ROOT

Cada nó N de T corresponde a uma localização K, de modo que:

- (1) INFO[K] contém os dados do nó N.
- (2) LEFT[K] contém a localização do filho à esquerda do nó N.
- (3) RIGHT[K] contém a localização do filho à direita do nó N.

Além disso, ROOT contém a localização da raiz R de T. Se qualquer subárvore for vazia, então o apontador correspondente contém o valor nulo; se a árvore T em si for vazia, então ROOT contém o valor nulo.

# Árvores binárias

## Representação ligada de árvores binárias

Exemplo:

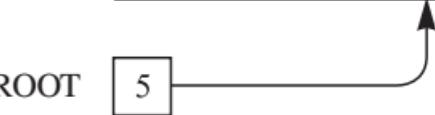
Considere a árvore binária T.

A representação ligada de T aparece na Figura,

Observe que  $\text{ROOT} = 5$  aponta para  $\text{INFO}[5] = A$ , uma vez que A é a raiz de T.

Note também que  $\text{LEFT}[5] = 10$  aponta para  $\text{INFO}[10] = B$ , pois B é o filho à esquerda de A, e  $\text{RIGHT}[5] = 2$  aponta para  $\text{INFO}[2] = C$ , uma vez que C é o filho à direita de A, e assim por diante.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
INFO	K	C	G		A	H	L			B		F	E			J	D	
LEFT	0	3	0		10	16	0			17		0	12			7	0	
RIGHT	0	6	0		2	1	0			13		0	0			0	0	

ROOT  5

# Árvores binárias

---

## Representação ligada de árvores binárias

```
6 references
class Nodo
{
    1 reference
    public int Key { get; set; }
    0 references
    public Nodo Left { get; set; }
    0 references
    public Nodo Right { get; set; }

    2 references
    public Nodo(int key)
    {
        this.Key = key;
    }
}
```

- Nesse caso a identificação o Nó raiz deverá ser algo a ser previamente definido.
- Os nós folhas serão aqueles que não tem nós nem a esquerda ou a direita
- E, a navegação pela árvore será feita de forma recursiva do nó raiz até o nível folha

# Árvores binárias

---

## Representação ligada de árvores binárias

```
static void Main(string[] args)
{
    Nodo n1 = new Nodo(1);
    Nodo n2 = new Nodo(2);
    Nodo raiz = new Nodo(0)
    {
        Left = n1,
        Right = n2
    };
}
```

- Nesse caso a identificação o Nó raiz deverá ser algo a ser previamente definido.
- Os nós folhas serão aqueles que não tem nós nem a esquerda ou a direita
- E, a navegação pela árvore será feita de forma recursiva do nó raiz até o nível folha

```
3 references
static void PercorrerArvore(Nodo no)
{
    if (no == null)
        return;

    System.Console.WriteLine(no.Key);
    PercorrerArvore(no.Left);
    PercorrerArvore(no.Right);
}
```

# Árvores binárias

---

## Representação sequencial de árvores binárias

Suponha que  $T$  é uma árvore binária completa ou quase completa.

Então existe uma maneira eficiente para manter  $T$  na memória, conhecida como a representação sequencial de  $T$ .

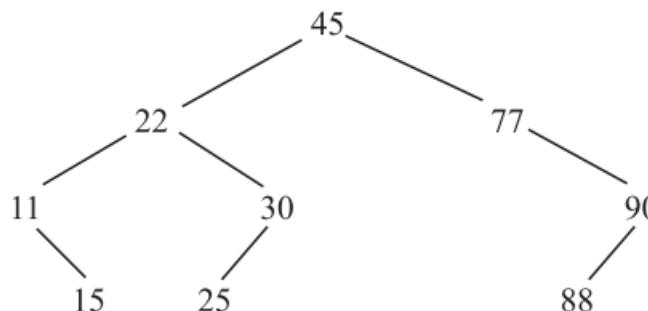
Tal representação usa somente um array linear  $\text{TREE}$  em parceria com uma variável apontadora  $\text{END}$ :

- (1) A raiz  $R$  de  $T$  é armazenada em  $\text{TREE}[1]$ .
- (2) Se um nó  $N$  ocupa  $\text{TREE}[K]$ , então seu filho à esquerda é armazenado em  $\text{TREE}[2*K]$  e o da direita é armazenado em  $\text{TREE}[2*K + 1]$ .
- (3)  $\text{END}$  contém a localização do último nó de  $T$ .

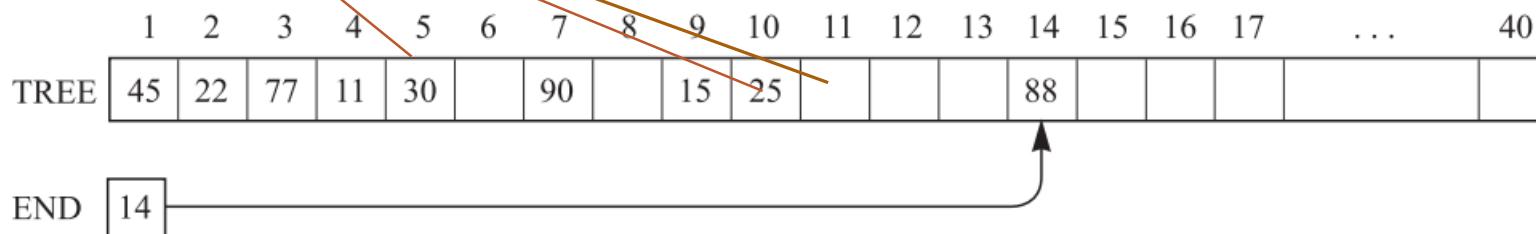
# Árvores binárias

## Representação sequencial de árvores binárias

Nessa representação quando um nó tem um filho, a posição do filho fica vazia



(a)



# Árvores binárias

---

## PERCORRENDO ÁRVORES BINÁRIAS

Existem três maneiras usuais de percorrer uma árvore  $T$  com raiz  $R$ . Esses três algoritmos, chamados de pré-ordem, inordem e pós-ordem são como se seguem:

- Pré-ordem:** (1) Processe a raiz  $R$ .  
(2) Percorra a subárvore à esquerda de  $R$  em pré-ordem.  
(3) Percorra a subárvore à direita de  $R$  em pré-ordem.

- Inordem:** (1) Percorra a subárvore à esquerda de  $R$  em inordem.  
(2) Processe a raiz  $R$ .  
(3) Percorra a subárvore à direita de  $R$  em inordem.

- Pós-ordem:** (1) Percorra a subárvore à esquerda de  $R$  em pós-ordem.  
(2) Percorra a subárvore à direita de  $R$  em pós-ordem.  
(3) Processe a raiz  $R$ .

# Árvores binárias

---

## PERCORRENDO ÁRVORES BINÁRIAS

**Pré-ordem:** (1) Processe a raiz  $R$ .

(2) Percorra a subárvore à esquerda de  $R$  em pré-ordem.

(3) Percorra a subárvore à direita de  $R$  em pré-ordem.

**Inordem:** (1) Percorra a subárvore à esquerda de  $R$  em inordem.

(2) Processe a raiz  $R$ .

(3) Percorra a subárvore à direita de  $R$  em inordem.

**Pós-ordem:** (1) Percorra a subárvore à esquerda de  $R$  em pós-ordem.

(2) Percorra a subárvore à direita de  $R$  em pós-ordem.

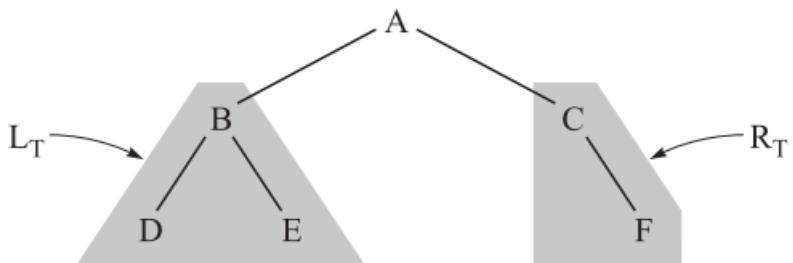
(3) Processe a raiz  $R$ .

- Note que cada algoritmo contém os mesmos três passos e que a subárvore à esquerda de  $R$  é sempre percorrida antes da subárvore à direita.
- A diferença entre os algoritmos é o momento em que a raiz é processada.
- Especificamente, no algoritmo “pré”, a raiz  $R$  é processada antes que as subárvores sejam percorridas;
- no algoritmo “in”, a raiz  $R$  é processada entre os percursos das subárvores;
- no algoritmo “pós”, a raiz  $R$  é processada depois que as subárvores foram percorridas.
- Os três algoritmos são, às vezes, chamados respectivamente de percursos nó-esquerda-direita (NLR), esquerda-nó-direita (LNR) e esquerda-direita-nó (LRN).

# Árvores binárias

---

Exemplo: Considere a árvore binária T. Observe que A é a raiz de T, a subárvore  $L_T$  à esquerda de T consiste nos nós B, D e E e que a subárvore  $R_T$  à direita de T é formada pelos nós C e F.

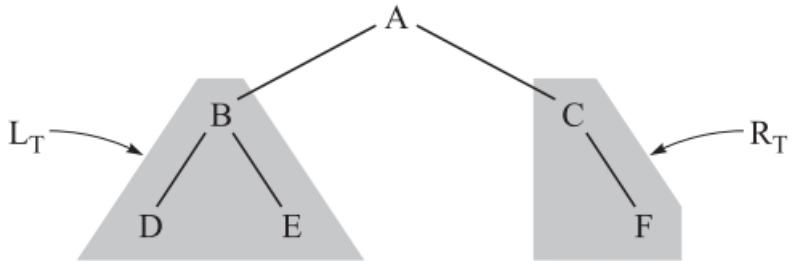


- O percurso pré-ordem de T processa A percorre  $L_T$  e percorre  $R_T$ .
- Porém, o percurso pré-ordem de  $L_T$  processa a raiz B e, em seguida, D e E;
- e o percurso pré-ordem de  $R_T$  processa a raiz C e então F.
- Assim, ABDECF é o percurso pré-ordem de T.

# Árvores binárias

---

Exemplo: Considere a árvore binária T. Observe que A é a raiz de T, a subárvore  $L_T$  à esquerda de T consiste nos nós B, D e E e que a subárvore  $R_T$  à direita de T é formada pelos nós C e F.

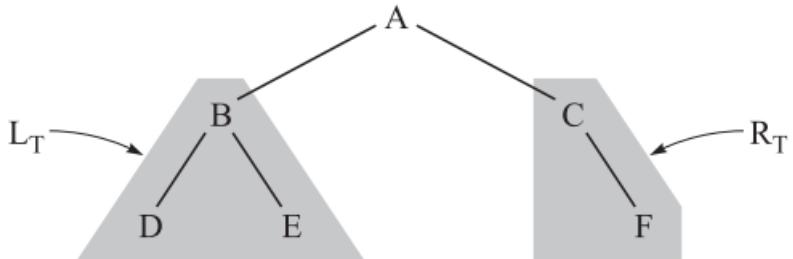


- O percurso inordem de T percorre  $L_T$ , processa A e percorre  $R_T$ .
- No entanto, o percurso inordem de  $L_T$  processa D, B e, em seguida, E;
- E o percurso inordem de  $R_T$  processa C e depois F.
- Desse modo, DBEACF é o percurso inordem de T.

# Árvores binárias

---

Exemplo: Considere a árvore binária T. Observe que A é a raiz de T, a subárvore  $L_T$  à esquerda de T consiste nos nós B, D e E e que a subárvore  $R_T$  à direita de T é formada pelos nós C e F.



- O percurso pós-ordem de T percorre  $L_T$  e processa A.
- Contudo, o percurso pós-ordem de  $L_T$  processa D, E e depois B, e o percurso pós-ordem de  $R_T$  processa F e, em seguida, C.
- Consequentemente, DEBFCA é o percurso pós-ordem de T.

# Árvores binárias

---

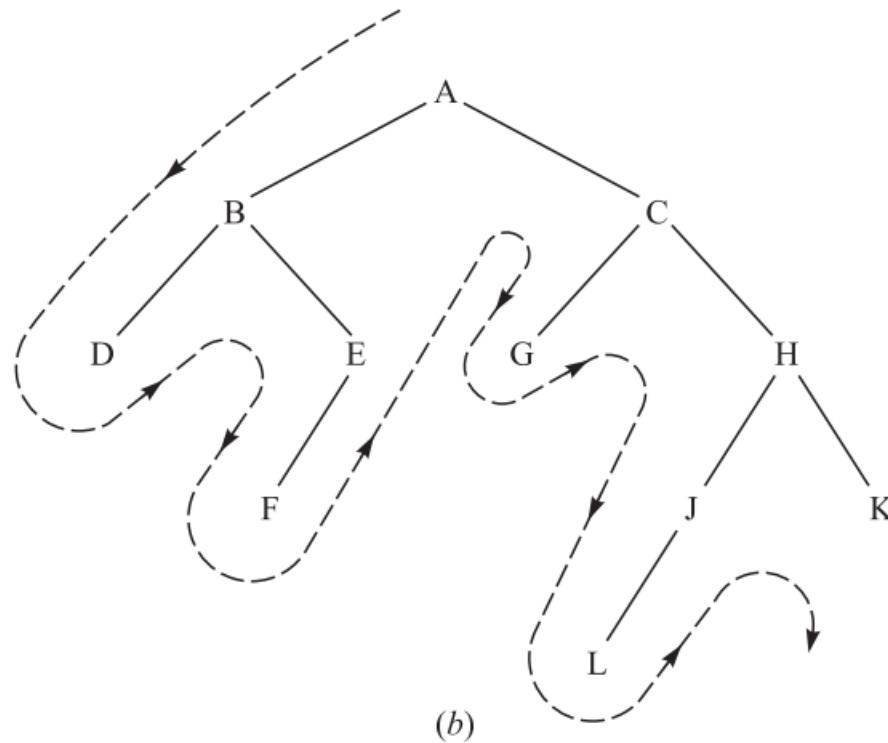
Seja T a árvore binária.

O percurso é como se segue:

(pré-ordem )A B D E F C G H J L K

(Inordem) D B F E A G C L J H K

(Pós-ordem) D F E B G L J K H C A

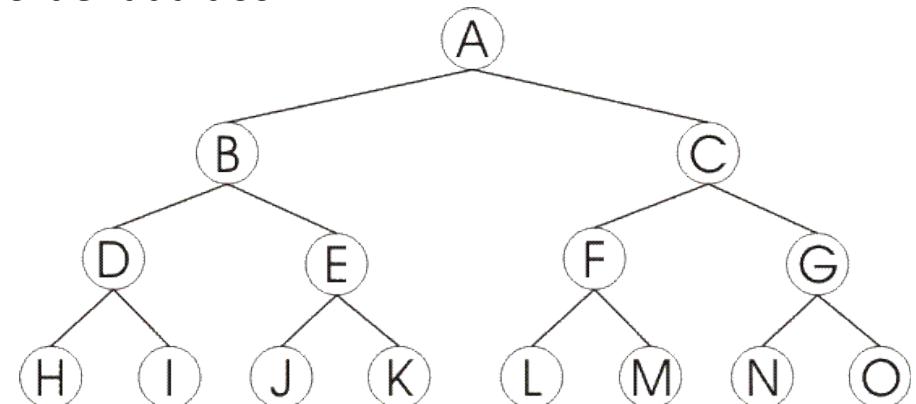
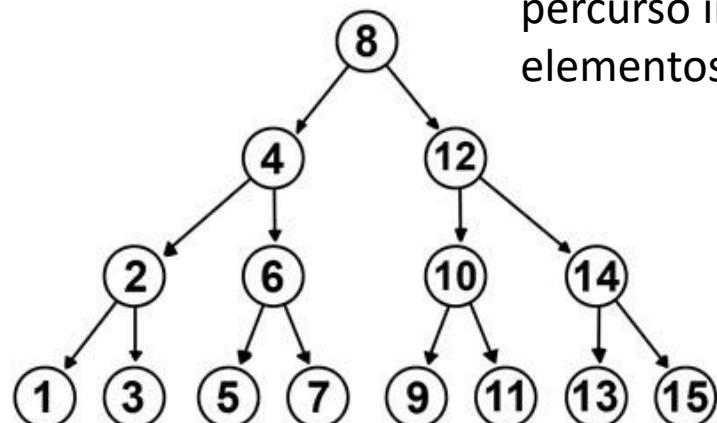


# ÁRVORES BINÁRIAS DE BUSCA

**Definição:** Suponha que T é uma árvore binária. Então T é chamada de árvore binária de busca se cada nó N de T tem a seguinte propriedade:

O valor de N é maior do que todo valor na subárvore à esquerda de N e é menor do que todo valor na subárvore à direita de N.

