

# Algoritmos e Estruturas de Dados I

## Aula 08: Grafos

Prof. Márcio Porto Basgalupp

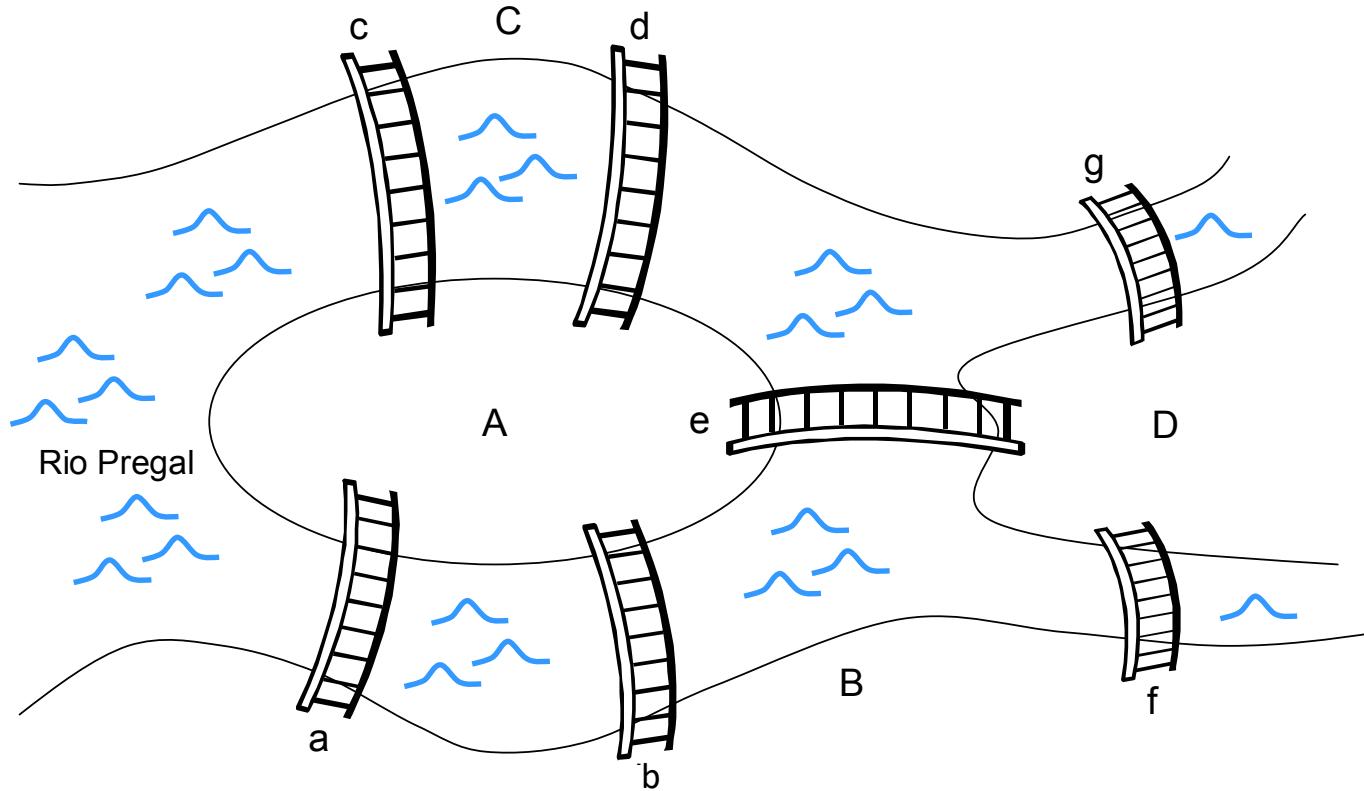
*créditos: Prof. Jurandy G. Almeida Jr.*

Universidade Federal de São Paulo  
Departamento de Ciência e Tecnologia

- A primeira evidência sobre **grafos** remonta a 1736, quando Euler fez uso deles para solucionar o problema clássico das pontes de Koenigsberg
- Na cidade de Koenigsberg (na Prússia Oriental), o rio Pregal flui em torno da ilha de Kneiphof, dividindo-se em seguida em duas partes
- O problema das pontes de Koenigsberg consiste em determinar se, ao partir de alguma área de terra, é possível atravessar todos os pontos exatamente uma vez, para, em seguida, retornar à área de terra inicial

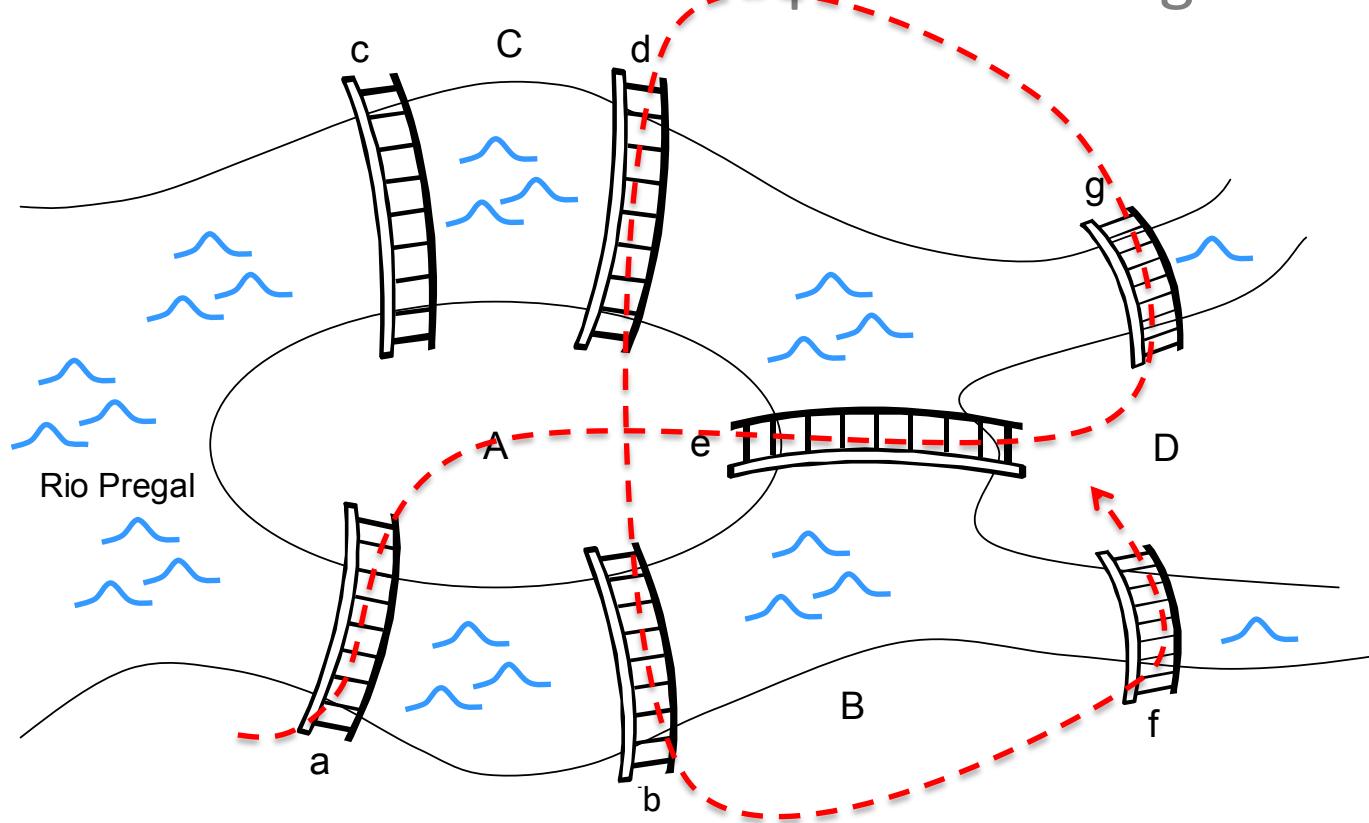
# Introdução

- É possível caminhar sobre cada ponte exatamente uma única vez e retornar ao ponto de origem?



# Introdução

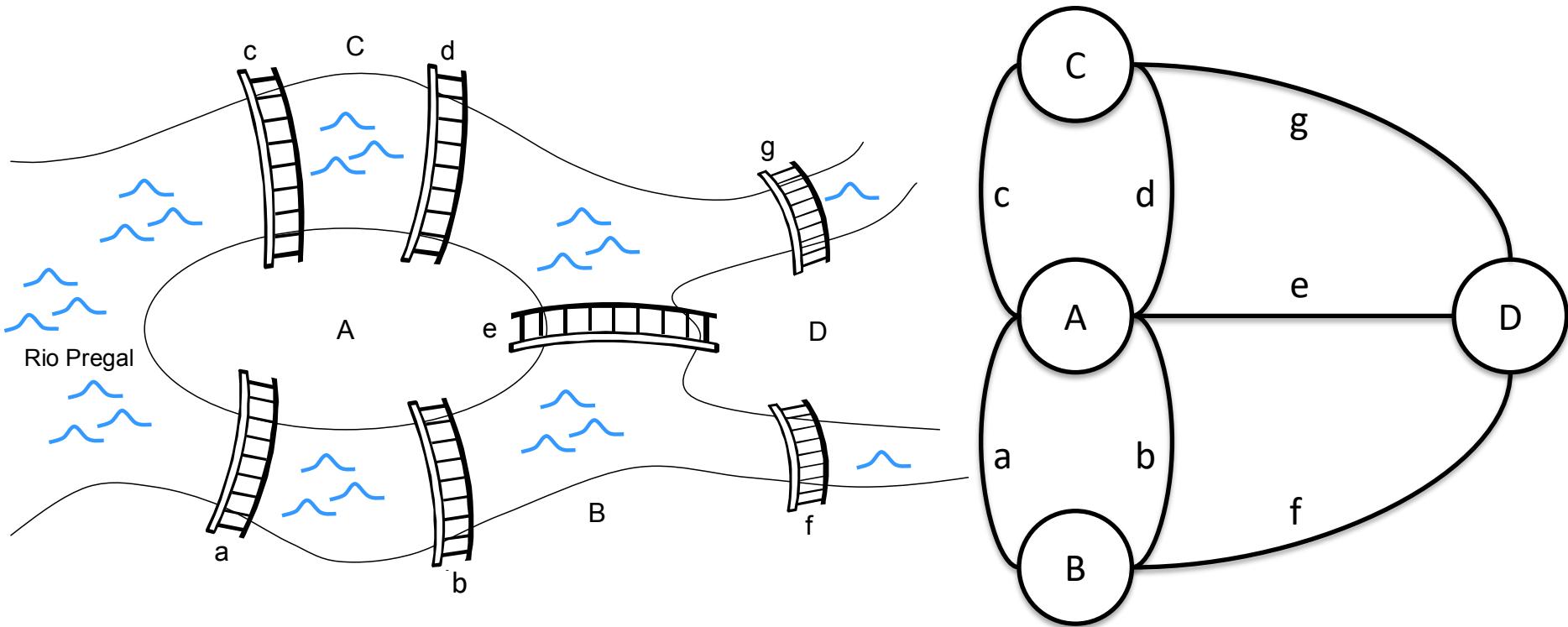
- É possível caminhar sobre cada ponte exatamente uma única vez e retornar ao ponto de origem?