Não é difícil perceber que a propriedade acima garante que o percurso inordem de T nos leva a uma lista ordenada dos elementos de T.



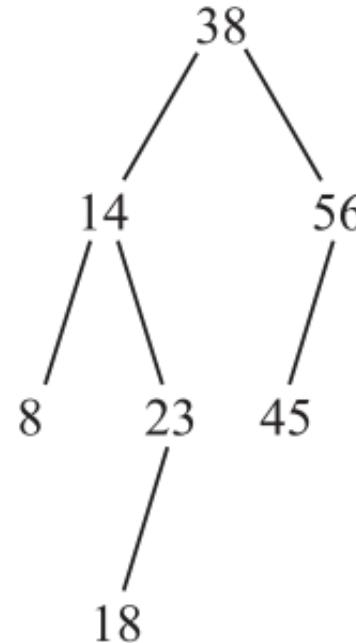
# ÁRVORES BINÁRIAS DE BUSCA

---

**Exemplo:** A árvore binária T é uma árvore binária de busca. Isto é, todo nó N em T excede cada número em subárvore à esquerda e é menor do que cada número em sua subárvore à direita.

Suponha que 23 fosse substituído por 35. Então T ainda seria uma árvore binária de busca.

Por outro lado, suponha que 23 fosse substituído por 40. Então T não seria uma árvore binária de busca, pois 40 estaria na subárvore de 38, porém,  $40 > 38$ .



# Buscando e inserindo em uma árvore binária de busca

---

Seja o Algoritmo:

**Algoritmo 10.1:** Uma árvore binária de busca  $T$  e um ITEM de informação são dados. O algoritmo encontra a localização de ITEM em  $T$ , ou insere ITEM como novo nó na árvore.

**Passo 1.** Compare ITEM com a raiz  $N$  da árvore.

- (a) Se  $\text{ITEM} < N$ , proceda para o filho à esquerda de  $N$ .
- (b) Se  $\text{ITEM} > N$ , proceda para o filho à direita de  $N$ .

**Passo 2.** Repita o Passo 1 até ocorrer o seguinte:

- (a) Encontramos um nó  $N$  tal que  $\text{ITEM} = N$ . Neste caso, a busca é bem-sucedida.
- (b) Encontramos uma subárvore vazia, o que indica que a busca é malsucedida. Insira ITEM no lugar da subárvore vazia.

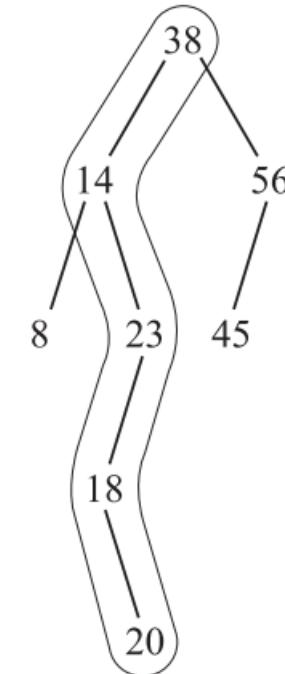
**Passo 3.** Saída.

# Buscando e inserindo em uma árvore binária de busca

---

**Exemplo:** Considere a árvore binária de busca T. Suponha que ITEM = 20 seja dado, e queremos encontrar ou inserir ITEM em T:

- (1) Compare ITEM = 20 com raiz R = 38. Como  $20 < 38$ , procedemos para o filho à esquerda de 38, que é 14.
- (2) Compare ITEM = 20 com 14. Como  $20 > 14$ , procedemos para o filho à direita de 14, que é 23.
- (3) Compare ITEM = 20 com 23. Como  $20 < 23$ , proceda para o filho à esquerda de 23, que é 18.
- (4) Compare ITEM = 20 com 18. Como  $20 > 18$  e 18 não tem filho à direita, insira 20 como o filho à direita de 18.



# Deletando em uma árvore binária de busca

---

**Algoritmo 10.2:** Uma árvore binária de busca  $T$  e um ITEM de informação são dados.  $P(N)$  denota o pai de um nó  $N$ , e  $S(N)$  corresponde ao sucessor inordem de  $N$ . O algoritmo deleta ITEM de  $T$ .

**Passo 1.** Use Algoritmo 10.1 para encontrar a localização do nó que contém ITEM e rastreie a localização do nó pai  $P(N)$ . (Se ITEM não estiver em  $T$ , então pare (STOP) e vá para a saída.)

**Passo 2.** Determine o número de filhos de  $N$ . Há três casos:

(a)  **$N$  não tem filhos.**  $N$  é deletado de  $T$ , simplesmente substituindo a localização de  $N$  no nó pai  $P(N)$  pelo apontador NULL.

(b)  **$N$  tem exatamente um filho  $M$ .**  $N$  é deletado de  $T$ , substituindo a localização de  $N$  no nó pai  $P(N)$  pela localização de  $M$ . (Isso troca  $N$  por  $M$ .)

(c)  **$N$  tem dois filhos.**

(i) Encontre o sucessor inordem  $S(N)$  de  $N$ . (Então  $S(N)$  não tem filho à esquerda.)

(ii) Delete  $S(N)$  de  $T$ , usando (a) ou (b).

(iii) Substitua  $N$  por  $S(N)$  em  $T$ .

**Passo 3.** Saída.

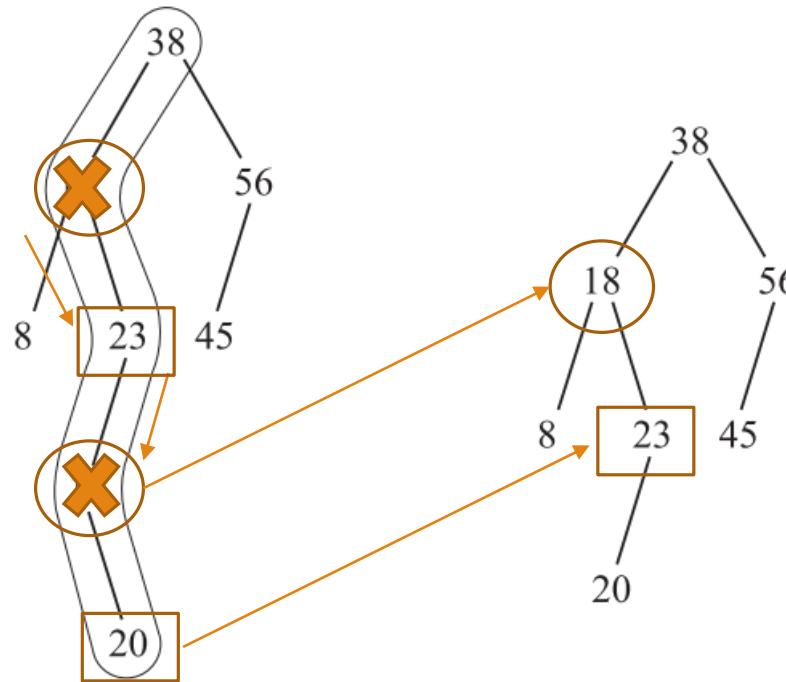
# Deletando em uma árvore binária de busca

**Exemplo** Considere a árvore.

Suponha que queremos deletar

ITEM = 14 de T.

- 1) Primeiro encontramos o nó N tal que  $N = 14$ . Note que  $N = 14$  tem dois filhos.
- 2) Movendo para a direita e então para esquerda, encontramos o sucessor inordem  $S(N) = 18$  de N.
- 3) Deletamos  $S(N) = 18$ , substituindo-o por seu único filho 20 e, então, substituímos N = 14 por  $S(N) = 18$ .



# FILAS DE PRIORIDADE, HEAPS

---

Seja  $S$  uma fila de prioridade.

$S$  é um conjunto cujos elementos podem ser periodicamente inseridos e deletados, mas no qual o maior elemento corrente (aquele com mais alta prioridade) é sempre deletado.

Pode-se manter  $S$  na memória como se segue:

- (a) Array linear: Aqui se pode facilmente inserir um elemento, apenas adicionando-o ao final do array. Contudo, é caro buscar e encontrar o maior elemento, uma vez que se deve usar uma busca linear
- (b) Array linear ordenado: Aqui o maior elemento é o primeiro ou o último e, assim, é facilmente deletado. No entanto, inserir e deletar elementos é caro

# FILAS DE PRIORIDADE, HEAPS

---

Suponha que  $H$  é uma árvore binária completa com  $n$  elementos. Assumimos que  $H$  é mantida na memória, usando sua representação sequencial e não uma representação ligada.

**Definição:** Suponha que  $H$  é uma árvore binária completa. Então  $H$  é chamada de heap, ou heap máximo, se cada nó tem a seguinte propriedade.

O valor de  $N$  é maior ou igual ao valor de cada um dos filhos de  $N$ .

Consequentemente, em um heap o valor de  $N$  excede o valor de cada um de seus descendentes. A raiz de  $H$ , em especial, é um valor maior de  $H$ . Um heap mínimo é definido de forma análoga: O valor de  $N$  é menor ou igual ao valor de cada um de seus filhos.

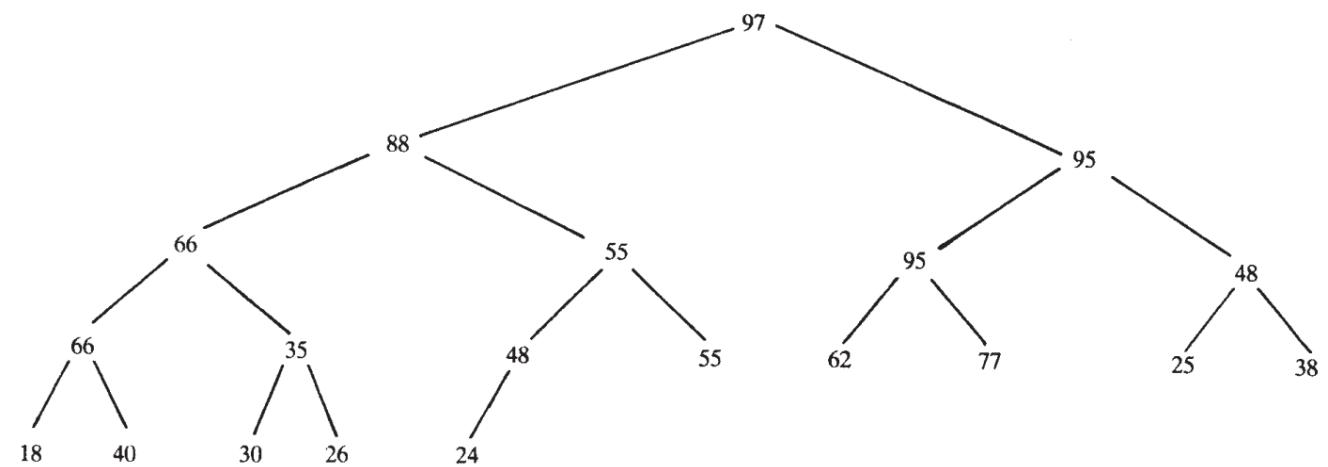
# FILAS DE PRIORIDADE, HEAPS

---

Considere a árvore binária completa H

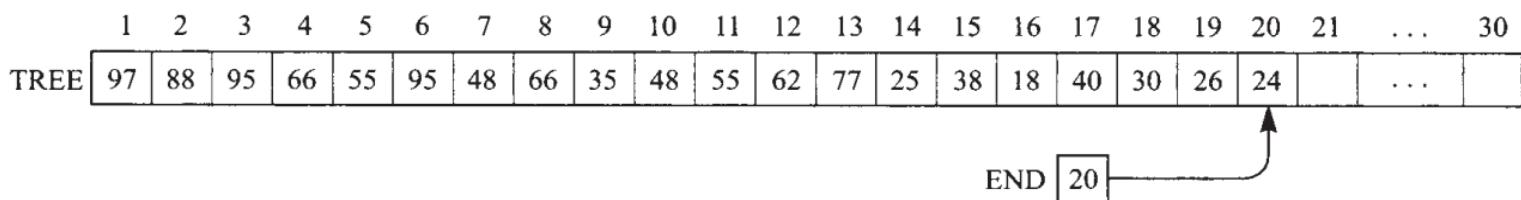
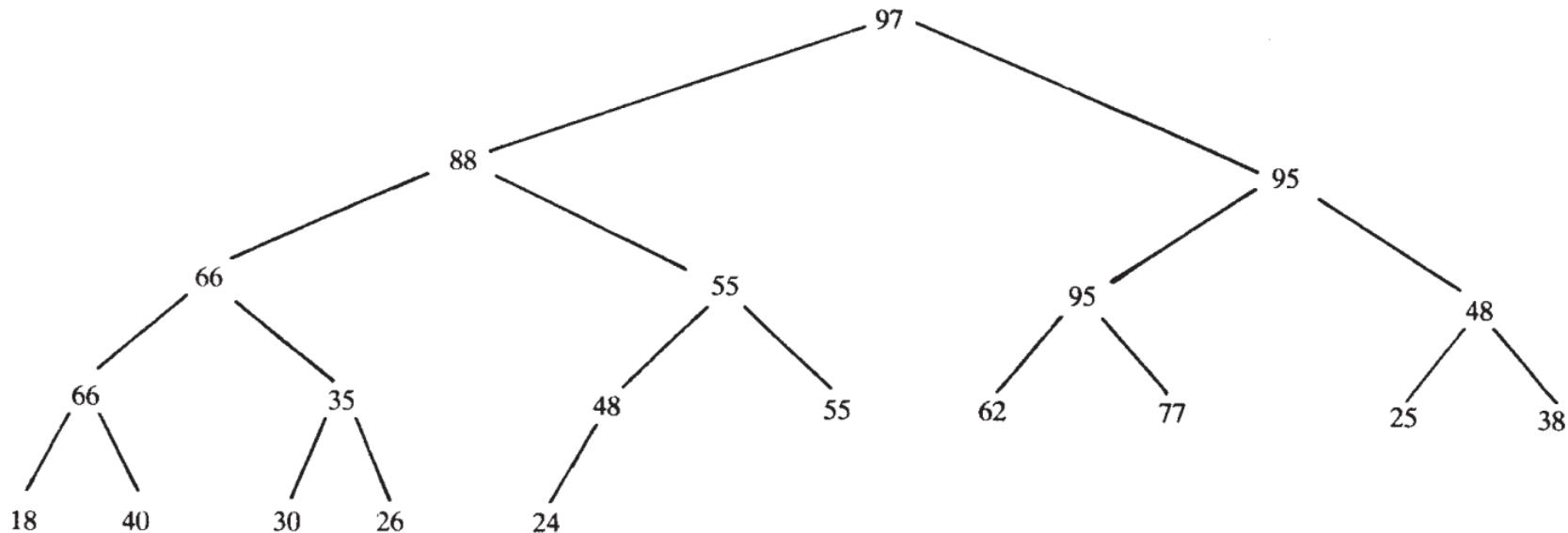
Observe que H é um heap. O maior elemento de H aparece no “topo” do heap:

- (a) TREE[1] é a raiz R de H.
- (b) TREE[2K] e TREE[2K + 1] são os filhos à esquerda e à direita de TREE[K].
- (c) A variável END = 20 aponta para o último elemento de H.
- (d) O pai de qualquer nó TREE(J) diferente da raiz é o nó TREE[J ÷ 2] (onde  $J \div 2$  é a divisão entre inteiros).