- Um caminho possível consistiria em iniciar na área de terra **B**, atravessar a ponte **a** para a ilha **A**; pegar a ponte **e** para chegar à área **D**, atravessar a ponte **g**, chegando a **C**; cruzar a ponte **d** até **A**; cruzar a ponte **b** até **B** e a ponte **f**, chegando a **D**
  - Esse caminho não atravessa todas as pontes uma vez, tampouco retorna à área inicial de terra **B**

- Euler provou que não é possível o povo de Koenigsberg atravessar cada ponte exatamente uma vez, retornando ao ponto inicial
  - Ele resolveu o problema, representando áreas de terra como vértices e as pontes como arestas de um grafo

- Problema das Pontes de Koenigsberg como um Grafo



# Introdução

- Aplicações envolvendo algoritmos em grafos:
  - Análise de circuitos elétricos
  - Verificação de caminhos mais curtos
  - Análise de planejamento de projetos
  - Identificação de compostos químicos
  - Genética
  - Linguística
  - Ciências sociais
  - ...
- Pode-se afirmar que, de todas as estruturas matemáticas, grafos são as que se encontram em uso mais amplo

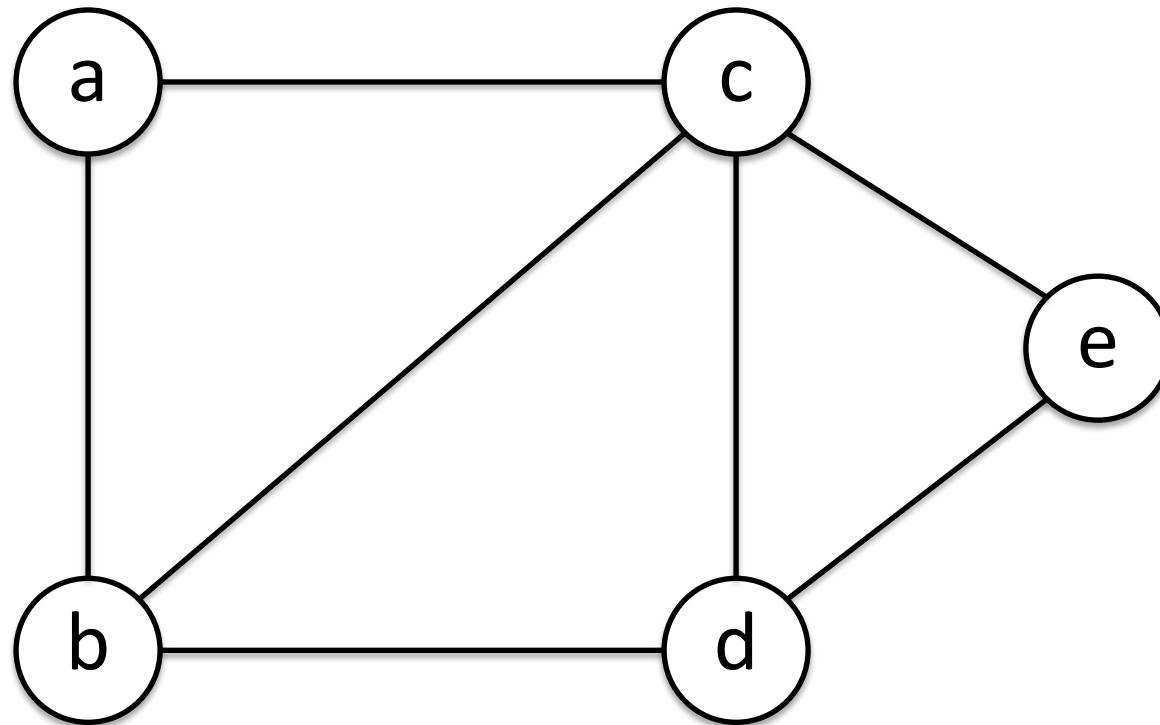
# CONCEITOS BÁSICOS

- Um grafo é um par  $\mathbf{G}=(\mathbf{V}, \mathbf{E})$  em que:
  - $\mathbf{V}$  é um conjunto finito de elementos chamados **vértices**
  - $\mathbf{E}$  é um conjunto finito de pares de vértices chamados **arestas**
- A seguinte notação será adotada:
  - $\mathbf{V}$  ou  $\mathbf{V}(\mathbf{G})$  para representar o conjunto de vértices de  $\mathbf{G}$
  - $\mathbf{E}$  ou  $\mathbf{E}(\mathbf{G})$  para representar o conjunto de arestas de  $\mathbf{G}$
  - $\mathbf{G}$  ou  $\mathbf{G}=(\mathbf{V}, \mathbf{E})$  ou  $\mathbf{G}(\mathbf{V}, \mathbf{E})$  para representar um grafo

# Conceitos Básicos

## Definição de Grafo

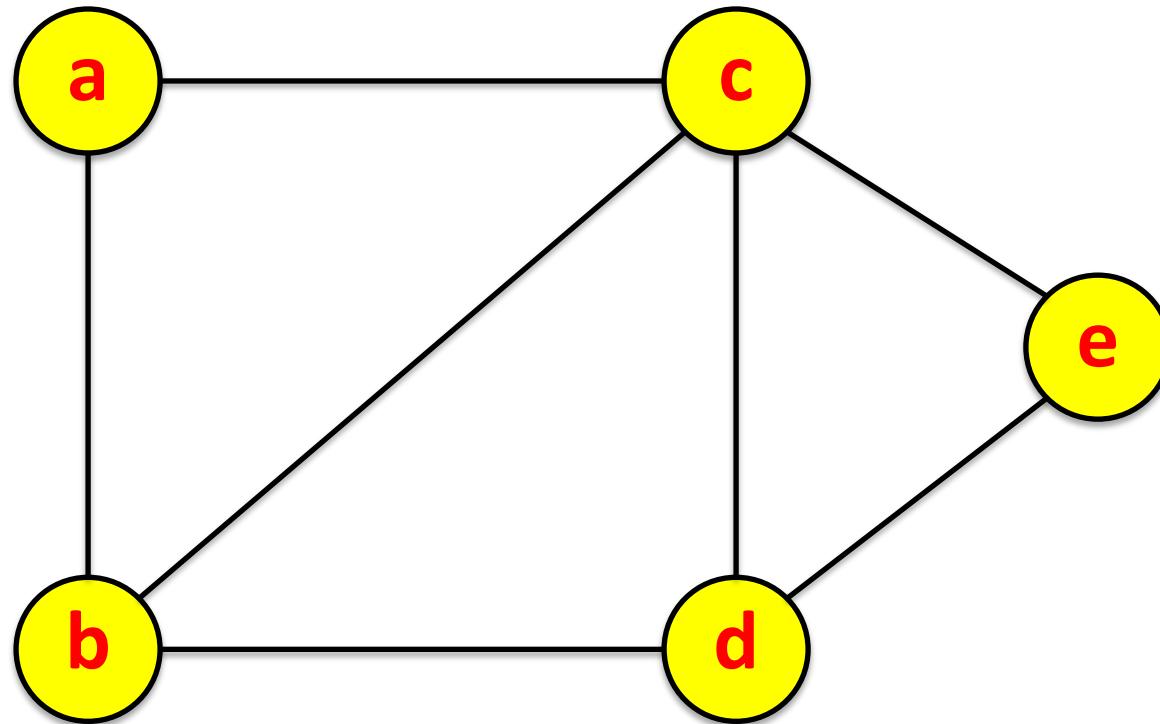
- $V = \{a, b, c, d, e\}$
- $E = \{(a, b); (a, c); (b, c); (b, d); (c, d); (c, e); (d, e)\}$



# Conceitos Básicos

## Definição de Grafo

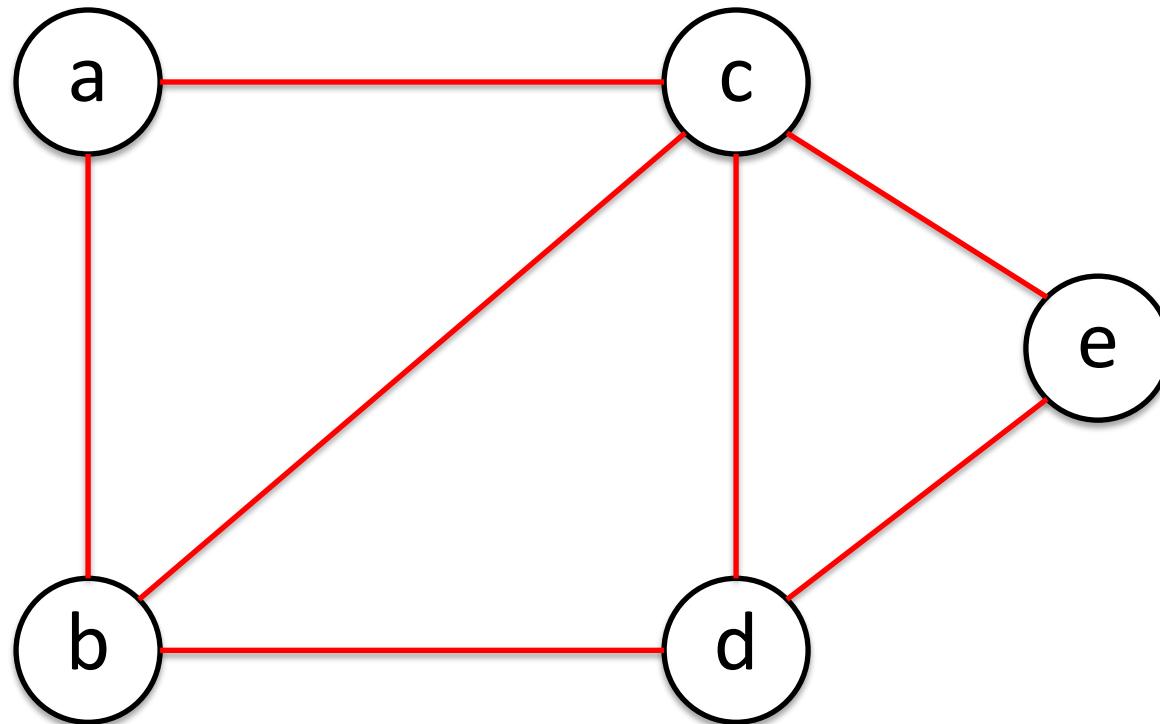
- $V = \{a, b, c, d, e\}$
- $E = \{(a, b); (a, c); (b, c); (b, d); (c, d); (c, e); (d, e)\}$



# Conceitos Básicos

## Definição de Grafo

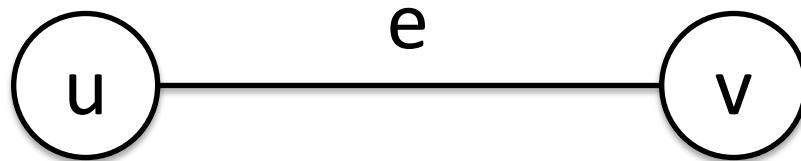
- $V = \{a, b, c, d, e\}$
- $E = \{(a, b); (a, c); (b, c); (b, d); (c, d); (c, e); (d, e)\}$



# Conceitos Básicos

## Definição de Grafo

- Sendo  $e=(u, v)$  uma aresta em  $E$ , dizemos que:
  - os vértices  $u$  e  $v$  são os **extremos** da aresta  $e$
  - os vértices  $u$  e  $v$  são vértices **adjacentes**
  - a aresta  $e$  é **incidente** aos vértices  $u$  e  $v$