# FILAS DE PRIORIDADE, HEAPS

---



# FILAS DE PRIORIDADE, HEAPS

---

Inserindo um elemento em um Heap

**Algoritmo 10.3:** Um heap  $H$  e um novo ITEM são dados. O algoritmo insere ITEM em  $H$ .

- Passo 1.** Junte ITEM no final de  $H$ , de modo que  $H$  ainda seja uma árvore completa, mas não necessariamente um heap.
- Passo 2.** (Refaça o heap) Deixe ITEM subir para seu “lugar apropriado” em  $H$ , de modo que  $H$  seja um heap. Isto é:
  - (a) Compare ITEM com seu pai  $P(\text{ITEM})$ . Se  $\text{ITEM} > P(\text{ITEM})$ , então permute ITEM e  $P(\text{ITEM})$ .
  - (b) Repita (a) até  $\text{ITEM} \leq P(\text{ITEM})$ .
- Passo 3.** Saída.

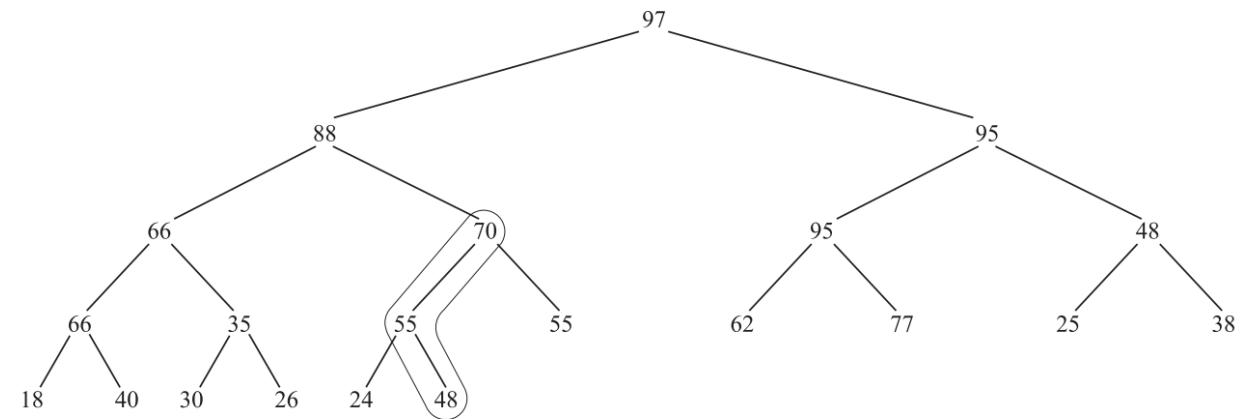
# FILAS DE PRIORIDADE, HEAPS

---

Em outras palavras, fazemos TREE[21] = 70 e END = 21.

Então refaça o heap, ou seja, deixamos ITEM subir para seu lugar adequado como se segue:

- (a) Compare ITEM = 70 com seu pai 48. Como  $70 > 48$ , permutamos 70 e 48.
- (b) Compare ITEM = 70 com seu novo pai 55. Como  $70 > 55$ , permutamos 70 e 55.
- (c) Compare ITEM = 70 com seu pai 88. Como  $70 < 88$ , ITEM = 70 subiu para seu lugar apropriado no heap H.



O caminho pela árvore por ITEM foi circundado.

# FILAS DE PRIORIDADE, HEAPS

---

## Deletando a raiz de um Heap

**Algoritmo 10.4:** O algoritmo deleta a raiz  $R$  de um dado heap  $H$ .

**Passo 1.** Assinale a raiz  $R$  a alguma variável ITEM.

**Passo 2.** Substitua a raiz deletada  $R$  pelo último nó de  $L$  de  $H$ , de modo que  $H$  ainda seja uma árvore binária completa, mas não necessariamente um heap. [Ou seja, faça  $\text{TREE}[1] := \text{TREE}[\text{END}]$  e então faça  $\text{END} := \text{END}-1$ .]

**Passo 3.** (Refaça o heap) Faça  $L$  cair para seu “lugar apropriado” em  $H$ , de modo que  $H$  seja um heap. Isto é:  
(a) Encontre o maior filho  $\text{LARGE}(L)$  de  $L$ . Se  $L < \text{LARGE}(L)$ , então permute  $L$  e  $\text{LARGE}(L)$ .  
(b) Repita (a) até  $L \geq \text{LARGE}(L)$ .

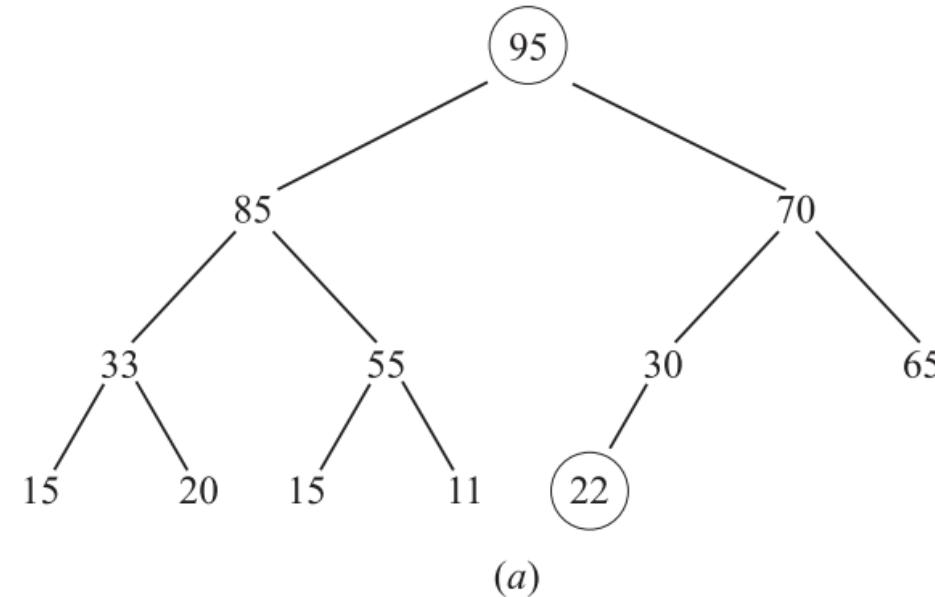
**Passo 4.** Saída.

# FILAS DE PRIORIDADE, HEAPS

---

Considere o heap H da figura (a), onde R = 95 é a raiz e L = 22 é o último nó de H.

Suponha que queremos deletar R = 95 do heap H.



# FILAS DE PRIORIDADE, HEAPS

---

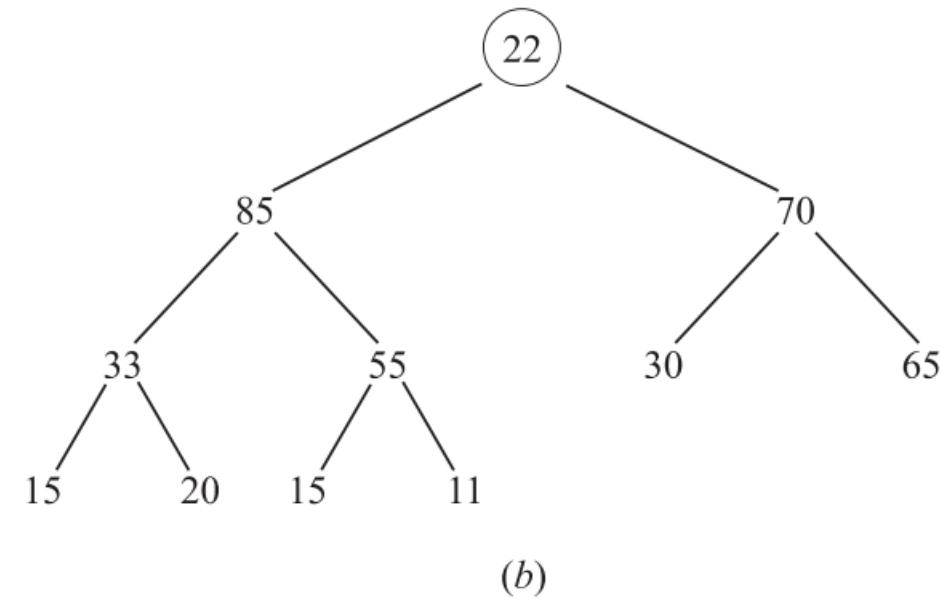
Considere o heap H da figura (a), onde R = 95 é a raiz e L = 22 é o último nó de H.

Suponha que queremos deletar R = 95 do heap H.

Primeiro “deletamos” R = 95, assinalando ITEM = 95, e então substituímos R = 95 por L = 22.

Isso nos leva à árvore completa da figura(b), que não é um heap. (Note que ambas as subárvores de 22 ainda são heaps.)

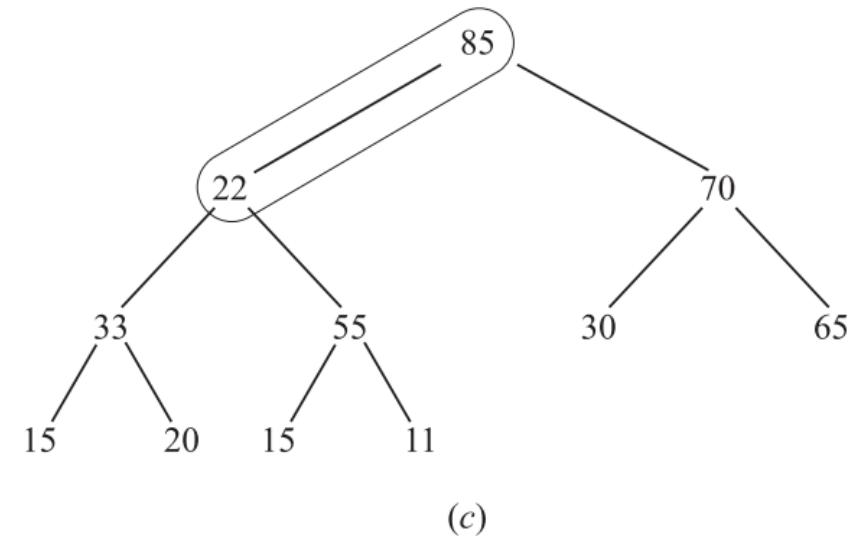
Em seguida, nós refazemos o heap, ou seja, fazemos L = 22 cair até seu lugar adequado como se segue:



# FILAS DE PRIORIDADE, HEAPS

---

(a) Os filhos de L = 22 são 85 e 70. O maior é 85. Como  $22 < 85$ , permutamos 22 e 85.  
Isso nos leva à árvore na figura(c).



# FILAS DE PRIORIDADE, HEAPS

---

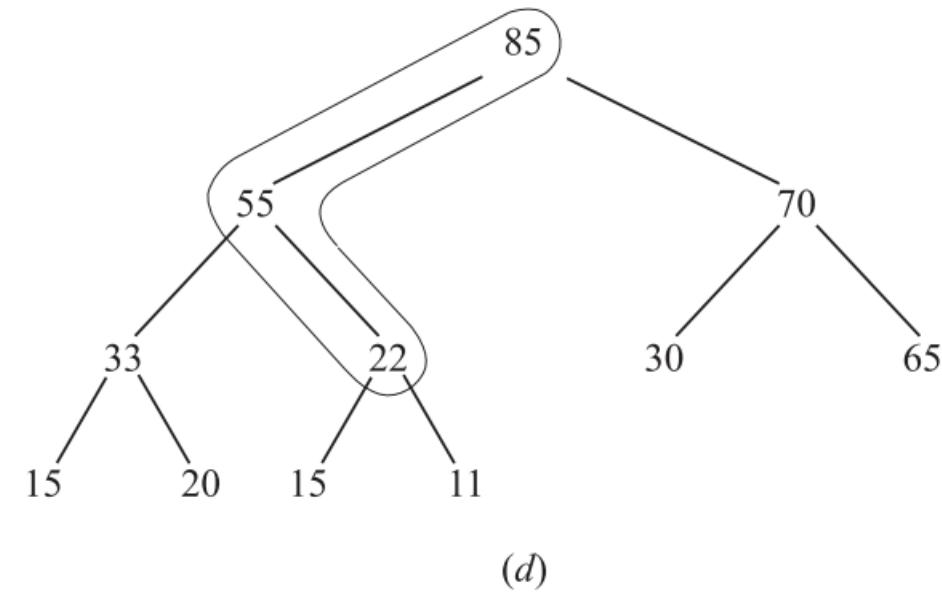
(a) Os filhos de  $L = 22$  são 85 e 70. O maior é 85. Como  $22 < 85$ , permutamos 22 e 85. Isso nos leva à árvore na figura(c).

(b) Os filhos de  $L = 22$  são agora 33 e 55. O maior é 55. Como  $22 < 55$ , permutamos 22 e 55. Isso nos leva à árvore na Figura(d).

(c) Os filhos de  $L = 22$  são agora 15 e 11. O maior é 15. Como  $22 > 15$ , o nó  $L = 22$  caiu para seu lugar apropriado no heap.

Assim, a Figura(d) é o heap H pedido sem sua raiz original  $R = 95$ .

Observe que destacamos os caminhos à medida que  $L = 22$  seguiu seu percurso para baixo na árvore.



# COMPRIMENTO DE CAMINHO, ALGORITMO DE HUFFMAN

---

## Comprimento de caminho ponderado

Seja  $T$  uma árvore binária estendida ou 2-árvore (Seção 10.3). Lembre que se  $T$  tem  $n$  nós externos, então  $T$  tem  $n - 1$  nós internos.