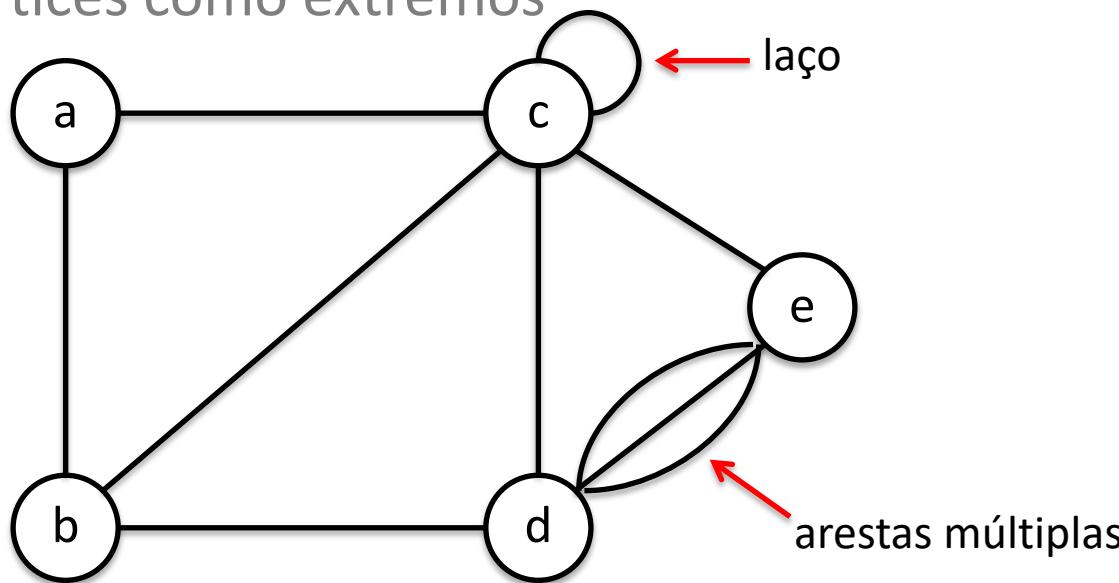


# Conceitos Básicos

## Grafo Simples

- Um grafo é **simples** quando não possui laços ou arestas múltiplas

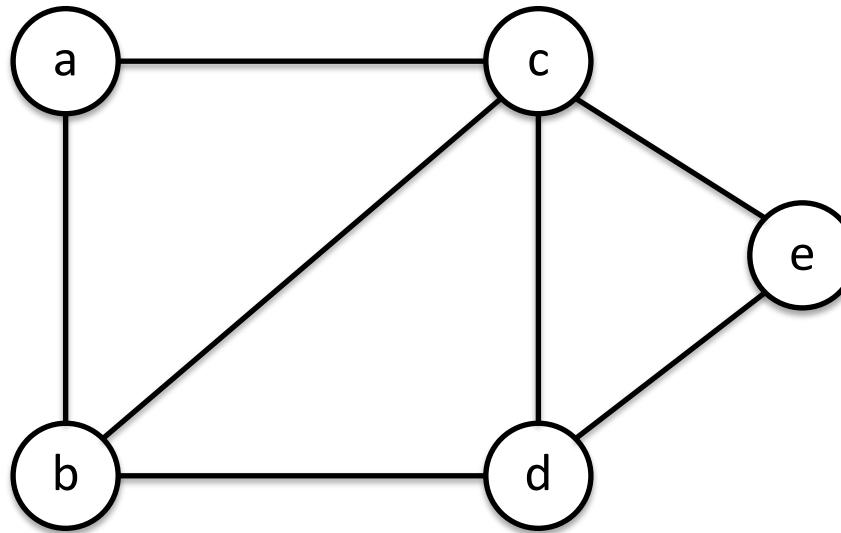
- Um **laço** é uma aresta com extremos idênticos
- **Arestas múltiplas** são duas ou mais arestas com o mesmo par de vértices como extremos



# Conceitos Básicos

## Cardinalidade de Vértices e Arestas

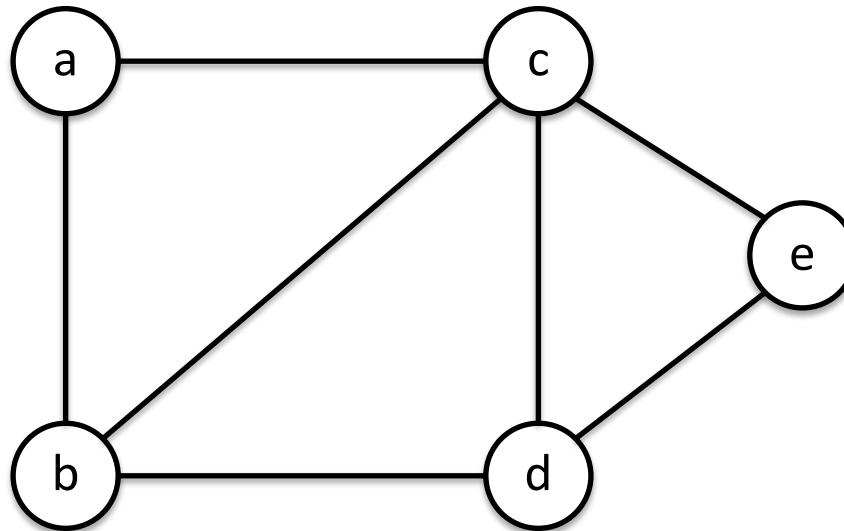
- Denota-se por  $|V|$  e  $|E|$  a cardinalidade dos conjuntos de vértices e arestas de um grafo  $G=(V, E)$ 
  - No exemplo abaixo, tem-se  $|V| = 5$  e  $|E| = 7$



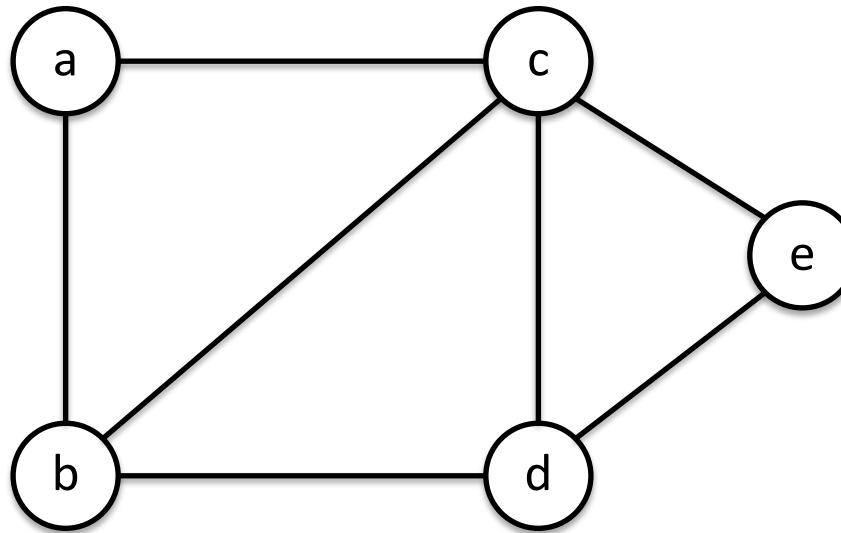
# Conceitos Básicos

## Ordem do Grafo

- A **ordem** do grafo **G** é dada por  $|V|$ 
  - No exemplo abaixo, a ordem do grafo é 5



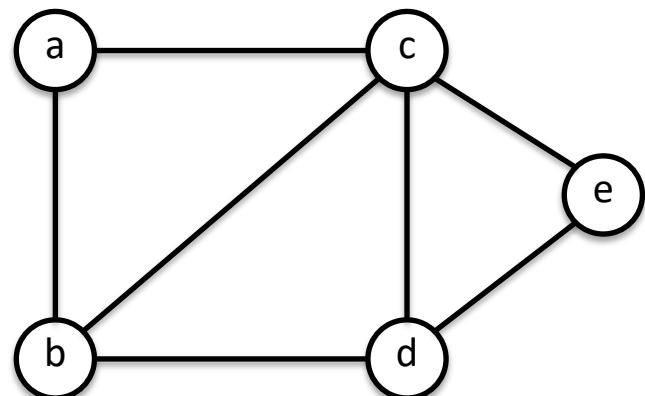
- O **tamanho** do grafo **G** é dado por  $|V| + |E|$ 
  - No exemplo abaixo, o tamanho do grafo é 12



# Conceitos Básicos

## Subgrafo

- Um **subgrafo**  $G'=(V', E')$  de um grafo  $G=(V, E)$  é um grafo tal que  $V' \subseteq V$  e  $E' \subseteq E$

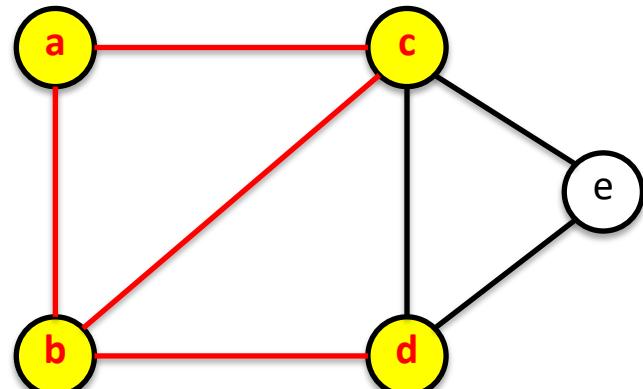


Grafo G

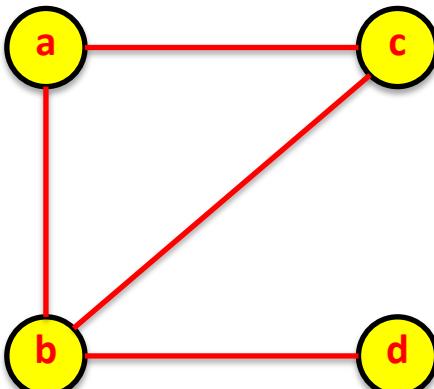
# Conceitos Básicos

## Subgrafo

- Um **subgrafo**  $G'=(V', E')$  de um grafo  $G=(V, E)$  é um grafo tal que  $V' \subseteq V$  e  $E' \subseteq E$



Grafo G

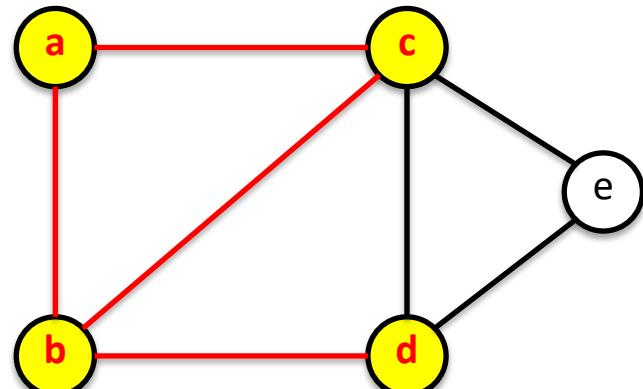


Subgrafo

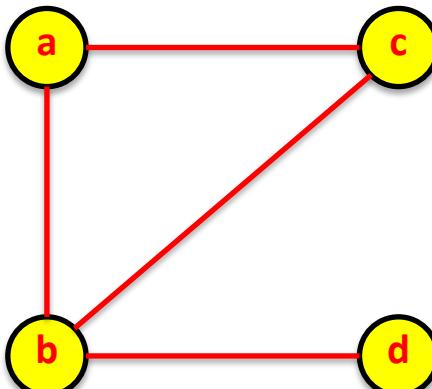
# Conceitos Básicos

## Subgrafo Gerador

- Um **subgrafo  $G'=(V', E')$**  de um grafo  **$G=(V, E)$**  é um grafo tal que  $V' \subseteq V$  e  $E' \subseteq E$
- Um **subgrafo gerador** de  **$G$**  é um subgrafo  **$G'$**  com  $V'=V$