Suponha que  $T$  é uma 2-árvore com  $n$  nós externos e que cada um deles é assinalado a um peso (não negativo). O comprimento do caminho ponderado (ou simplesmente comprimento do caminho)  $P$  da árvore  $T$  é definido como a soma:

$$P = W_1L_1 + W_2L_2 + \cdots + W_nL_n$$

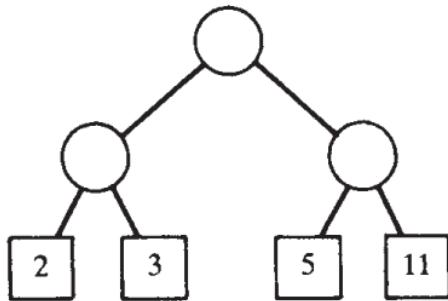
- Onde  $W_i$  é o peso em um nó externo  $N_i$ , e  $L_i$  é o comprimento do caminho da raiz  $R$  ao nó  $L_i$ .
- O comprimento do caminho  $P$  existe mesmo para 2-árvores não ponderadas, onde simplesmente se assume o peso 1 em cada nó externo

# COMPRIMENTO DE CAMINHO, ALGORITMO DE HUFFMAN

---

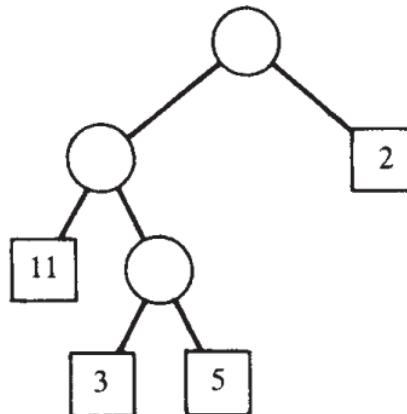
**Exemplo:** Sejam as 2-árvores

$$P_1 = 2(2) + 3(2) + 5(2) + 11(2) = 42$$

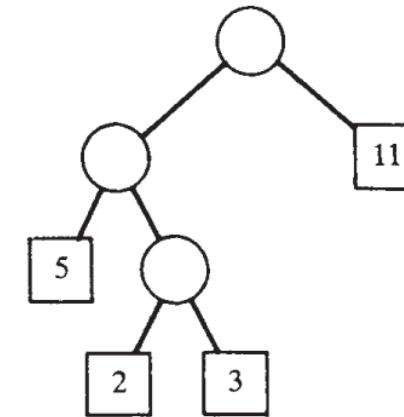


(a)  $T_1$

$$P_3 = 2(3) + 3(3) + 5(2) + 11(1) = 36$$



(b)  $T_2$



(c)  $T_3$

$$P_2 = 2(1) + 3(3) + 5(3) + 11(2) = 48$$

# ALGORITMO DE HUFFMAN

Suponha que uma lista de  $n$  pesos seja dada:  $W_1, W_2, \dots, W_n$

**Algoritmo 10.5 (Huffman):** O algoritmo encontra recursivamente uma 2-árvore ponderada  $T$  com  $n$  pesos dados  $w_1, w_2, \dots, w_n$  que tem um comprimento de caminho ponderado mínimo.

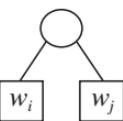
**Passo 1.** Suponha que  $n = 1$ . Seja  $T$  a árvore com um nó  $N$  com peso  $w_1$ , então vá para a Saída.

**Passo 2.** Suponha que  $n > 1$ .

- (a) Encontre dois pesos mínimos, digamos  $w_i$  e  $w_j$ , entre os  $n$  pesos dados.
- (b) Substitua  $w_i$  e  $w_j$  na lista por  $w_i + w_j$ , de modo que a lista tenha  $n - 1$  pesos.
- (c) Encontre uma árvore  $T'$  que fornece um comprimento de caminho ponderado mínimo para os  $n - 1$  pesos.
- (d) Na árvore  $T'$ , substitua o nó externo

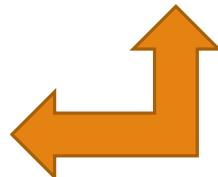
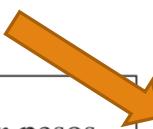
$$w_i + w_j$$

pela estrutura



- (e) Saída.

Entre todas as 2-árvore com  $n$  nós externos e com os  $n$  pesos dados, encontre uma árvore  $T$  com um comprimento de caminho ponderado mínimo. (Tal árvore é raramente única.)

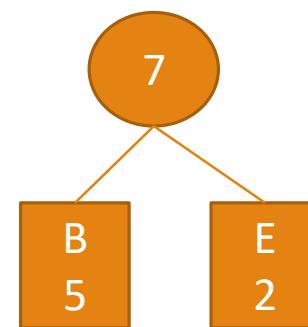


# ALGORITMO DE HUFFMAN

**Exemplo 10.12** Sejam  $A, B, C, D, E, F, G, H$  oito itens de dados com os seguintes pesos designados:

Item de dados:	$A$	$B$	$C$	$D$	$E$	$F$	$G$	$H$
Peso:	22	5	11	19	2	11	25	5

Passo	$A$	$B$	$C$	$D$	$E$	$F$	$G$	$H$
(1)	22	5	11	19	2	11	25	5
(2)	22		11	19	7	11	25	5

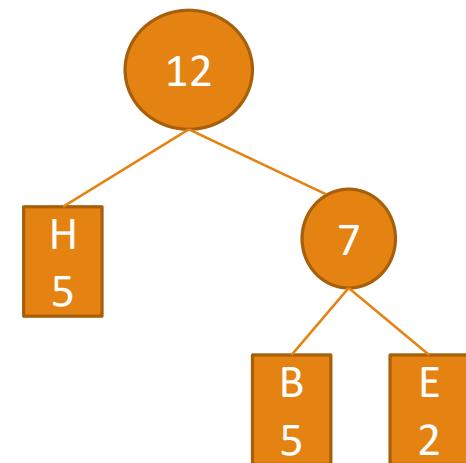


# ALGORITMO DE HUFFMAN

**Exemplo 10.12** Sejam  $A, B, C, D, E, F, G, H$  oito itens de dados com os seguintes pesos designados:

Item de dados:	$A$	$B$	$C$	$D$	$E$	$F$	$G$	$H$
Peso:	22	5	11	19	2	11	25	5

Passo	$A$		$C$	$D$		$F$	$G$	$H$
(2)	22		11	19	7	11	25	5
(3)	22		11	19		11	25	12

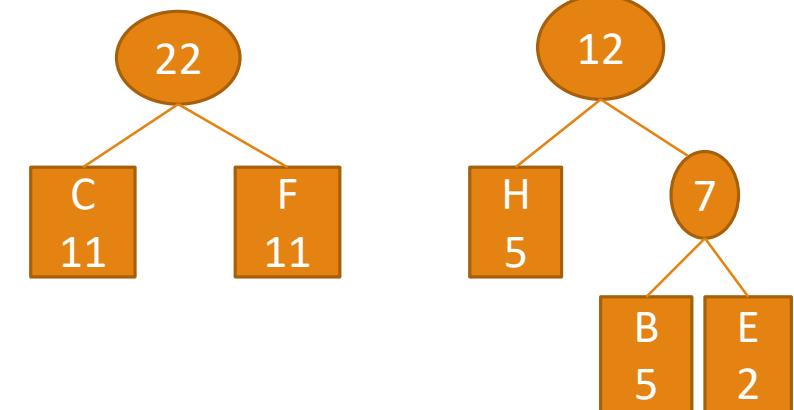


# ALGORITMO DE HUFFMAN

**Exemplo 10.12** Sejam  $A, B, C, D, E, F, G, H$  oito itens de dados com os seguintes pesos designados:

Item de dados:	$A$	$B$	$C$	$D$	$E$	$F$	$G$	$H$
Peso:	22	5	11	19	2	11	25	5

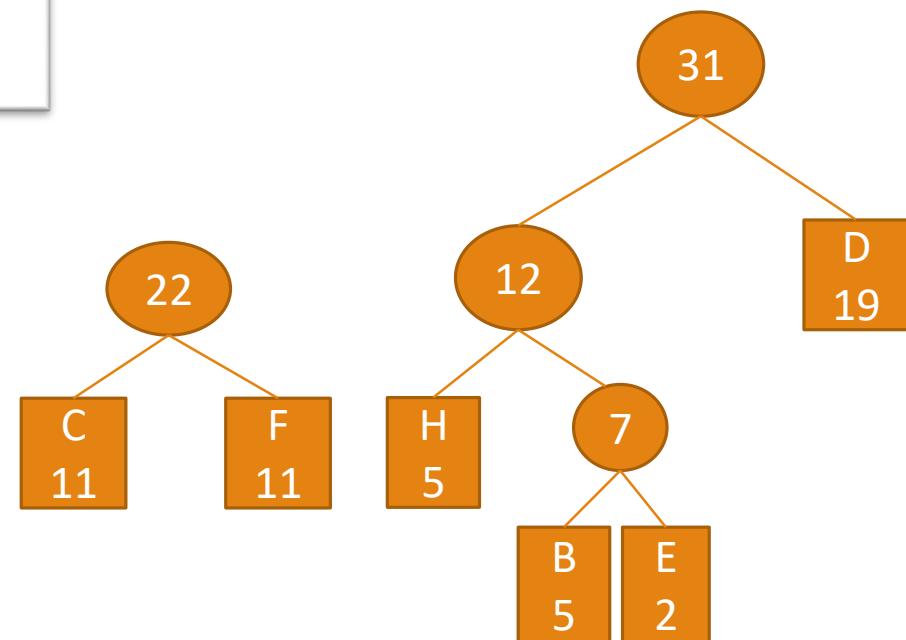
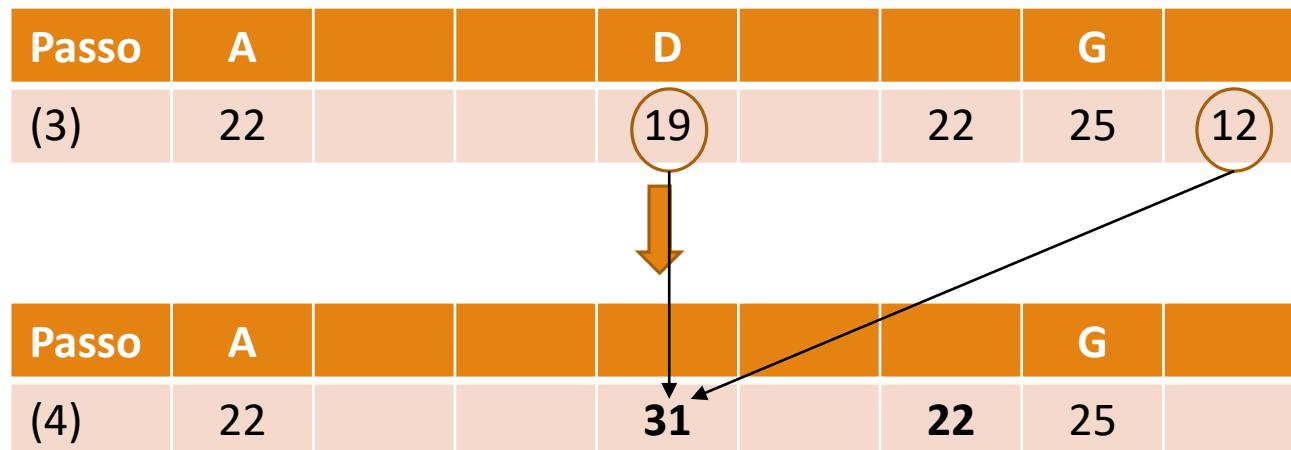
Passo	$A$		$C$	$D$		$F$	$G$	
(3)	22		11	19		11	25	12
(4)	22				19		25	12



# ALGORITMO DE HUFFMAN

**Exemplo 10.12** Sejam A, B, C, D, E, F, G, H oito itens de dados com os seguintes pesos designados:

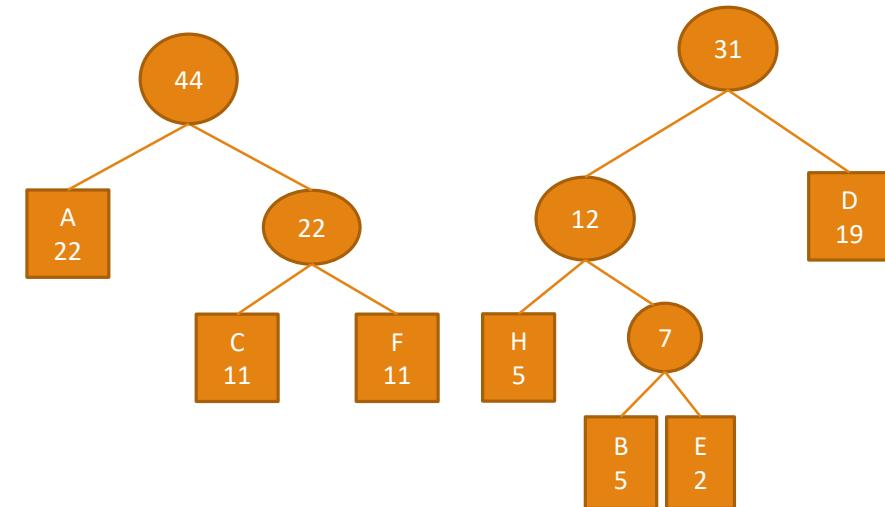
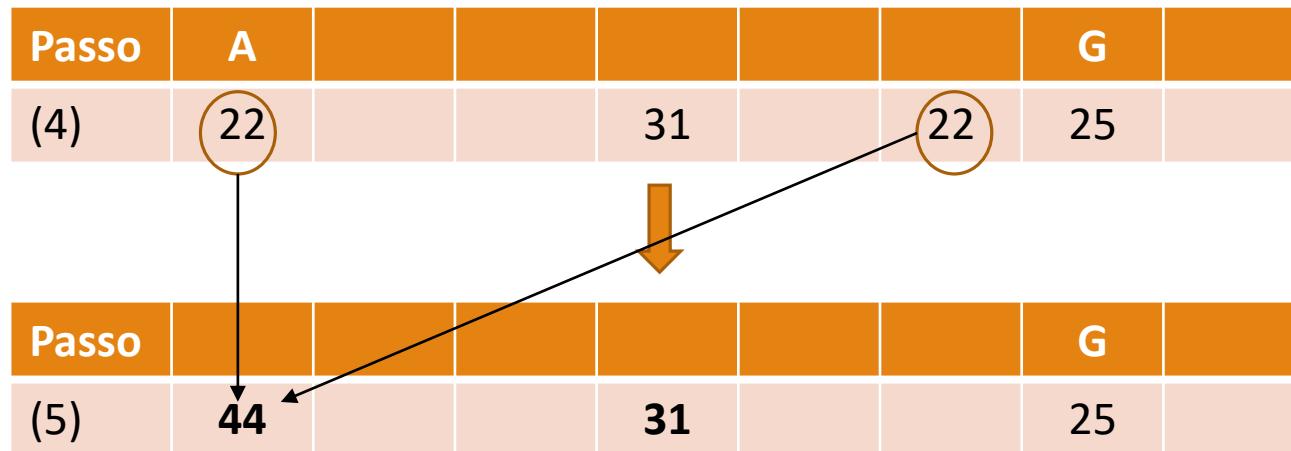
Item de dados:	A	B	C	D	E	F	G	H
Peso:	22	5	11	19	2	11	25	5