Grafo G

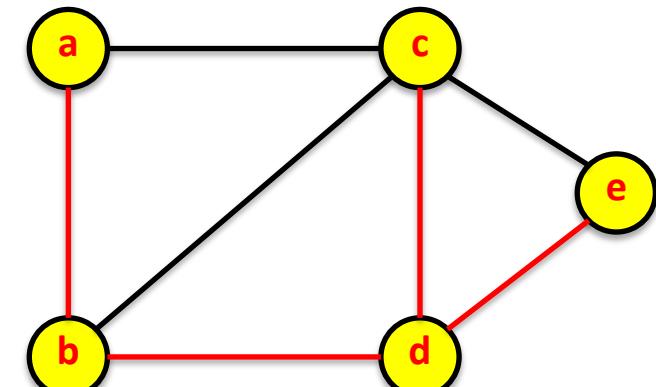


Subgrafo não gerador

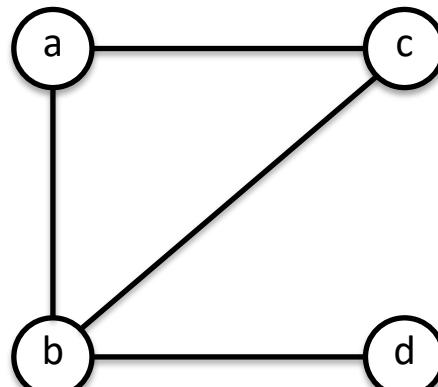
# Conceitos Básicos

## Subgrafo Gerador

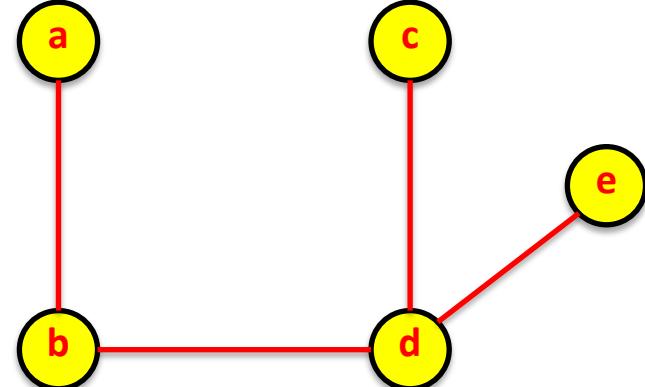
- Um **subgrafo  $G'=(V', E')$**  de um grafo  **$G=(V, E)$**  é um grafo tal que  $V' \subseteq V$  e  $E' \subseteq E$
- Um **subgrafo gerador** de  **$G$**  é um subgrafo  **$G'$**  com  $V'=V$



Grafo G



Subgrafo não gerador

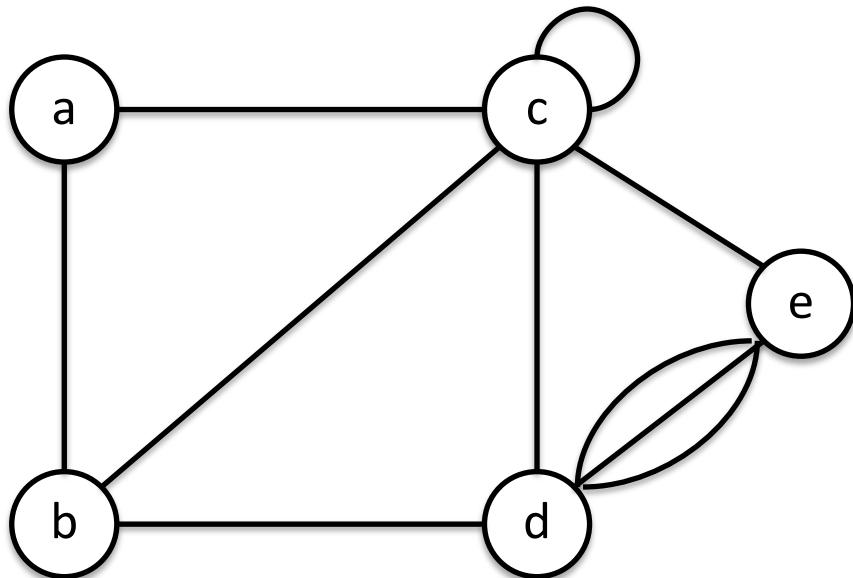


Subgrafo gerador

# Conceitos Básicos

## Grau de um Vértice

- O **grau** de um vértice  $v$ , denotado por  $d(v)$ , é o número de arestas incidentes a  $v$ 
  - Os laços são contados duas vezes

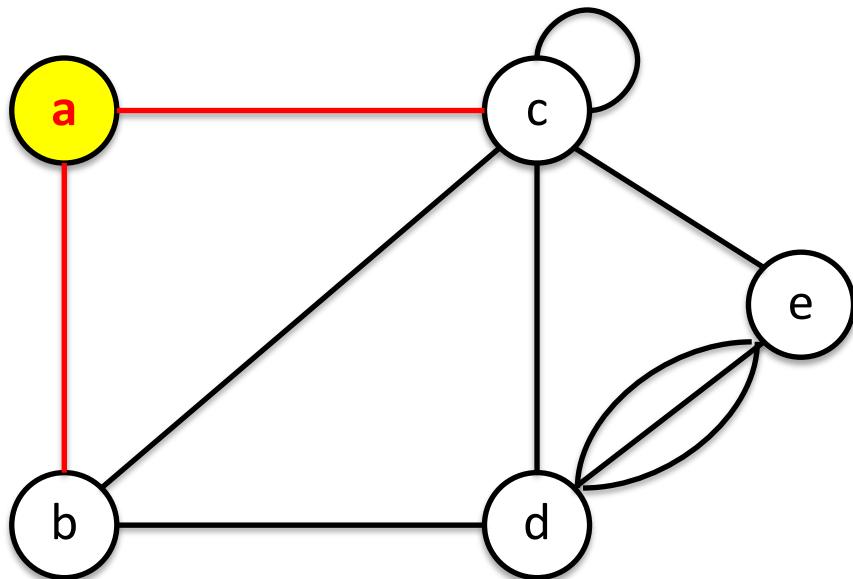


Vértice	Grau
a	
b	
c	
d	
e	

# Conceitos Básicos

## Grau de um Vértice

- O **grau** de um vértice  $v$ , denotado por  $d(v)$ , é o número de arestas incidentes a  $v$ 
  - Os laços são contados duas vezes

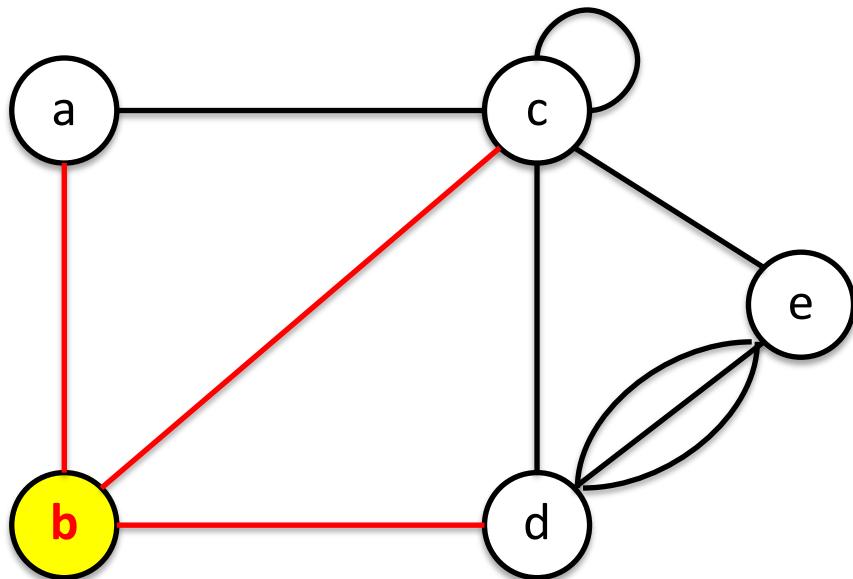


Vértice	Grau
a	2
b	
c	
d	
e	

# Conceitos Básicos

## Grau de um Vértice

- O **grau** de um vértice  $v$ , denotado por  $d(v)$ , é o número de arestas incidentes a  $v$ 
  - Os laços são contados duas vezes

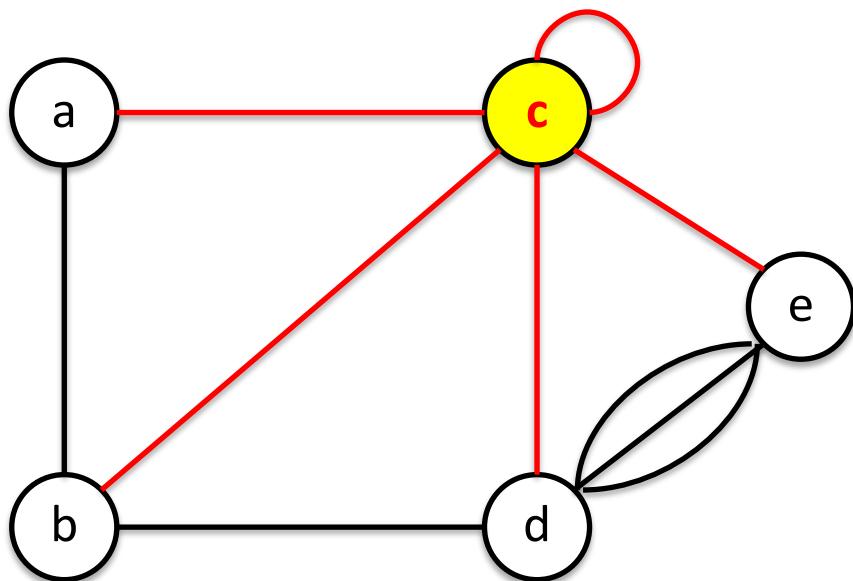


Vértice	Grau
a	2
b	3
c	
d	
e	

# Conceitos Básicos

## Grau de um Vértice

- O **grau** de um vértice  $v$ , denotado por  $d(v)$ , é o número de arestas incidentes a  $v$ 
  - Os laços são contados duas vezes

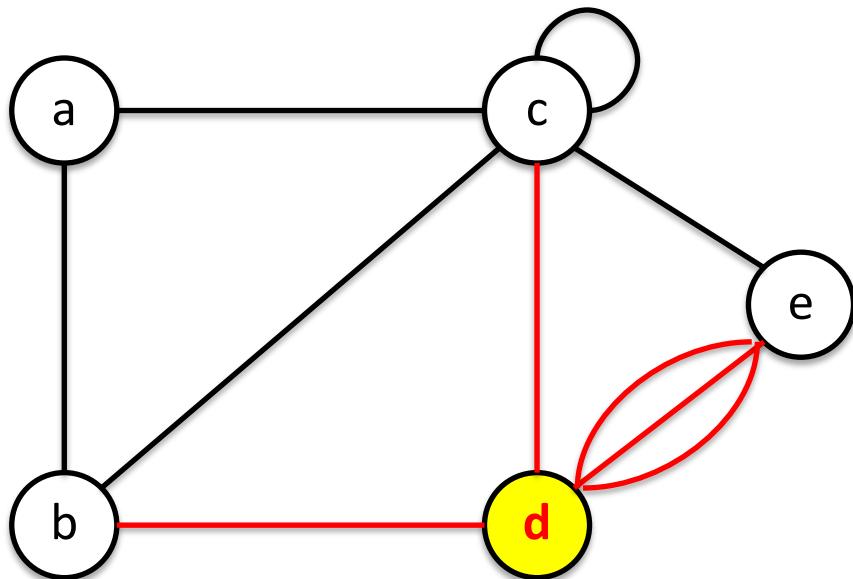


Vértice	Grau
a	2
b	3
c	6
d	
e	

# Conceitos Básicos

## Grau de um Vértice

- O **grau** de um vértice  $v$ , denotado por  $d(v)$ , é o número de arestas incidentes a  $v$ 
  - Os laços são contados duas vezes

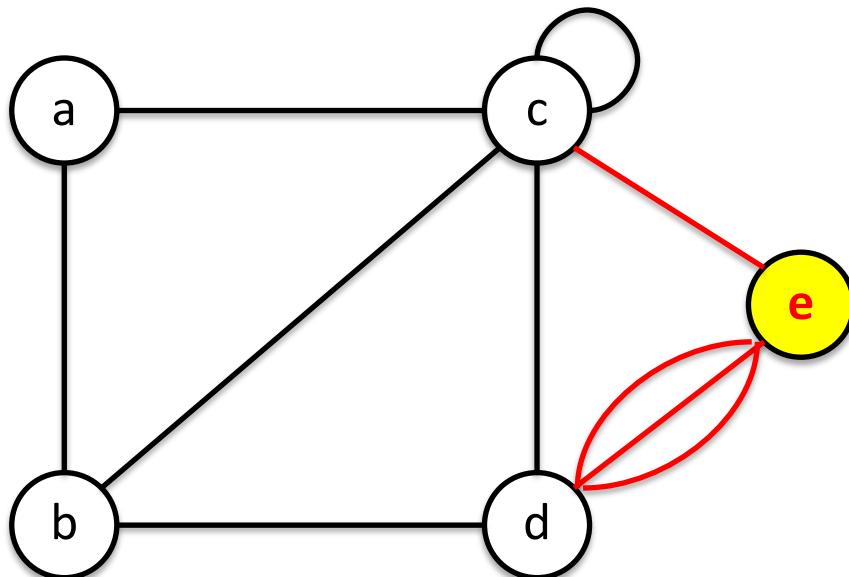


Vértice	Grau
a	2
b	3
c	6
<b>d</b>	<b>5</b>
e	

# Conceitos Básicos

## Grau de um Vértice

- O **grau** de um vértice  $v$ , denotado por  $d(v)$ , é o número de arestas incidentes a  $v$ 
  - Os laços são contados duas vezes

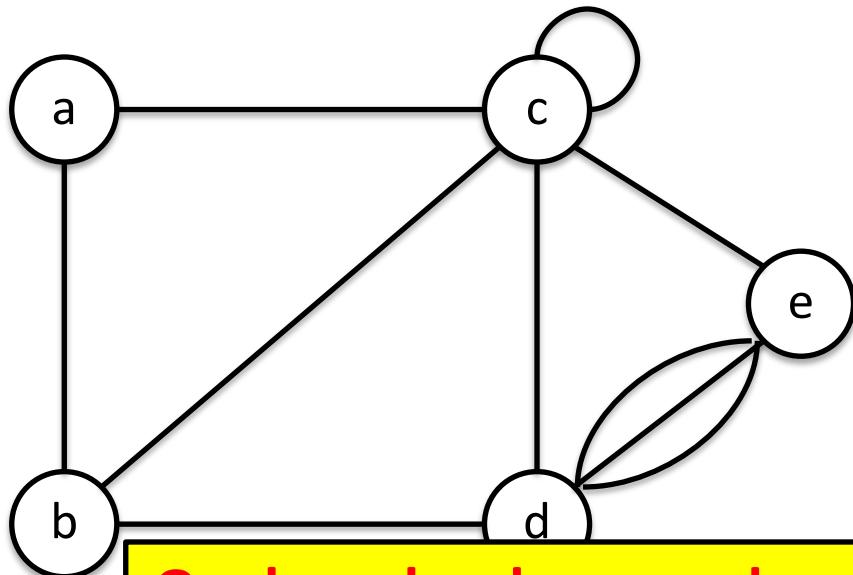


Vértice	Grau
a	2
b	3
c	6
d	5
e	4

# Conceitos Básicos

## Grau de um Vértice

- O **grau** de um vértice  $v$ , denotado por  $d(v)$ , é o número de arestas incidentes a  $v$ 
  - Os laços são contados duas vezes



Vértice	Grau
a	2
b	3
c	6
d	5
e	4

Qual o valor da soma dos graus de todos os vértices?

- O **grau** de um vértice  $v$ , denotado por  $d(v)$ , é o número de arestas incidentes a  $v$ 
  - Os laços são contados duas vezes

### Teorema (*Handshaking lemma*)

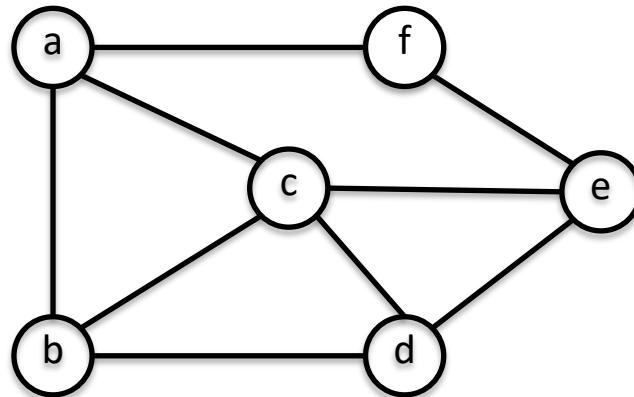
Para todo grafo  $\mathbf{G}=(V, E)$ , tem-se que:

$$\sum_{v \in V} d(v) = 2|E|$$

# Conceitos Básicos

## Caminhos em Grafos

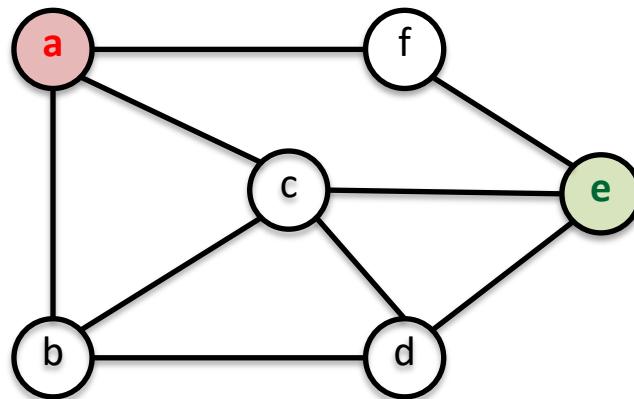
- Um **caminho** de  $v_0$  a  $v_n$  no grafo  $\mathbf{G}$  é uma sequência finita e não vazia  $(v_0, e_1, v_1, \dots, e_n, v_n)$ , cujos elementos são alternadamente vértices e arestas e tal que, para todo  $1 \leq i \leq n$ ,  $v_{i-1}$  e  $v_i$  são extremos de  $e_i$ .



# Conceitos Básicos

## Caminhos em Grafos

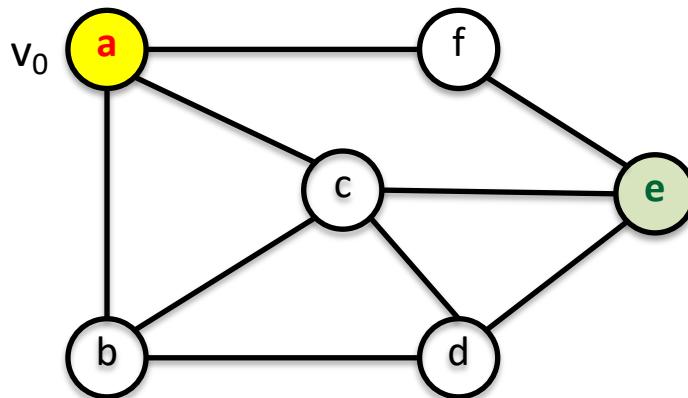
- Um **caminho** de  $v_0$  a  $v_n$  no grafo  $\mathbf{G}$  é uma sequência finita e não vazia  $(v_0, e_1, v_1, \dots, e_n, v_n)$ , cujos elementos são alternadamente vértices e arestas e tal que, para todo  $1 \leq i \leq n$ ,  $v_{i-1}$  e  $v_i$  são extremos de  $e_i$ .



# Conceitos Básicos

## Caminhos em Grafos

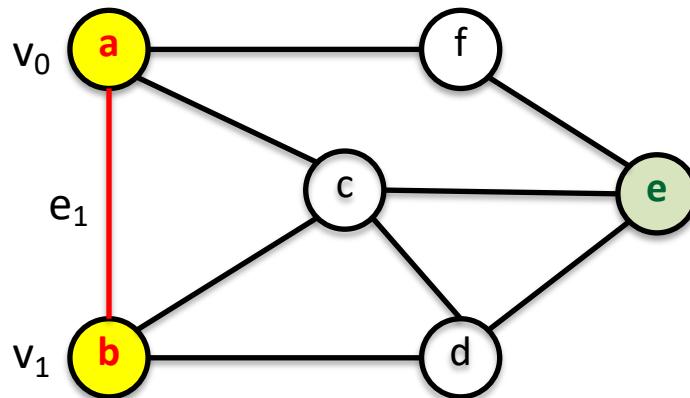
- Um **caminho** de  $v_0$  a  $v_n$  no grafo  $\mathbf{G}$  é uma sequência finita e não vazia  $(v_0, e_1, v_1, \dots, e_n, v_n)$ , cujos elementos são alternadamente vértices e arestas e tal que, para todo  $1 \leq i \leq n$ ,  $v_{i-1}$  e  $v_i$  são extremos de  $e_i$ .



# Conceitos Básicos

## Caminhos em Grafos

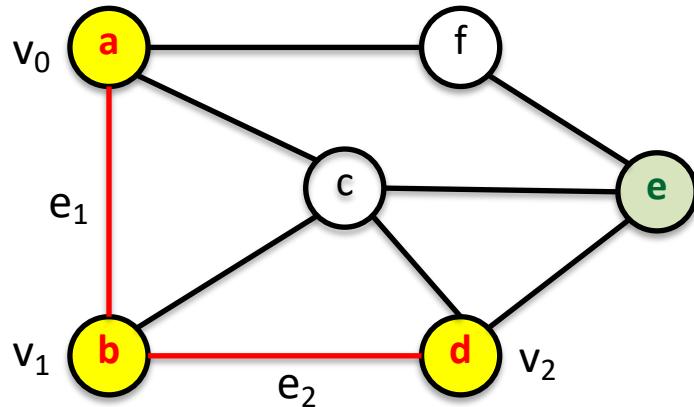
- Um **caminho** de  $v_0$  a  $v_n$  no grafo  $\mathbf{G}$  é uma sequência finita e não vazia ( $v_0, e_1, v_1, \dots, e_n, v_n$ ), cujos elementos são alternadamente vértices e arestas e tal que, para todo  $1 \leq i \leq n$ ,  $v_{i-1}$  e  $v_i$  são extremos de  $e_i$ .