# ALGORITMO DE HUFFMAN

**Exemplo 10.12** Sejam A, B, C, D, E, F, G, H oito itens de dados com os seguintes pesos designados:

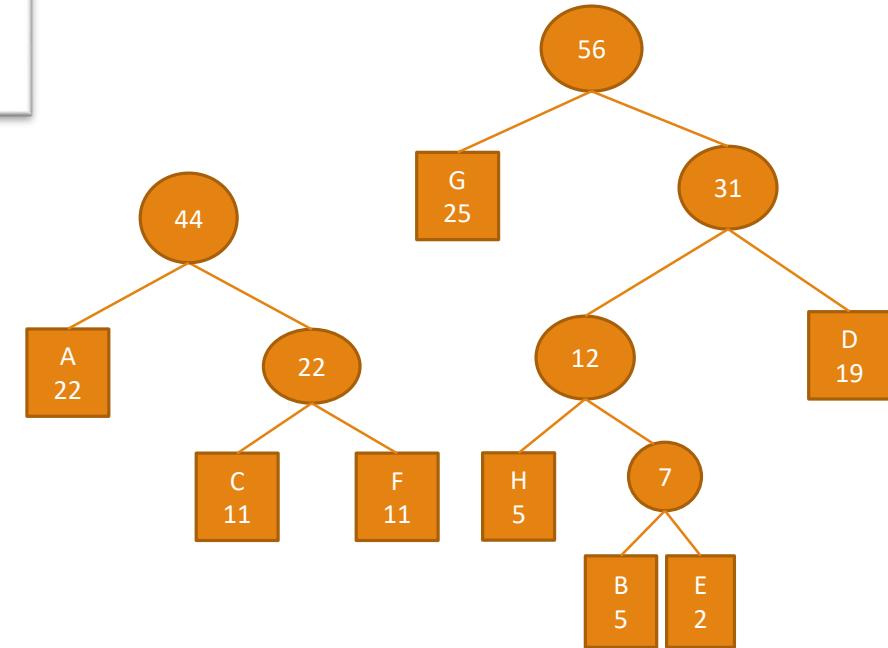
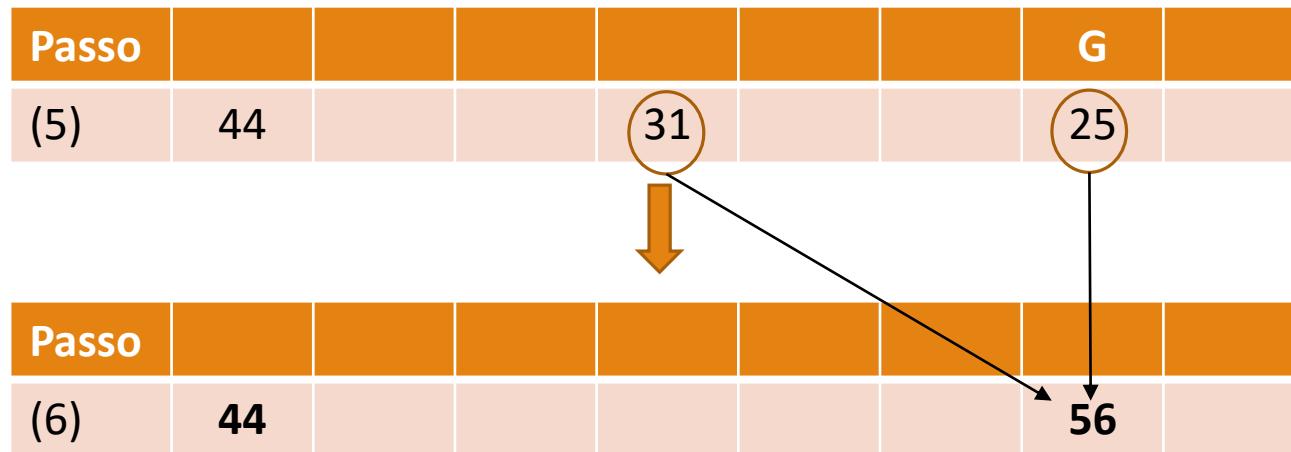
Item de dados:	A	B	C	D	E	F	G	H
Peso:	22	5	11	19	2	11	25	5



# ALGORITMO DE HUFFMAN

**Exemplo 10.12** Sejam  $A, B, C, D, E, F, G, H$  oito itens de dados com os seguintes pesos designados:

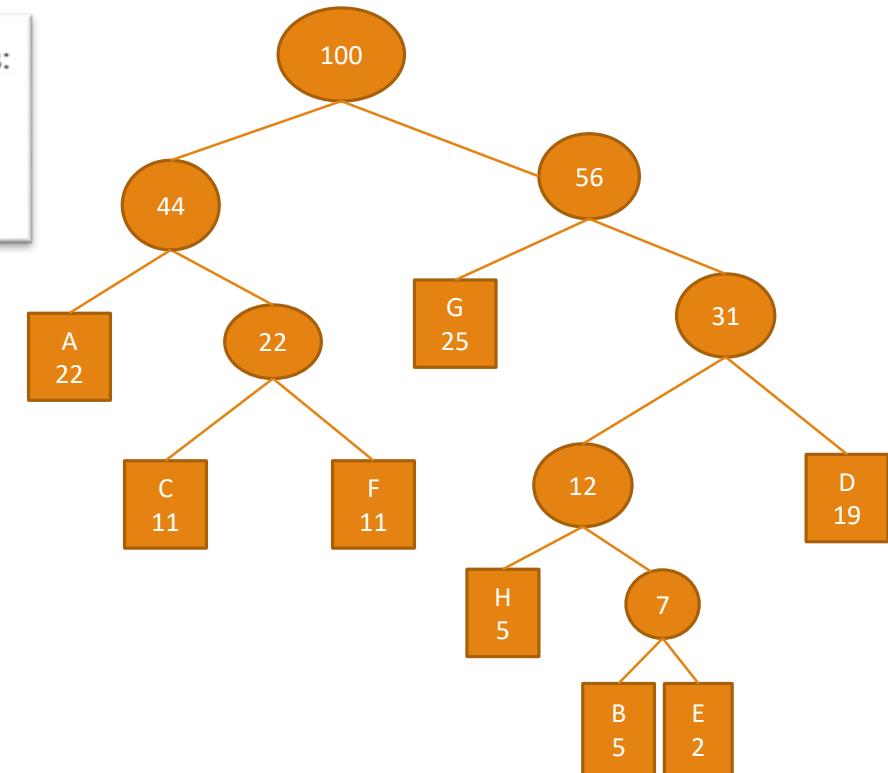
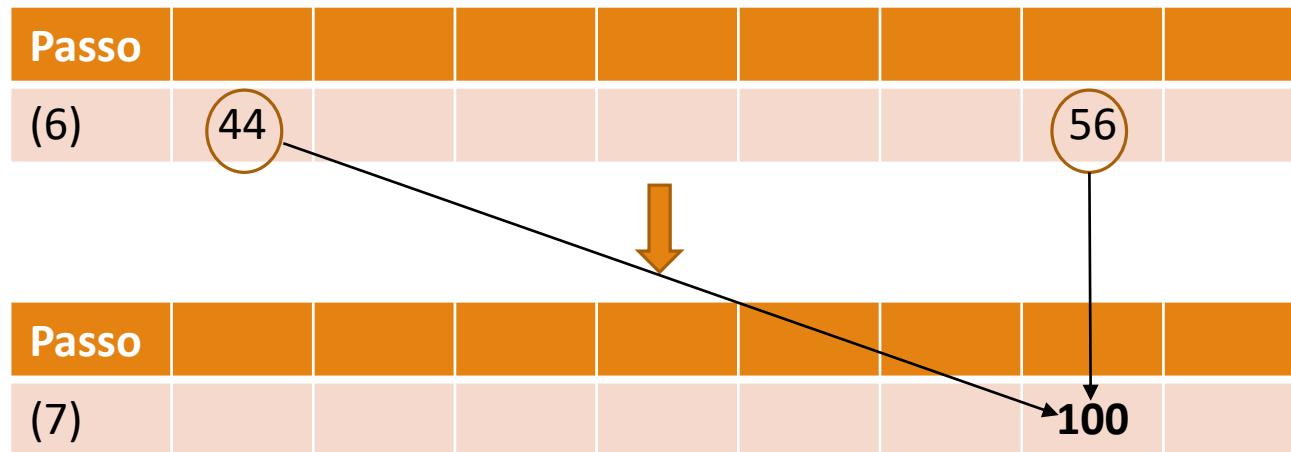
Item de dados:	$A$	$B$	$C$	$D$	$E$	$F$	$G$	$H$
Peso:	22	5	11	19	2	11	25	5



# ALGORITMO DE HUFFMAN

**Exemplo 10.12** Sejam  $A, B, C, D, E, F, G, H$  oito itens de dados com os seguintes pesos designados:

Item de dados:	$A$	$B$	$C$	$D$	$E$	$F$	$G$	$H$
Peso:	22	5	11	19	2	11	25	5



# Código de Huffman

---

Seja  $T$  a árvore de Huffman para os  $n$  itens de dados ponderados  $A_1, A_2, \dots, A_n$

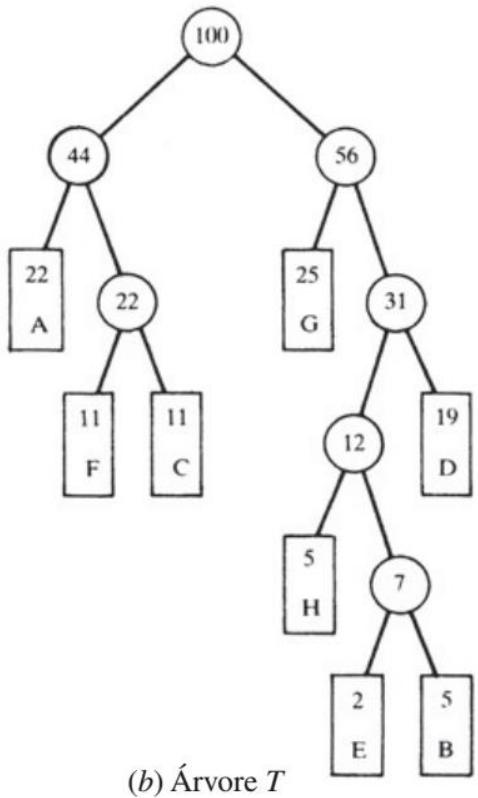
Cada aresta de  $T$  é assinalada 0 ou 1, dependendo se a aresta aponta para um filho à esquerda ou à direita.

A codificação de Huffman assinala a cada nó externo  $A_i$  a sequência de bits da raiz  $R$  da árvore ao nó  $A$ .

O código de Huffman citado tem a propriedade “prefixo”, ou seja, o código de qualquer item não é um substring inicial do código de qualquer outro item. Isso significa que não pode haver ambiguidade alguma na decodificação de qualquer mensagem usando um código de Huffman.

# Código de Huffman

---



A: 00  
B: 11011  
C: 011  
D: 111  
E: 11010  
F: 010  
G: 10  
H: 1100.

Por exemplo, para chegar a E, a partir da raiz, o caminho consiste em uma aresta à direita, aresta à direita, aresta à esquerda, aresta à direita e aresta à esquerda, levando ao código 11010 para E.

Busca em profundidade  
(DFS)

Busca em largura (BFS)

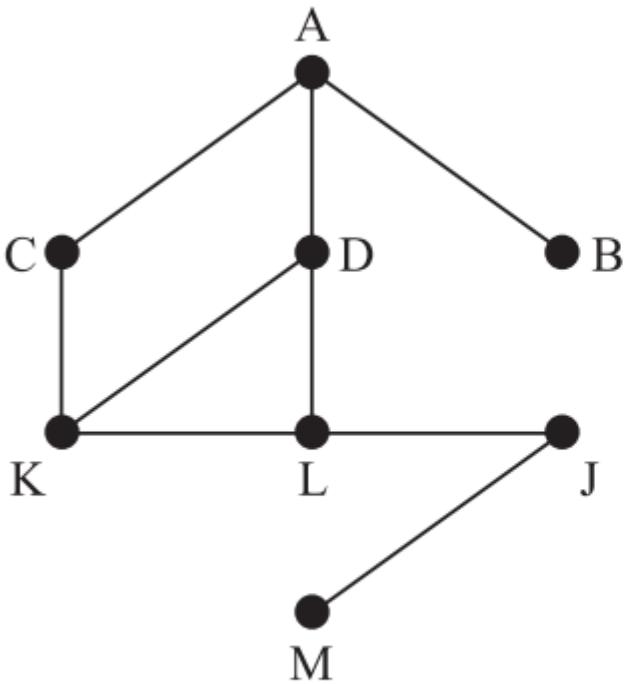
---

# Introdução

---

Qualquer algoritmo de grafos em particular pode depender da maneira como  $G$  é mantido na memória.

Aqui assumimos que  $G$  é mantido na memória por sua estrutura de adjacência.



(a)

Vértice	Lista de adjacência
A	B, C, D
B	A
C	A, K
D	A, K, L
J	L, M
K	C, D, L
L	D, J, K
M	J

(b)

# Introdução

---

Durante a execução de nosso algoritmo, cada vértice (nó)  $N$  de  $G$  está em um entre três estados, chamados de status de  $N$ , como se segue:

- STATUS = 1: (Estado pronto) O estado inicial do vértice  $N$ .
- STATUS = 2: (Estado de espera) O vértice  $N$  está em uma lista (de espera), aguardando para ser processado.
- STATUS = 3: (Estado processado) O vértice  $N$  foi processado.

A lista de espera para a busca em profundidade (DFS) é uma PILHA (modificada, a qual escrevemos horizontalmente com o topo da PILHA à esquerda), enquanto a lista de espera para a busca em largura (BFS) é uma FILA.

# Busca em profundidade (DFS)

---

## *Ideia geral:*

Primeiro, processamos o vértice inicial A.

Em seguida, processamos cada vértice N ao longo de um caminho P que começa em A; ou seja, processamos um vizinho de A, depois outro vizinho de A, e assim por diante.

Após chegar a um “beco sem saída”, isto é, a um vértice sem vizinho não processado, voltamos pelo caminho P até ser possível continuar por outro caminho P’, e assim por diante.

Usa-se uma PILHA para manter os vértices iniciais de possíveis caminhos futuros.

Um campo STATUS nos diz o atual estado de qualquer vértice, de modo que nenhum deles seja processado mais de uma vez.

# Busca em profundidade

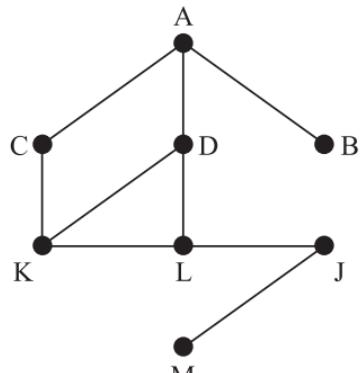
---

**Algoritmo 8.5 (busca em profundidade):** Esse algoritmo executa uma busca em profundidade sobre um grafo  $G$  começando com um vértice inicial  $A$ .

- Passo 1.** Inicialize todos os vértices com o estado pronto ( $\text{STATUS} = 1$ ).
- Passo 2.** Jogue o vértice inicial  $A$  na PILHA e mude o estado de  $A$  para o de espera ( $\text{STATUS} = 2$ ).
- Passo 3.** Repita os Passos 4 e 5 até a PILHA estar vazia.
- Passo 4.** Mova o vértice  $N$  do topo da PILHA. Processe  $N$  e faça  $\text{STATUS}(N) = 3$ , o estado processado.
- Passo 5.** Examine cada vizinho  $J$  de  $N$ .
  - (a) Se  $\text{STATUS}(J) = 1$  (estado pronto), jogue  $J$  em PILHA e mude  $\text{STATUS}(N) = 2$ .
  - (b) Se  $\text{STATUS}(J) = 2$  (estado de espera), delete o  $J$  anterior da PILHA e jogue o atual  $J$  na PILHA.
  - (c) Se  $\text{STATUS}(J) = 3$  (estado processado), ignore o vértice  $J$ .
- [Fim do ciclo do Passo 3.]
- Passo 6.** Saída.

# Busca em profundidade

Grafo Exemplo



(a)

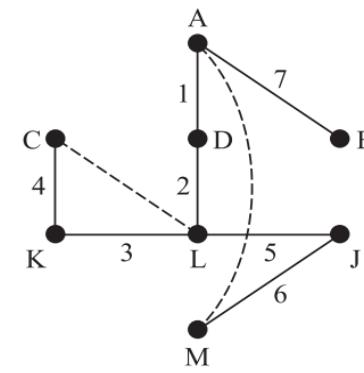
Vértice	Lista de adjacência
A	B, C, D
B	A
C	A, K
D	A, K, L
E	F, G
F	E, G, H
G	F, H
H	G, I
I	H, J
J	I, L
K	C, D, L
L	D, J, K
M	J

(b)

DFS

PILHA	Vértice
A	A
D, C, B	D
L, K, C, B	L
K, J, L, C, B	K
C, J, Ø, B	C
J, B	J
M, B	M
B	B
Ø	

(a)



(b)

Cada vértice, excluindo A, vem de uma lista de adjacência e, portanto, corresponde a uma aresta do grafo. Essas arestas formam uma árvore geradora de  $G$  que é representada na Fig.(b). Os números indicam a ordem que as arestas são adicionadas à árvore geradora, e as linhas tracejadas indicam voltas.