# Conceitos Básicos

## Caminhos em Grafos

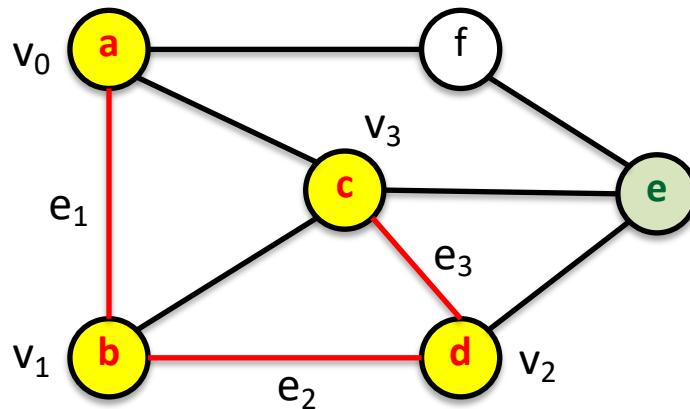
- Um **caminho** de  $v_0$  a  $v_n$  no grafo  $\mathbf{G}$  é uma sequência finita e não vazia ( $v_0, e_1, v_1, \dots, e_n, v_n$ ), cujos elementos são alternadamente vértices e arestas e tal que, para todo  $1 \leq i \leq n$ ,  $v_{i-1}$  e  $v_i$  são extremos de  $e_i$ .



# Conceitos Básicos

## Caminhos em Grafos

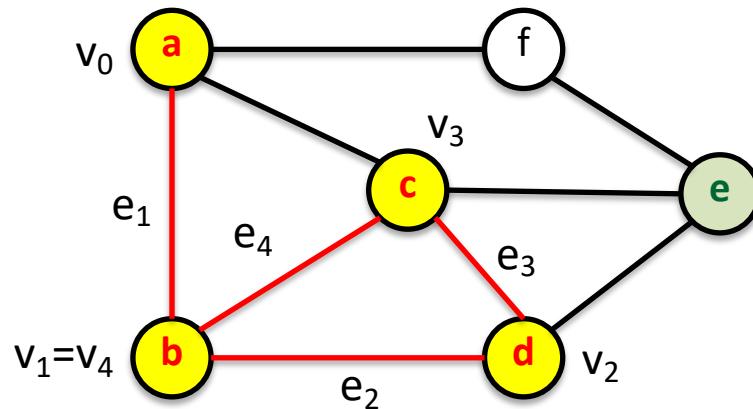
- Um **caminho** de  $v_0$  a  $v_n$  no grafo  $\mathbf{G}$  é uma sequência finita e não vazia ( $v_0, e_1, v_1, \dots, e_n, v_n$ ), cujos elementos são alternadamente vértices e arestas e tal que, para todo  $1 \leq i \leq n$ ,  $v_{i-1}$  e  $v_i$  são extremos de  $e_i$ .



# Conceitos Básicos

## Caminhos em Grafos

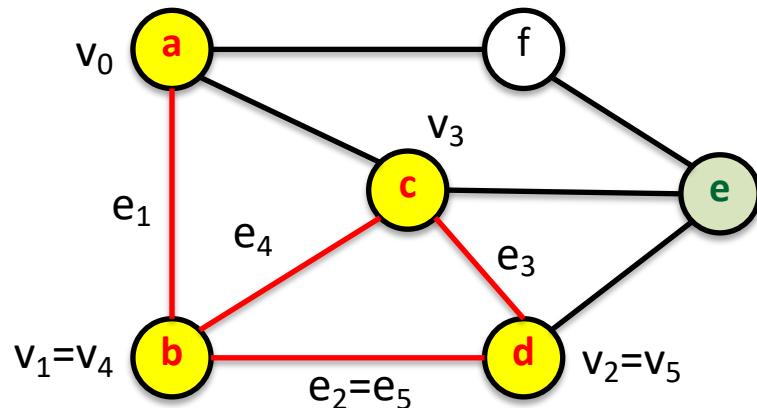
- Um **caminho** de  $v_0$  a  $v_n$  no grafo  $\mathbf{G}$  é uma sequência finita e não vazia ( $v_0, e_1, v_1, \dots, e_n, v_n$ ), cujos elementos são alternadamente vértices e arestas e tal que, para todo  $1 \leq i \leq n$ ,  $v_{i-1}$  e  $v_i$  são extremos de  $e_i$ .



# Conceitos Básicos

## Caminhos em Grafos

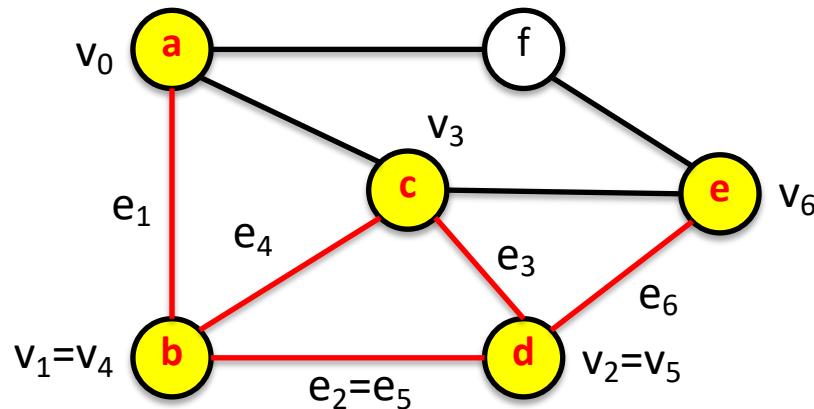
- Um **caminho** de  $v_0$  a  $v_n$  no grafo  $G$  é uma sequência finita e não vazia ( $v_0, e_1, v_1, \dots, e_n, v_n$ ), cujos elementos são alternadamente vértices e arestas e tal que, para todo  $1 \leq i \leq n$ ,  $v_{i-1}$  e  $v_i$  são extremos de  $e_i$ .



# Conceitos Básicos

## Caminhos em Grafos

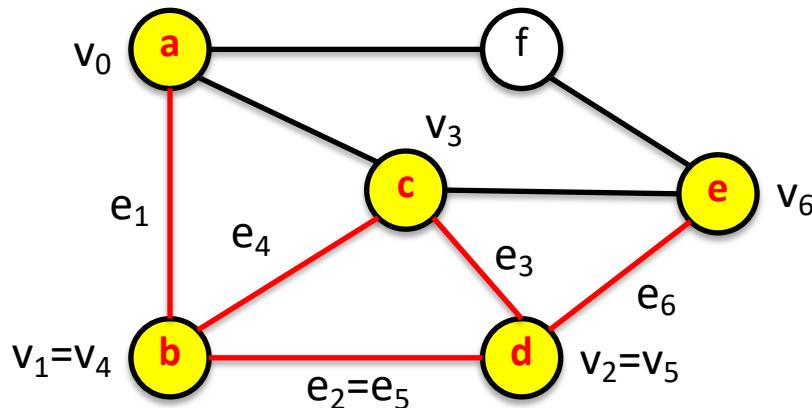
- Um **caminho** de  $v_0$  a  $v_n$  no grafo  $G$  é uma sequência finita e não vazia ( $v_0, e_1, v_1, \dots, e_n, v_n$ ), cujos elementos são alternadamente vértices e arestas e tal que, para todo  $1 \leq i \leq n$ ,  $v_{i-1}$  e  $v_i$  são extremos de  $e_i$ .



# Conceitos Básicos

## Comprimento de um Caminho

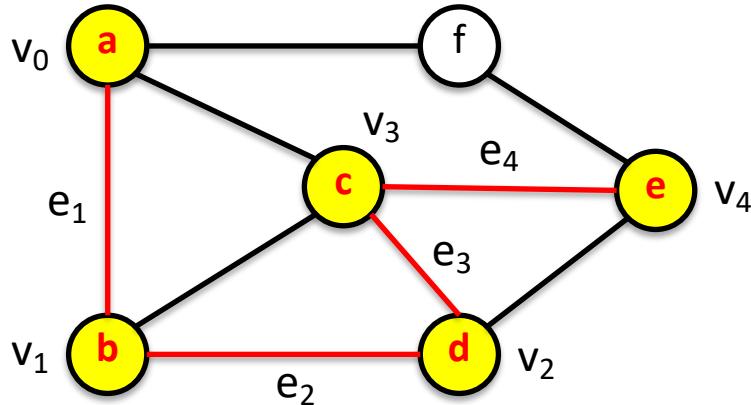
- O **comprimento** de um caminho  $(v_0, e_1, v_1, \dots, e_n, v_n)$  é dado pelo seu número de arestas, ou seja,  $n$ 
  - No exemplo abaixo, o comprimento do caminho é 6



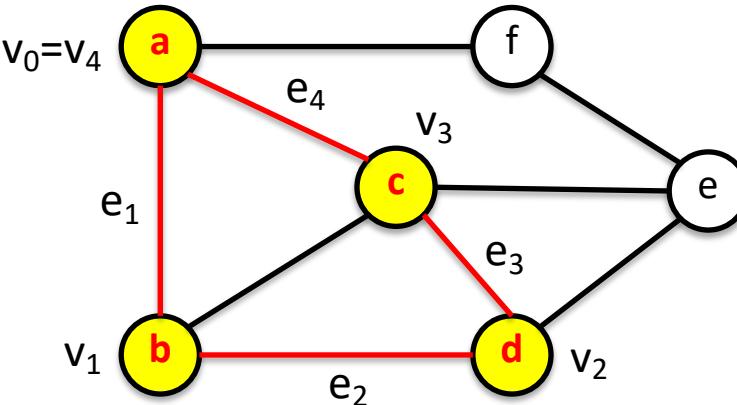
# Conceitos Básicos

## Caminhos Simples e Ciclos

- Um **caminho simples** é um caminho em que não há repetição de vértices e nem de arestas na sequência
- Um **ciclo** ou **caminho fechado** é um caminho em que  $v_0 = v_n$



Caminho Simples

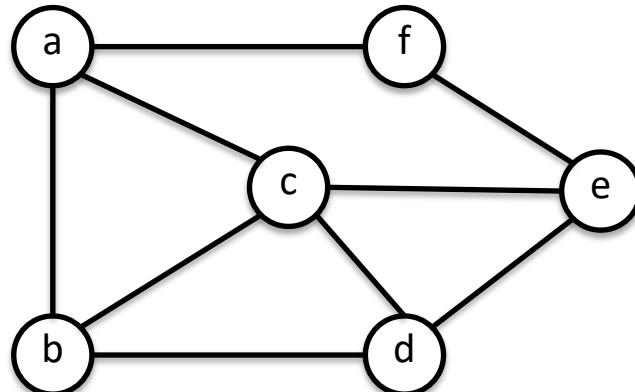


Ciclo

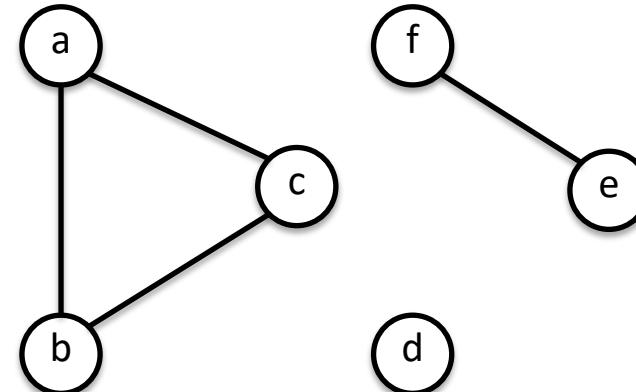
# Conceitos Básicos

## Grafo Conexo

- Um grafo é **conexo** se, para qualquer par de vértices **u** e **v** de **V**, existe um caminho de **u** a **v** em **G**
- Um grafo **G** que não é conexo pode ser partitionado em **componentes conexos**