# Busca em largura (BFS)

---

Primeiro, processamos o vértice inicial.

Em seguida, processamos todos os vizinhos de A.

Depois, processamos todos os vizinhos dos vizinhos de A, e assim por diante.

Naturalmente, precisamos acompanhar os vizinhos de um vértice e garantir que nenhum vértice seja processado duas vezes:

- Usa-se uma FILA para manter os vértices que estão esperando para serem processados
- um campo STATUS que nos diz o atual estado de um vértice.

# Busca em largura (BFS)

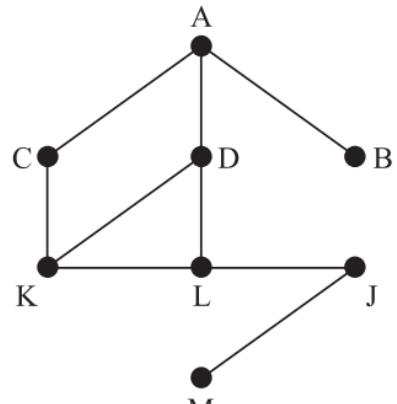
**Algoritmo 8.6 (busca em largura):** Esse algoritmo executa uma busca em largura em um grafo  $G$  começando com um vértice inicial  $A$ .

- Passo 1.** Inicialize todos os vértices com o estado pronto ( $\text{STATUS} = 1$ ).
- Passo 2.** Coloque o vértice inicial  $A$  em FILA e mude o estado de  $A$  para o modo de espera ( $\text{STATUS} = 2$ ).
- Passo 3.** Repita os Passos 4 e 5 até FILA estar vazia.
- Passo 4.** Remova o vértice da frente  $N$  de FILA. Processe  $N$  e faça  $\text{STATUS}(N) = 3$ , o estado processado.
- Passo 5.** Examine cada vizinho  $J$  de  $N$ .
  - (a) Se  $\text{STATUS}(J) = 1$  (estado pronto), adicione  $J$  para o final de FILA e redefina  $\text{STATUS}(J) = 2$  (estado de espera).
  - (b) Se  $\text{STATUS}(J) = 2$  (estado de espera) ou  $\text{STATUS}(J) = 3$  (estado processado), ignore o vértice  $J$ .

[Fim do ciclo do Passo 3.]
- Passo 6.** Saída.

# Busca em largura (BFS)

Grafo Exemplo



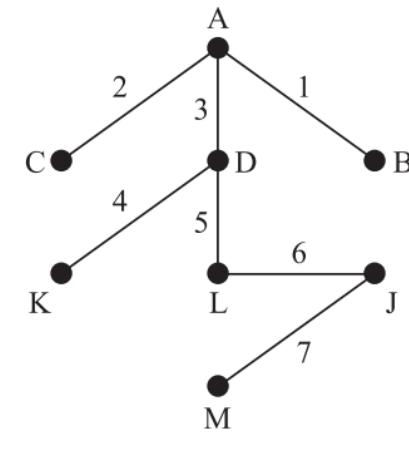
Vértice	Lista de adjacência
A	B, C, D
B	A
C	A, K
D	A, K, L
J	L, M
K	C, D, L
L	D, J, K
M	J

(b)

BFS

FILA	Vértice
A	A
D, C, B	B
D, C	C
D	D
L, K	K
L	L
J	J
M	M
∅	

(a)



Novamente, cada vértice, excluindo A, vem de uma lista de adjacência e, portanto, corresponde a uma aresta do grafo. Essas arestas formam uma árvore geradora de G que é retratada na (b). De novo, os números indicam a ordem em que as arestas são adicionadas à árvore geradora. Observe que essa árvore geradora é diferente daquela da (b) que veio de uma busca em profundidade.

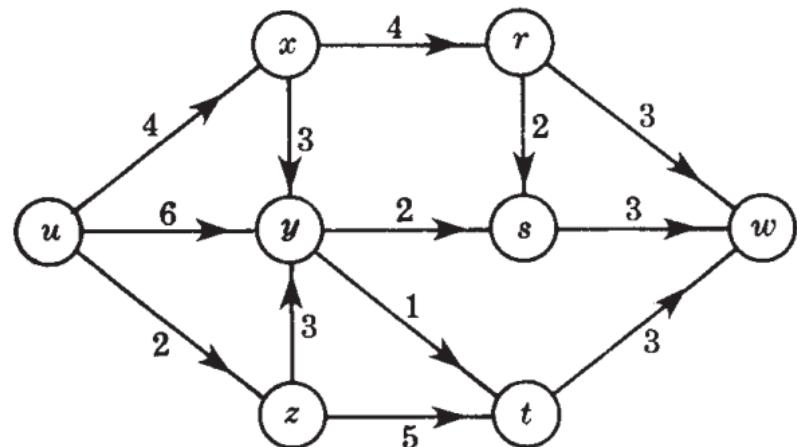
# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO

Seja  $G$  um grafo orientado livre de ciclos e ponderado.

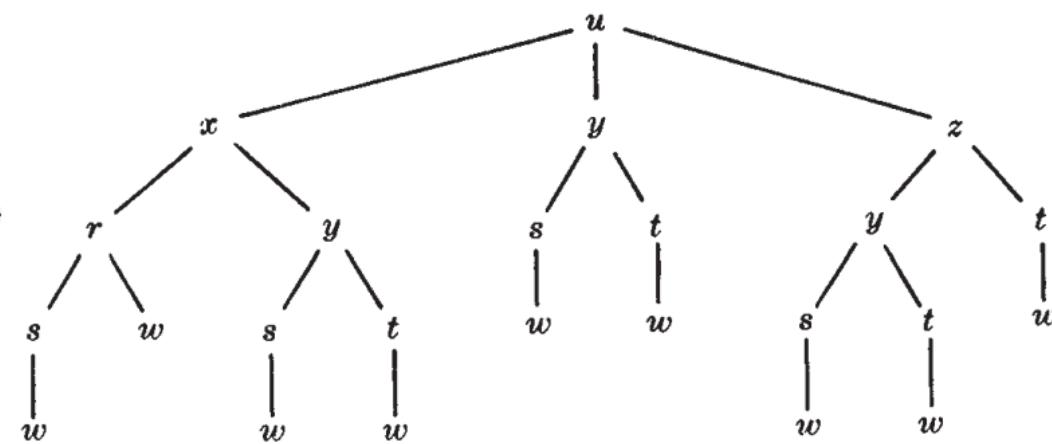
Buscamos pelo caminho mais curto entre dois vértices, digamos  $u$  e  $w$ .

$G$  é finito, de modo que em cada passo existe um número finito de movimentos.

$G$  é livre de ciclos, todos os caminhos entre  $u$  e  $w$  podem ser dados por uma árvore na qual  $u$  é a raiz.



(a)



(b)

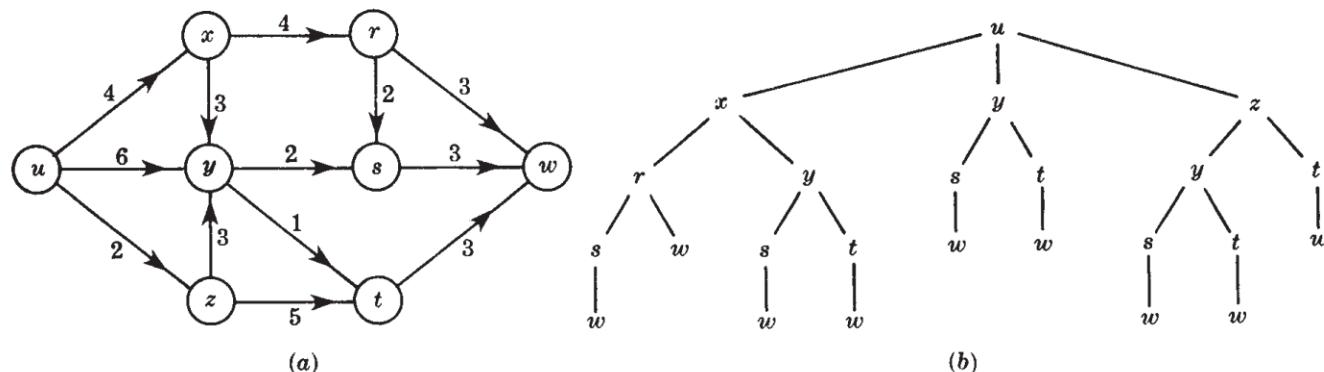
# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO

Encontrar o caminho mais curto entre  $u$  e  $w$  é simplesmente calculando os comprimentos de todos os caminhos da árvore enraizada correspondente.

Por outro lado, suponha que dois caminhos parciais conduzam a um vértice intermediário  $v$ .

Precisamos apenas considerar o caminho parcial mais curto;

Podamos a árvore no vértice correspondente ao caminho parcial mais longo.



# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO

---

Encontra o caminho mais curto entre um vértice  $u$  e um vértice  $w$  em um grafo  $G$  orientado livre de ciclos e ponderado. O algoritmo tem as seguintes propriedades:

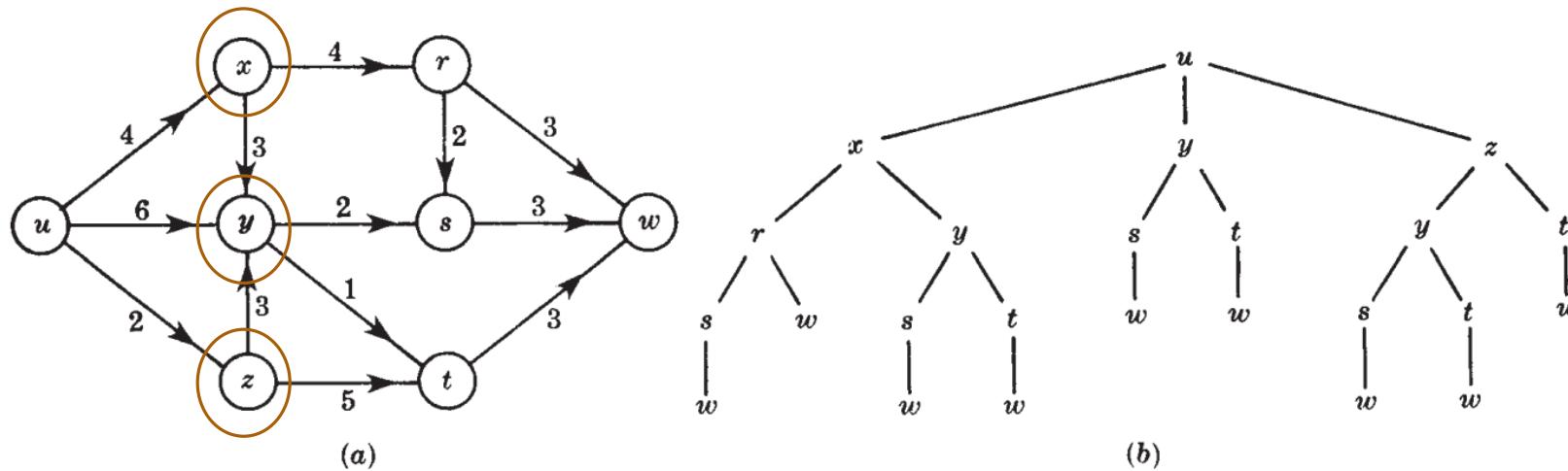
- (a) Durante a execução, cada vértice  $v'$  de  $G$  é assinalado a duas coisas:
  - (1) Um número  $\ell(v')$  denotando o atual comprimento mínimo de um caminho de  $u$  a  $v'$ .
  - (2) Um caminho  $p(v')$  de  $u$  a  $v'$  com comprimento  $\ell(v')$ .
- (b) Inicialmente, fazemos  $\ell(u) = 0$  e  $p(u) = u$ . Todos os outros vértices  $v$  são inicialmente assinalados com  $\ell(v) = \infty$  e  $p(v) = \emptyset$ .
- (c) Cada passo do algoritmo examina uma aresta  $e = (v', v)$  de  $v'$  a  $v$  com, digamos, comprimento  $k$ . Calculamos  $\ell(v') + k$ .
  - (1) Suponha que  $\ell(v') + k < \ell(v)$ . Então encontramos um caminho mais curto de  $u$  a  $v$ . Assim, atualizamos:
$$\ell(v) = \ell(v') + k \text{ e } p(v) = p(v')v$$

(Isso é sempre verdade quando  $\ell(v) = \infty$ , ou seja, quando entramos primeiro com o vértice  $v$ .)

  - (2) Caso contrário, não alteramos  $\ell(v)$  e  $p(v)$ .
- (d) O algoritmo termina quando  $p(w)$  é determinado.

Se nenhuma outra aresta não examinada entra com  $v$ , dizemos que  $p(v)$  foi determinada.

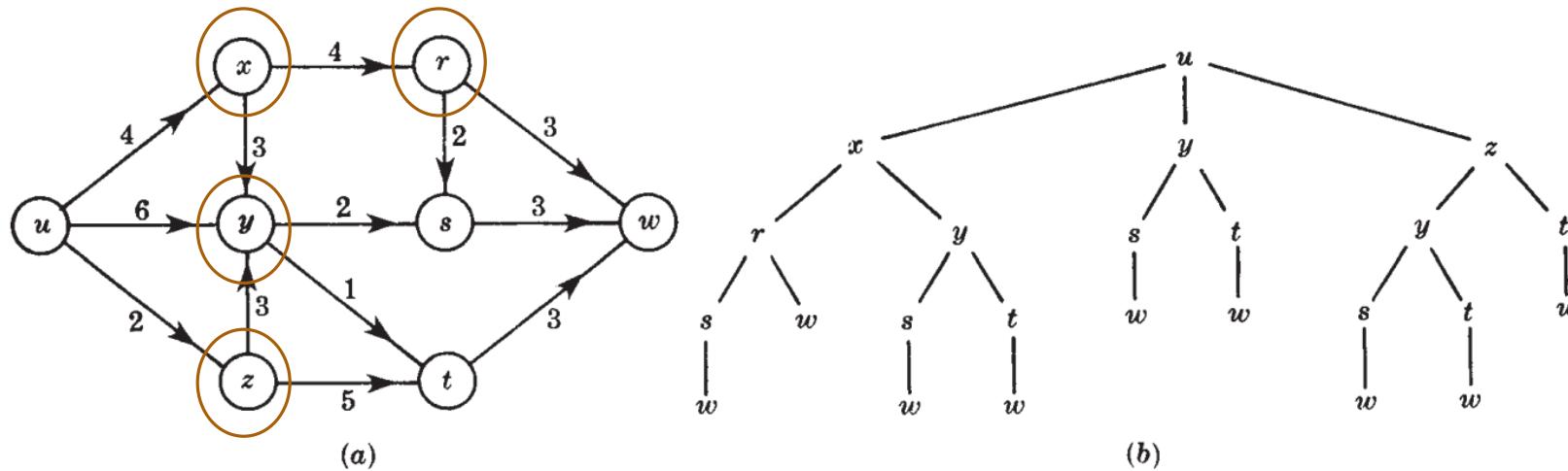
# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO



A partir de  $u$ : Os vértices sucessivos são  $x$ ,  $y$  e  $z$ , os quais são todos inseridos pela primeira vez.  
Logo:

- (1) faça  $l(x) = 4$ ,  $p(x) = ux$ .
- (2) faça  $l(y) = 6$ ,  $p(y) = uy$ . Note que  $p(x)$  e  $p(z)$  foram determinados.
- (3) faça  $l(z) = 2$ ,  $p(z) = uz$ .

# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO



A partir de x: Os vértices sucessivos são r, inserido pela primeira vez, e y. Logo:

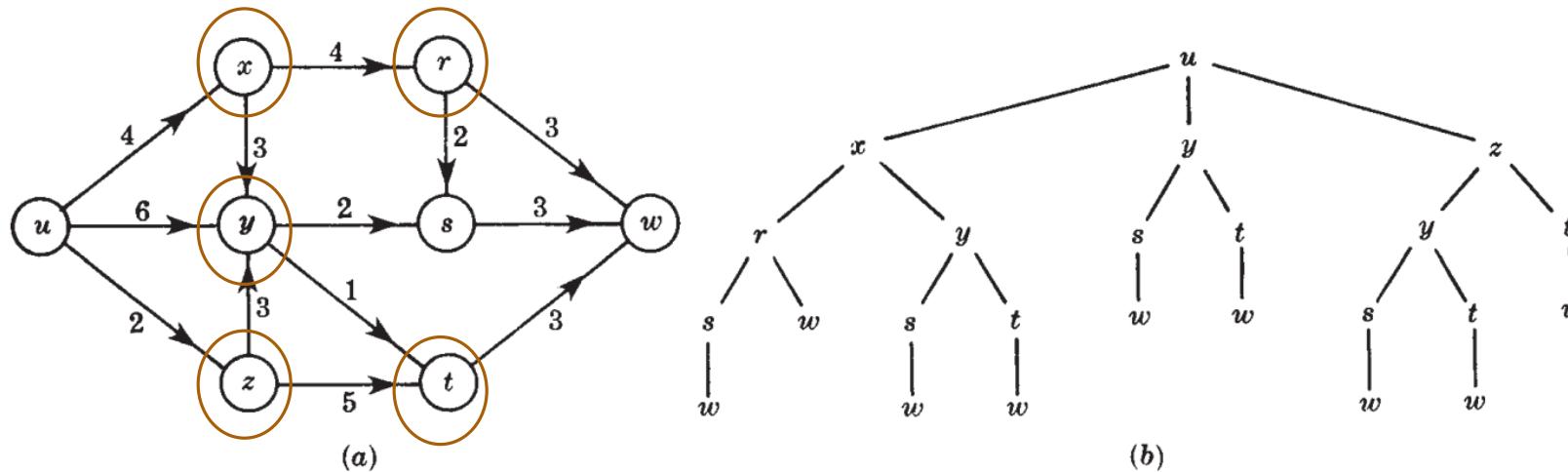
(1) faça  $l(r) = 4 + 4 = 8$  e  $p(r) = p(x)r = uxr$

(2) Calculamos:  $l(x) + k = 4 + 3 = 7$  que não é menor do que  $l(y) = 6$ .

Assim, deixamos  $l(y)$  e  $p(y)$  como estão.

Note que  $p(r)$  foi determinado.

# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO



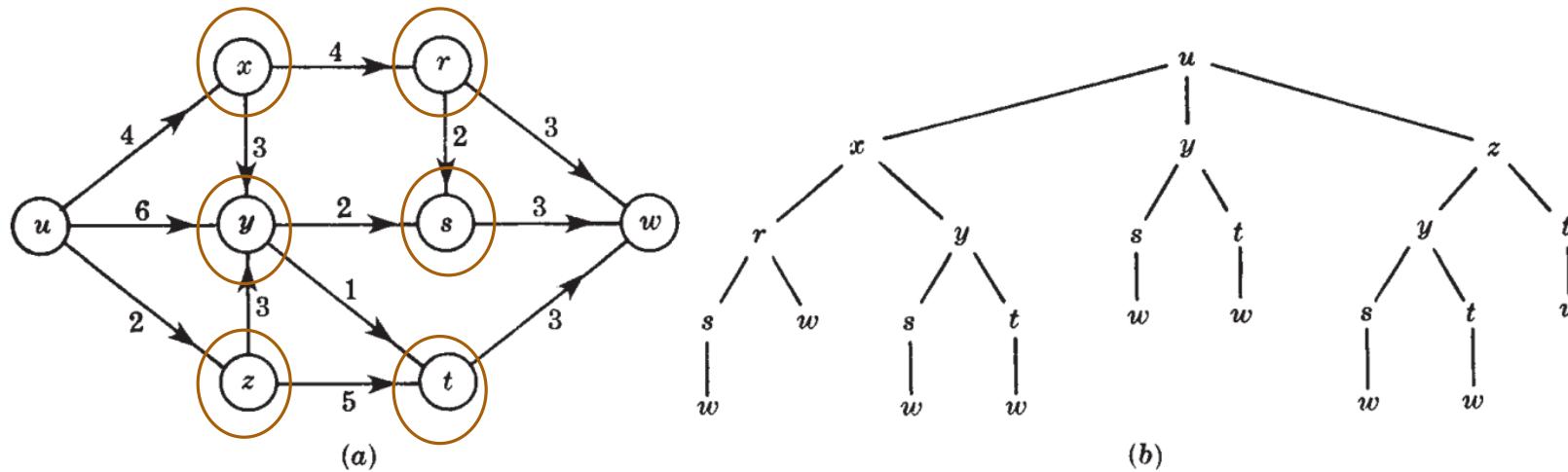
A partir de  $z$ : Os vértice sucessivos são  $t$ , inserido pela primeira vez, e  $y$ . Logo:

(1) Faça  $l(t) = l(z) + k = 2 + 5 = 7$  e  $p(t) = p(z)t = urt$

(2) Calculamos:  $l(z) + k = 2 + 3 = 5$  que é menor do que  $l(y) = 6$ . Encontramos um caminho mais curto para  $y$  e, assim, atualizamos  $l(y)$  e  $p(y)$ ; faça:  $l(y) = p(z) + k = 5$  e  $p(y) = p(z)y = uzy$ .

Agora  $p(y)$  foi determinado.

# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO

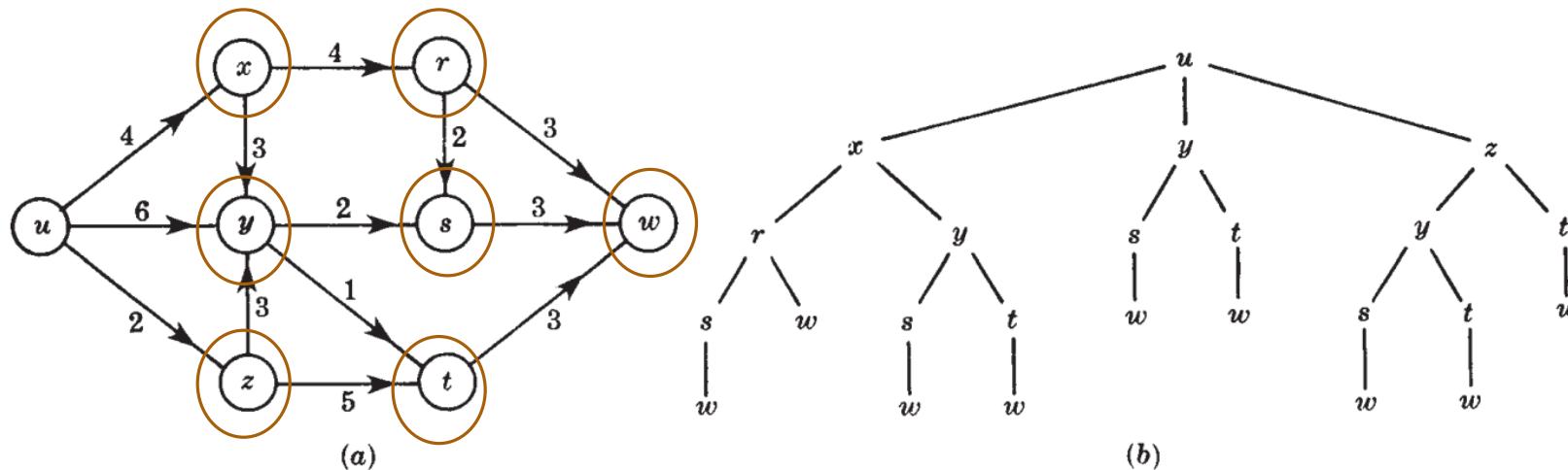


A partir de  $y$ : Os vértices sucessivos são  $s$ , inserido pela primeira vez, e  $t$ . Logo:

- (1) Fazemos  $l(s) = l(y) + k = 5 + 2 = 7$  e  $p(s) = p(y)s = uzys$ .
- (2) Calculamos:  $l(y) + k = 5 + 1 = 6$ , que é menor do que  $l(t) = 7$ . Logo, mudamos  $l(t)$  e  $p(t)$  para:  $l(t) = l(y) + 1 = 6$  e  $p(t) = p(y)t = uzyt$ .

Agora  $p(t)$  for determinado.

# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO



A partir de  $r$ : Os vértices sucessivos são  $w$ , inserido pela primeira vez, e  $s$ . Assim:

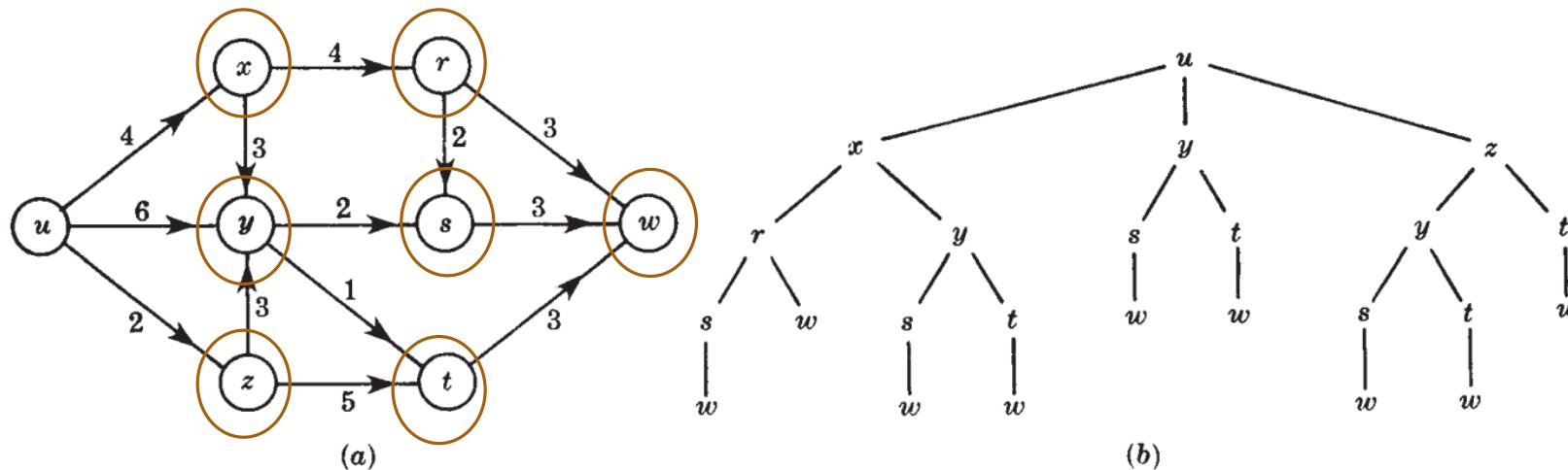
(1) Fazemos  $l(w) = l(r) + 3 = 11$  e  $p(w) = p(r)w = uxrw$ .

(2) Calculamos:  $l(r) + k = 8 + 2 = 10$  que não é menor do que  $l(s) = 7$ . Logo, deixamos  $l(s)$  e  $p(s)$  como estão.

Note que  $p(s)$  foi determinado.

# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO

---

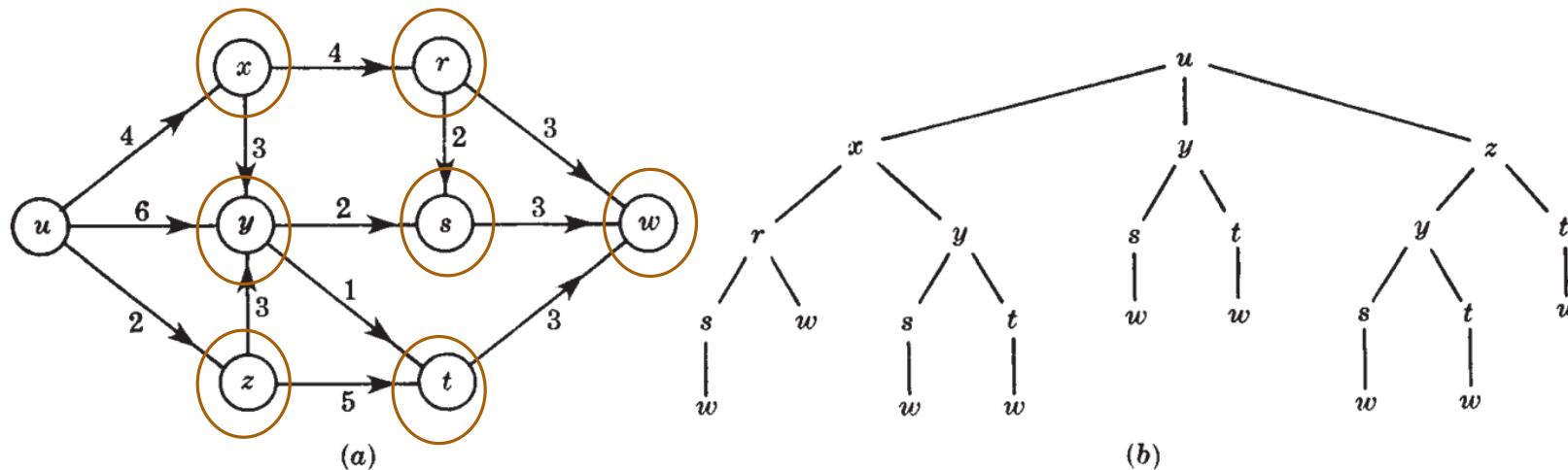


A partir de  $s$ : O vértice sucessivo é  $w$ . Calculamos:

$$l(s) + k = 7 + 3 = 10 \text{ que é menor do que } l(w) = 11.$$

Logo, mudamos  $l(w)$  e  $p(w)$  para:  $l(w) = l(s) + 3 = 10$  e  $p(w) = p(s)w = uzysw$ .

# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO



A partir de  $t$ : O vértice sucessivo é  $w$ . Calculamos:

$$l(t) + k = 6 + 3 = 9 \text{ que é menor do que } l(w) = 10$$

Logo, atualizamos  $l(w)$  e  $p(w)$  como se segue:  $l(w) = l(t) + 3 = 9$  e  $p(w) = p(t) = uzytw$

Agora  $p(w)$  foi determinado.