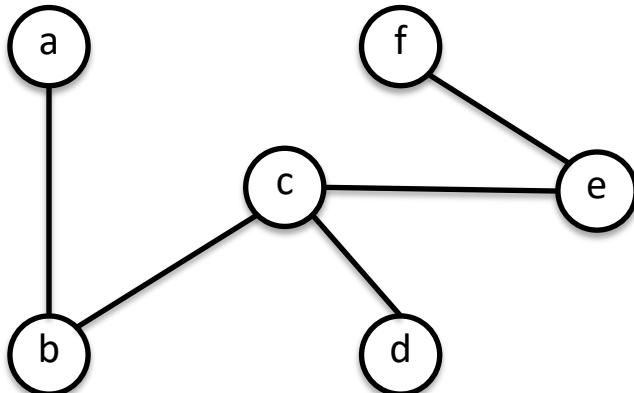


Conexo

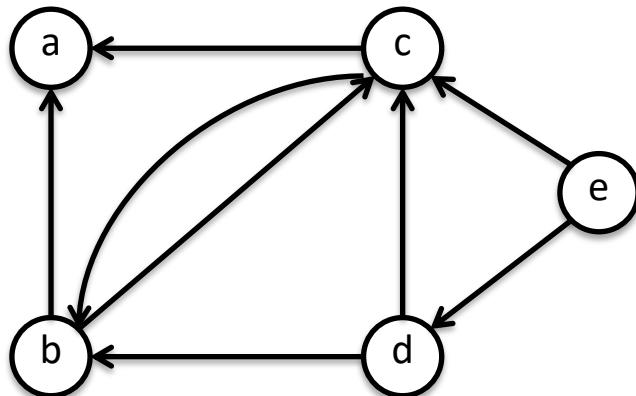


Não-Conexo com  
3 Componentes Conexos

- Um grafo **G** é uma **árvore** se for conexo e não possuir ciclos (acíclico)
- Uma árvore possui exatamente  $|V| - 1$  arestas



- Em um **grafo não-orientado** não há ordenação especial no par de vértices que representa uma aresta
  - Assim, os pares  $(u, v)$  e  $(v, u)$  representam a mesma aresta
- Num **grafo orientado**, cada aresta é representada por um par ordenado de vértices
  - Assim, os pares  $(u, v)$  e  $(v, u)$  representam arestas distintas



- Se  $e=(u, v)$  é uma aresta de um grafo orientado  $\mathbf{G}$ , então dizemos que  $e$  sai de  $u$  e entra em  $v$
- O grau de saída,  $d^+(v)$ , de um vértice  $v$  é o número de arestas que saem de  $v$
- O grau de entrada,  $d^-(v)$ , de um vértice  $v$  é o número de arestas que entram em  $v$

## Teorema

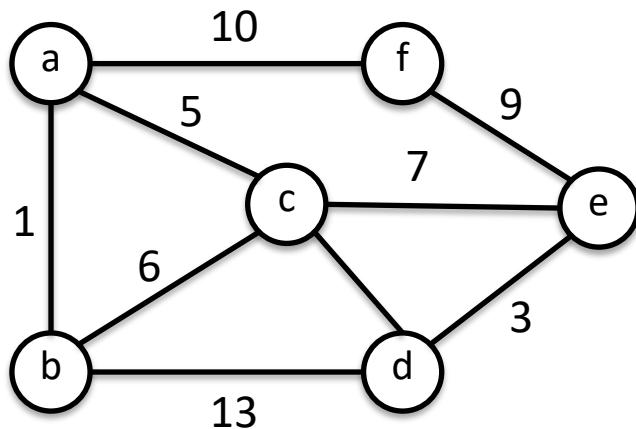
Para todo grafo orientado  $\mathbf{G}=(V, E)$ , tem-se que:

$$\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v) = |E|$$

# Conceitos Básicos

## Grafo Ponderado

- Um grafo (orientado ou não) é **ponderado** se a cada aresta **e** do grafo está associado um valor real **c(e)**, denominado de **custo** (ou **peso**) da aresta



# TAD GRAFO

- **O que o TAD Grafo deveria conter?**
  - Representação do tipo do grafo
  - Conjunto de operações que atuam sobre o grafo
  
- **Quais operações deveriam fazer parte da TAD?**

- **O que o TAD Grafo deveria conter?**
  - Representação do tipo do grafo
  - Conjunto de operações que atuam sobre o grafo
  
- **Quais operações deveriam fazer parte da TAD?**

**O conjunto de operações a ser definido  
depende de cada aplicação**

- Um conjunto de operações necessário a uma maioria de aplicações é:
  1. Criar um grafo vazio
  2. Inserir uma aresta no grafo
  3. Retirar uma aresta do grafo
  4. Verificar se existe uma dada aresta no grafo
  5. Obter a lista de vértices adjacentes a um dado vértice

# Exemplo de Protótipo

## ■ Exemplo de Conjunto de Operações

- **TGrafo\_Inicia(Grafo, n)**: Inicia um grafo de ordem  $n$
- **TGrafo\_ExisteAresta(Grafo, u, v)**: Retorna *true* se existe a aresta  $e=(u, v)$  no grafo; caso contrário, retorna *false*
- **TGrafo\_InsereAresta(Grafo, u, v, e)**: Insere a aresta  $e$  incidente aos vértices  $u$  e  $v$  no grafo
- **TGrafo\_RetiraAresta(Grafo, u, v, e)**: Retorna a aresta  $e$  incidente aos vértices  $u$  e  $v$  no grafo, retirando-a do grafo
- **TGrafo\_ListaAdj(Grafo, u)**: Retorna a lista de vértices adjacentes ao vértice  $u$  no grafo
- **TGrafo\_NVertices(Grafo)**: Retorna o número de vértices do grafo
- **TGrafo\_NArestas(Grafo)**: Retorna o número de arestas do grafo

# Implementação de um Grafo

- A complexidade dos algoritmos para solução de problemas modelados por grafos depende fortemente da sua representação interna
- As duas representações mais utilizadas são:
  - Implementação por meio de **matrizes**
  - Implementação por meio de **listas lineares**
- A escolha de uma determinada representação depende da aplicação que se tem em vista e das operações que se espera realizar no grafo

# Implementação de um Grafo

- Há outras alternativas para representar grafos, mas matrizes e listas lineares são as mais usadas
- Elas podem ser adaptadas para representar grafos ponderados, grafos com laços e arestas múltiplas, etc
- Em alguns problemas é essencial ter estruturas de dados adicionais para melhorar a eficiência dos algoritmos

# IMPLEMENTAÇÃO POR MATRIZES

- Seja  $\mathbf{G}=(\mathbf{V}, \mathbf{E})$  um grafo simples (orientado ou não)
- Os conjuntos de vértices  $\mathbf{V}$  e arestas  $\mathbf{E}$  do grafo  $\mathbf{G}$  estão associados aos elementos de uma matriz
- Existem duas opções básicas de representação:
  - **Matriz de Adjacência**
  - **Matriz de Incidência**

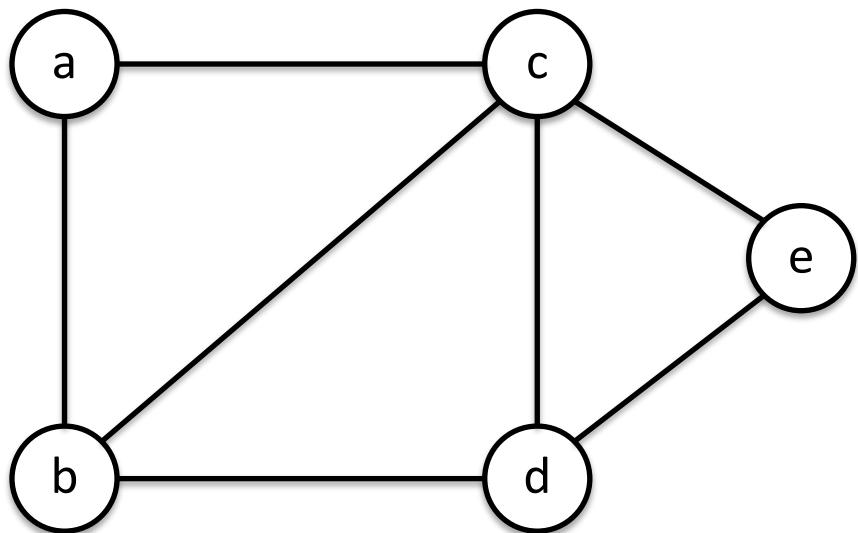
# Matriz de Adjacência

- A matriz de adjacência de  $\textcolor{red}{G}$  é uma matriz quadrada  $\textcolor{red}{A}$  de ordem  $|\textcolor{red}{V}|$ , cujas linhas e colunas são indexadas pelos vértices em  $\textcolor{red}{V}$ , tal que:

$$A[i, j] = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{caso contrário} \end{cases}$$

# Matriz de Adjacência

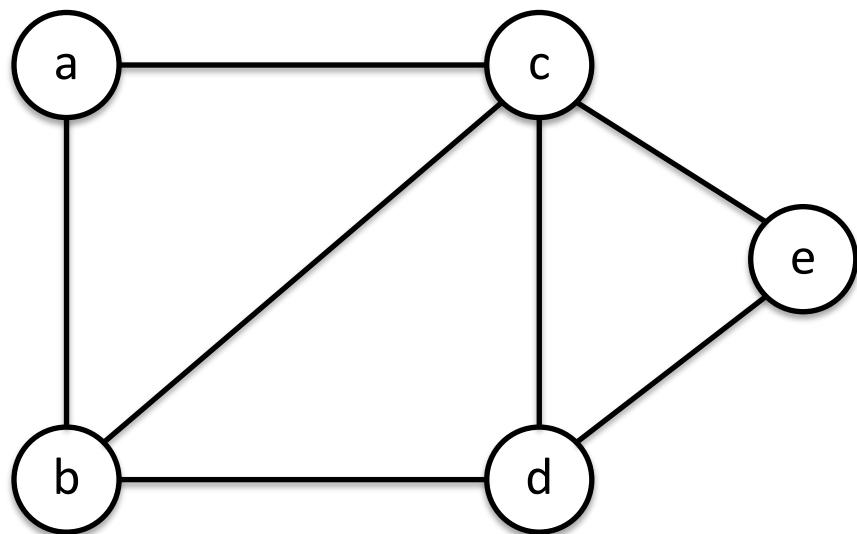
- Exemplo de um grafo **não orientado** e a sua matriz de adjacência correspondente



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	1	0
c	1	1	0	1	1
d	0	1	1	0	1
e	0	0	1	1	0

# Matriz de Adjacência

- Exemplo de um grafo **não orientado** e a sua matriz de adjacência correspondente