# ALGORITMOS DE PODA PARA CAMINHO MAIS CURTO

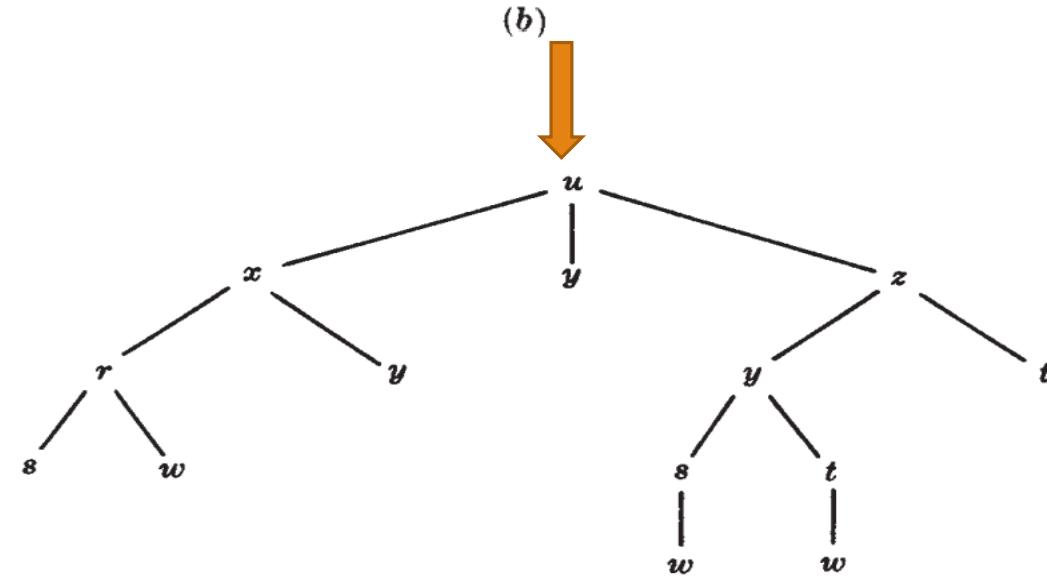
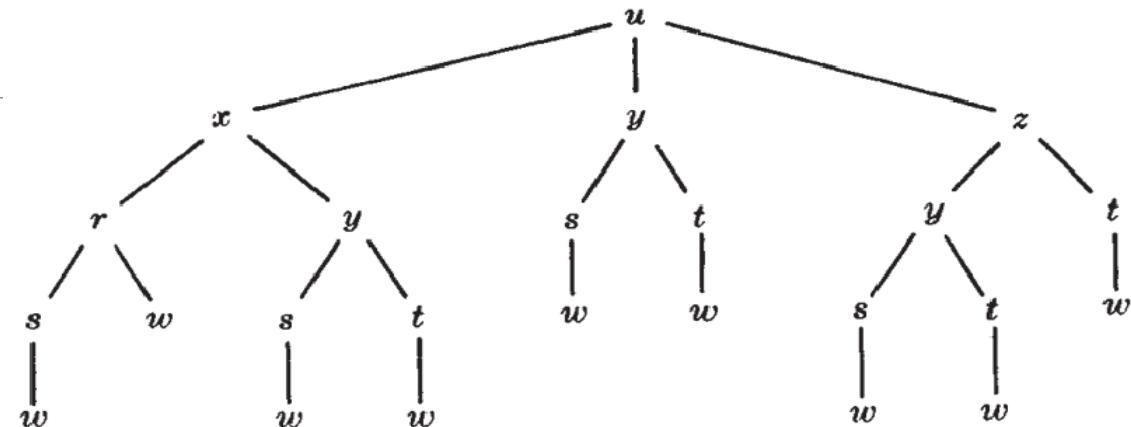
O algoritmo é encerrado, pois  $p(w)$  foi determinado.

Portanto,  $p(w) = uzytw$  é o caminho mais curto de  $u$  a  $w$  e  $l(w) = 9$ .

As arestas que foram examinadas no exemplo formam a árvore enraizada

Essa é a árvore nos vértices pertencentes a caminhos parciais mais longos.

Observe que apenas 13 das 23 arestas originais da árvore tiveram que ser examinadas.





# Dijkstra

---

O algoritmo de Dijkstra, concebido pelo cientista da computação holandês Edsger Dijkstra em 1956 e publicado em 1959,[1][2] soluciona o problema do caminho mais curto num grafo dirigido ou não dirigido com arestas de peso não negativo.

O algoritmo considera um conjunto  $S$  de menores caminhos, iniciado com um vértice inicial  $I$ . A cada passo do algoritmo busca-se nas adjacências dos vértices pertencentes a  $S$  aquele vértice com menor distância relativa a  $I$  e adiciona-o a  $S$  e, então, repetindo os passos até que todos os vértices alcançáveis por  $I$  estejam em  $S$ . Arestras que ligam vértices já pertencentes a  $S$  são desconsideradas.

# Dijkstra

---

Seja  $G(V,A)$  um grafo orientado e  $s$  um vértice de  $G$ :

Atribua valor zero à estimativa do custo mínimo do vértice  $s$  (a raiz da busca) e infinito às demais estimativas;

Atribua um valor qualquer aos precedentes (o precedente de um vértice  $t$  é o vértice que precede  $t$  no caminho de custo mínimo de  $s$  para  $t$ );

Enquanto houver vértice aberto:

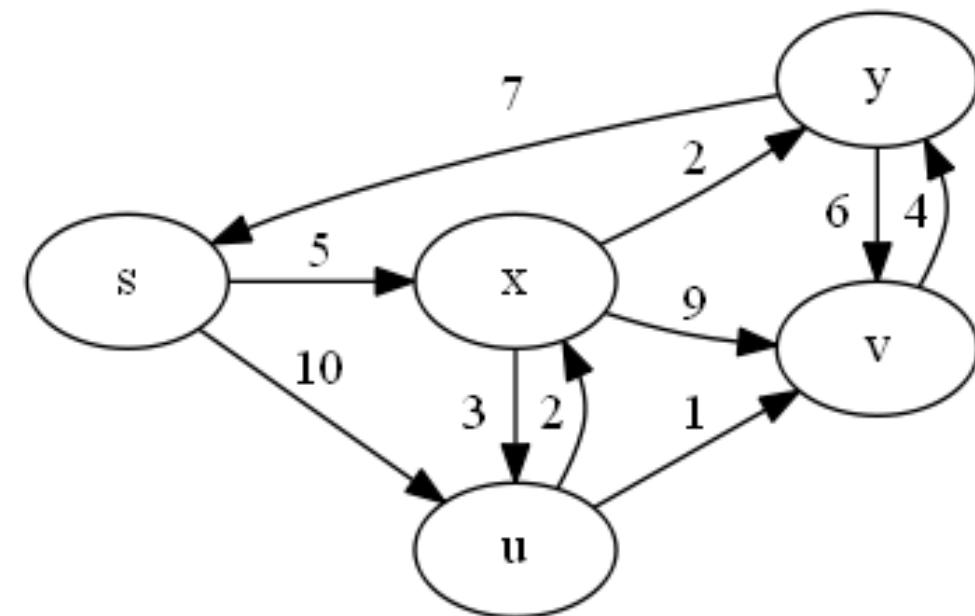
- seja  $k$  um vértice ainda aberto cuja estimativa seja a menor dentre todos os vértices abertos;
- feche o vértice  $k$
- Para todo vértice  $j$  ainda aberto que seja sucessor de  $k$  faça:
  - some a estimativa do vértice  $k$  com o custo do arco que une  $k$  a  $j$ ;
  - caso esta soma seja melhor que a estimativa anterior para o vértice  $j$ , substitua-a e anote  $k$  como precedente de  $j$ .

# Dijkstra

---

Inicialmente todos os nodos tem um custo infinito, exceto  $s$  (a raiz da busca) que tem valor 0:

vértices	$s$	$u$	$v$	$x$	$y$
estimativas	0	$\infty$	$\infty$	$\infty$	$\infty$
precedentes	-				

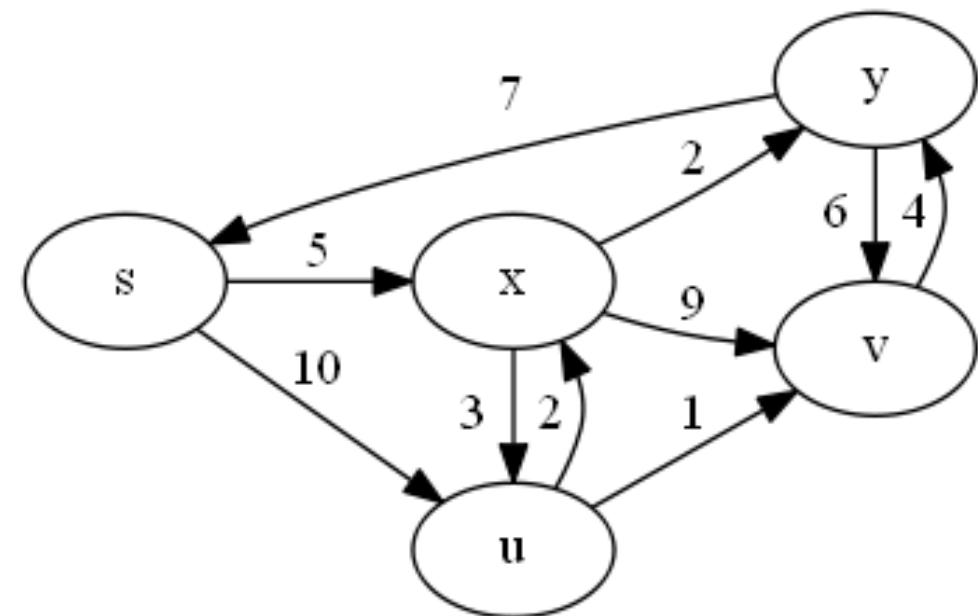


# Dijkstra

---

- selecione **s** (vértice aberto de estimativa mínima)
- feche **s**
- recalcule as estimativas de u e x

vértices	s	u	v	x	y
estimativas	0	10	$\infty$	5	$\infty$
precedentes	s	s	-	s	-

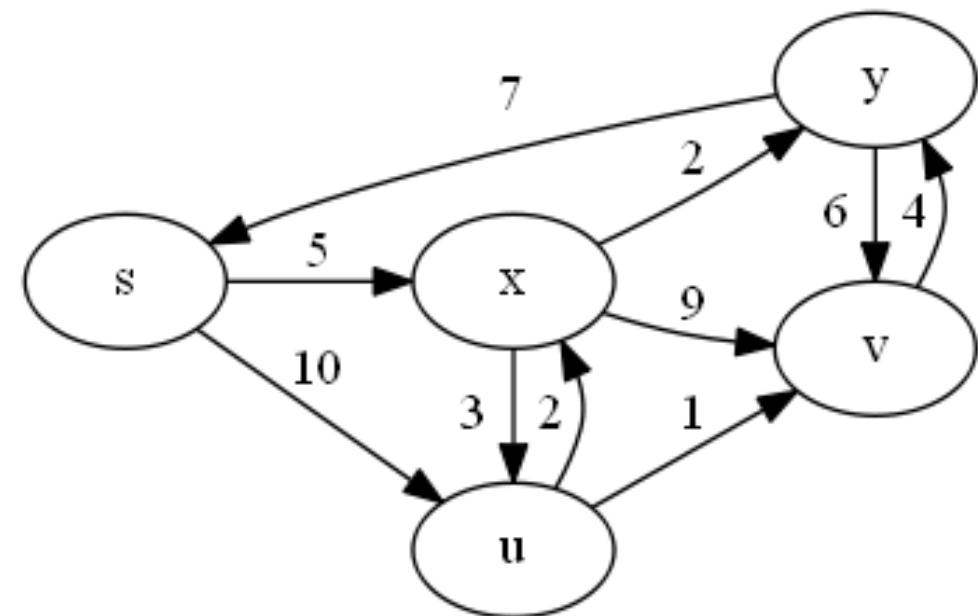


# Dijkstra

---

- selecione x (vértice aberto de estimativa mínima)
- feche x
- recalcule as estimativas de u,v e y

vértices	s	u	v	x	y
estimativas	0	8	14	5	7
precedentes	s	x	x	s	x

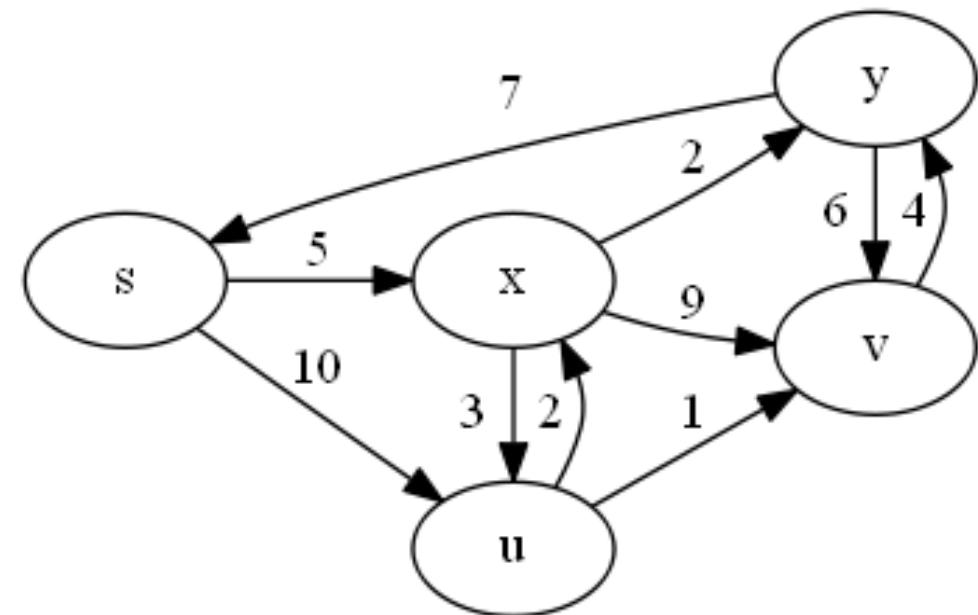


# Dijkstra

---

- selecione **y** (vértice aberto de estimativa mínima)
- feche **y**
- recalcule a estimativa de v

vértices	s	u	v	x	y
estimativas	0	8	13	5	7
precedentes	s	x	y	s	x

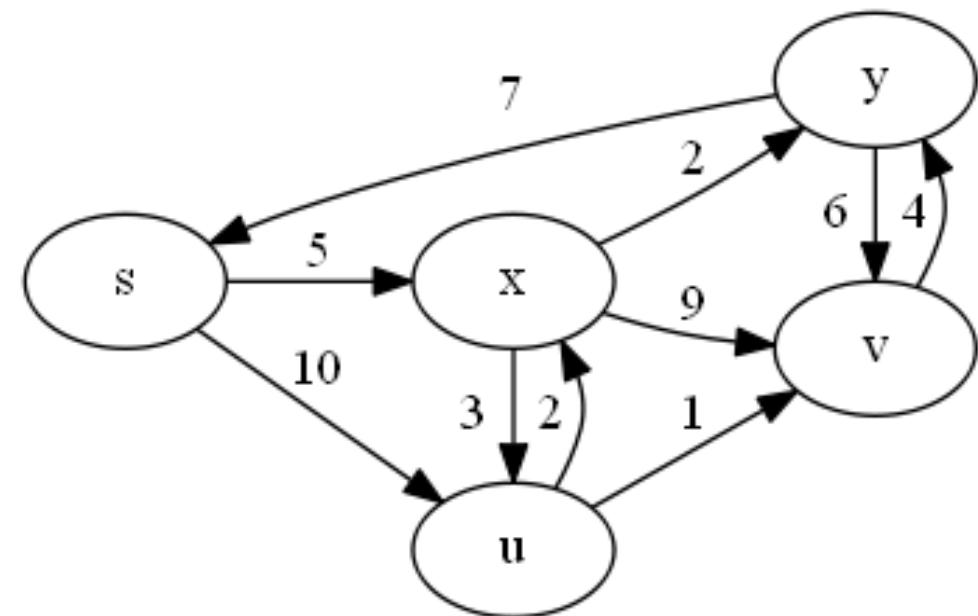


# Dijkstra

---

- selecione **u** (vértice aberto de estimativa mínima)
- feche **u**
- recalcule a estimativa de v

vértices	s	u	v	x	y
estimativas	0	8	9	5	7
precedentes	s	x	u	s	x



# Dijkstra

---

- selecione **v** (vértice aberto de estimativa mínima)
- feche **v**

vértices	s	u	v	x	y
estimativas	0	8	9	5	7
precedentes	s	x	u	s	x

Quando todos os vértices tiverem sido fechados, os valores obtidos serão os custos mínimos dos caminhos que partem do vértice tomado como raiz da busca até os demais vértices do grafo. O caminho propriamente dito é obtido a partir dos vértices chamados acima de precedentes.