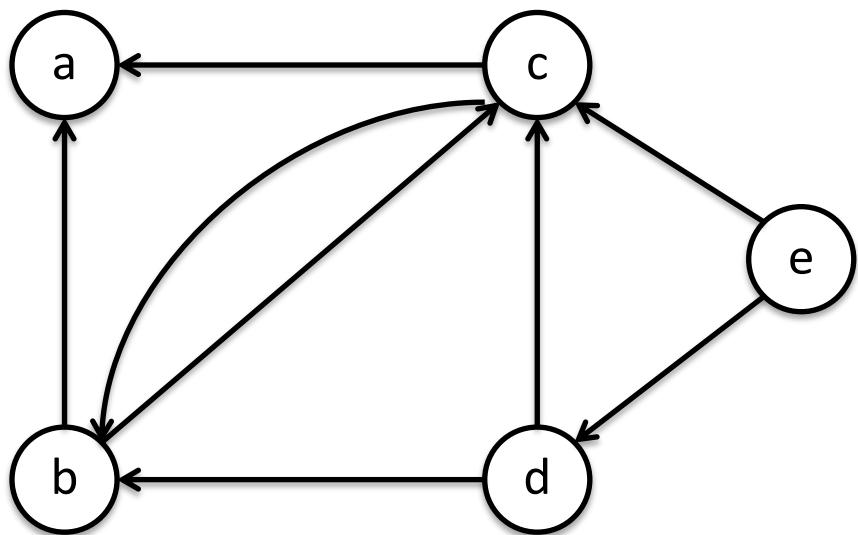


	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	1	0
c	1	1	0	1	1
d	0	1	1	0	1
e	0	0	1	1	0

Se **G** é **não orientado**, então a matriz **A** é **simétrica**

# Matriz de Adjacência

- Exemplo de um grafo **orientado** e a sua matriz de adjacência correspondente



	a	b	c	d	e
a	0	0	0	0	0

# Estrutura da Matriz de Adjacência

```
#define MAXVERTICES 100

typedef int TVertice;

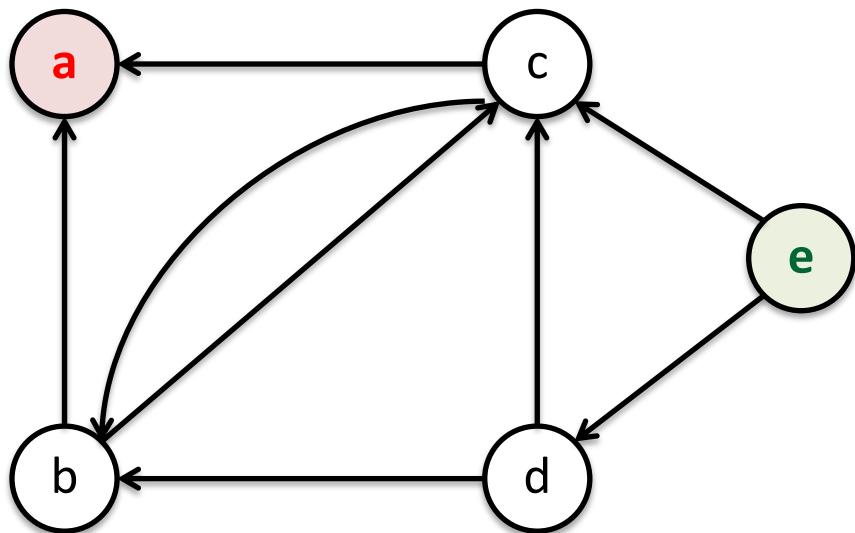
typedef int TAresta;

typedef struct {
    int IncideAresta;
    TAresta Aresta;
} TAdjacencia;

typedef struct {
    TAdjacencia Adj [MAXVERTICES] [MAXVERTICES];
    int NVertices;
    int NAreastas;
} TGrafo;
```

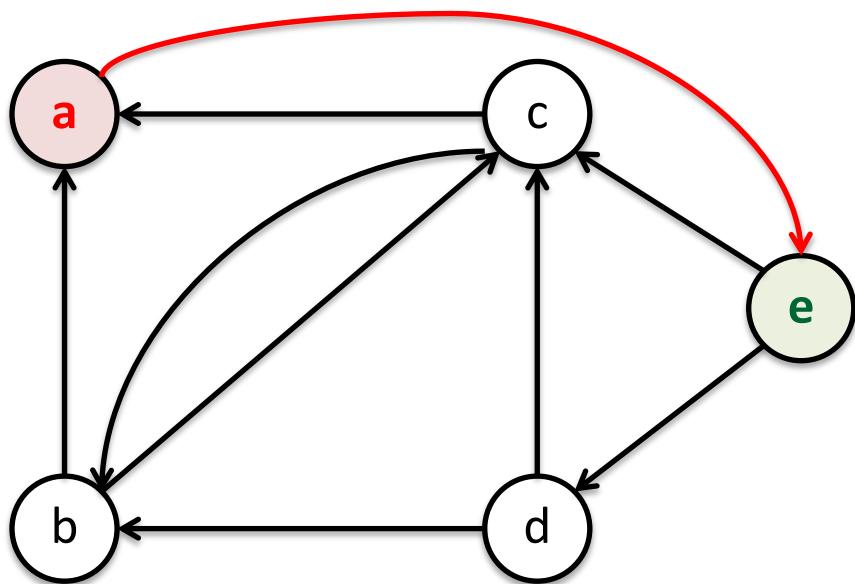


# Inserção de Arestas no Grafo



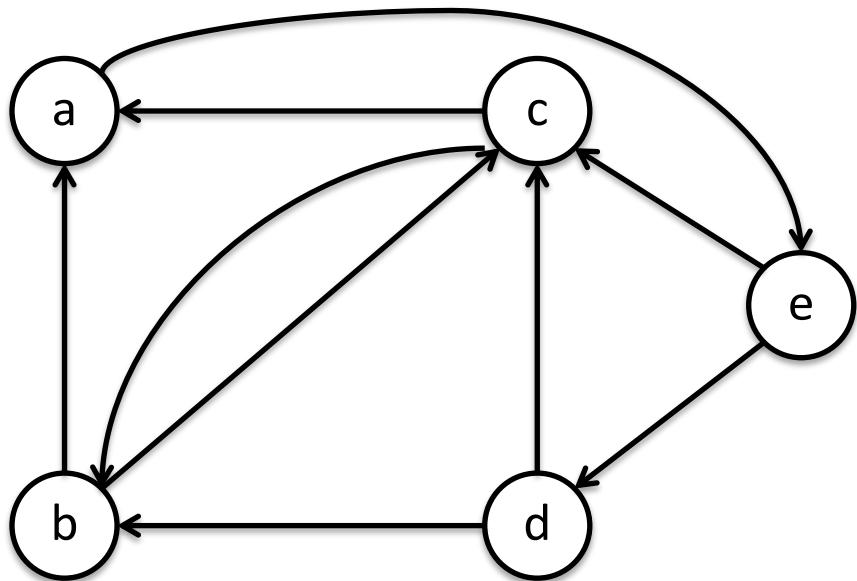
	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Inserção de Arestas no Grafo



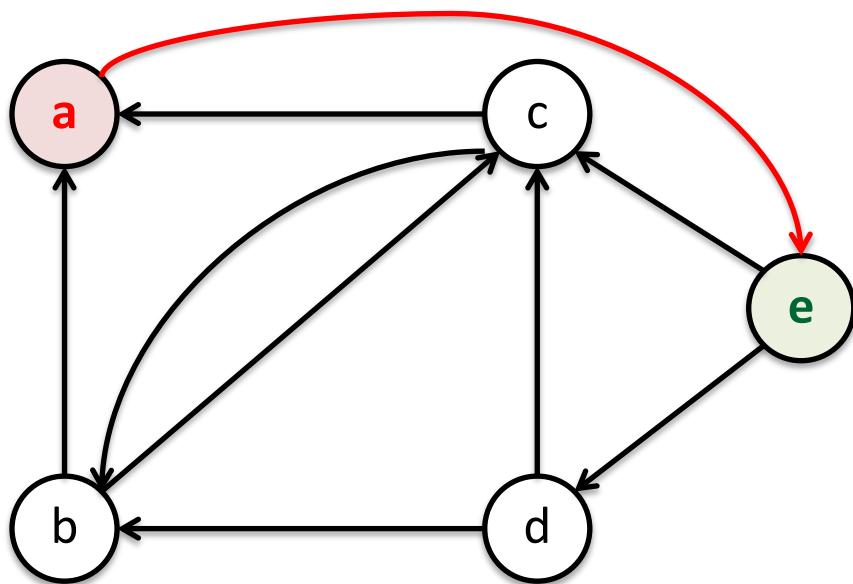
	a	b	c	d	e
a	0	0	0	0	1
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Inserção de Arestas no Grafo



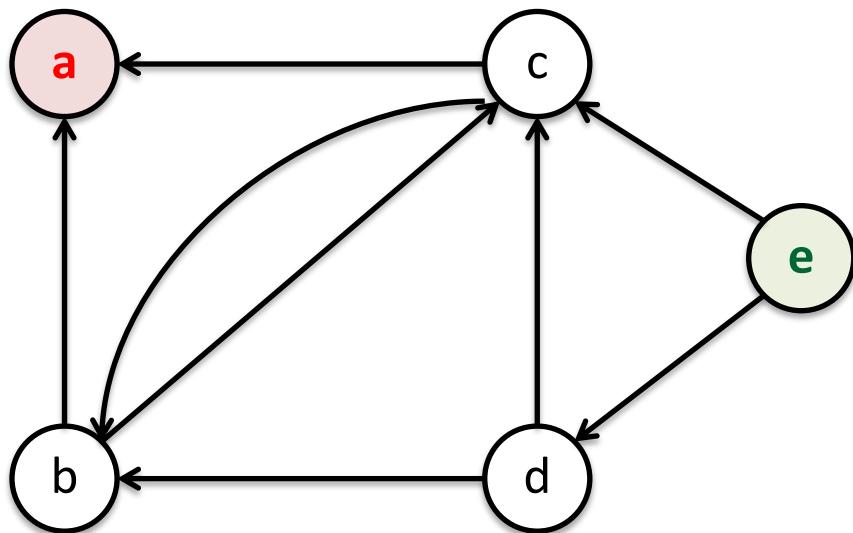
	a	b	c	d	e
a	0	0	0	0	1
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Retirada de Arestas do Grafo



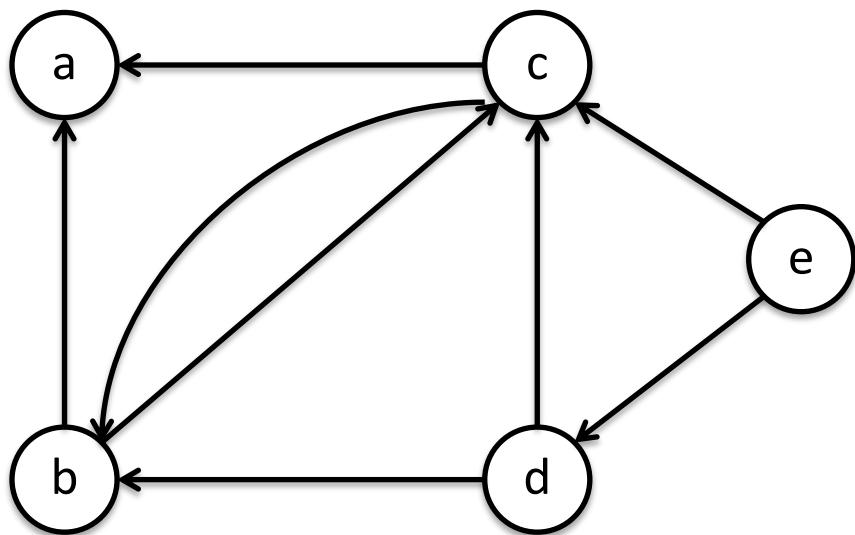
	a	b	c	d	e
a	0	0	0	0	1
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Retirada de Arestas do Grafo



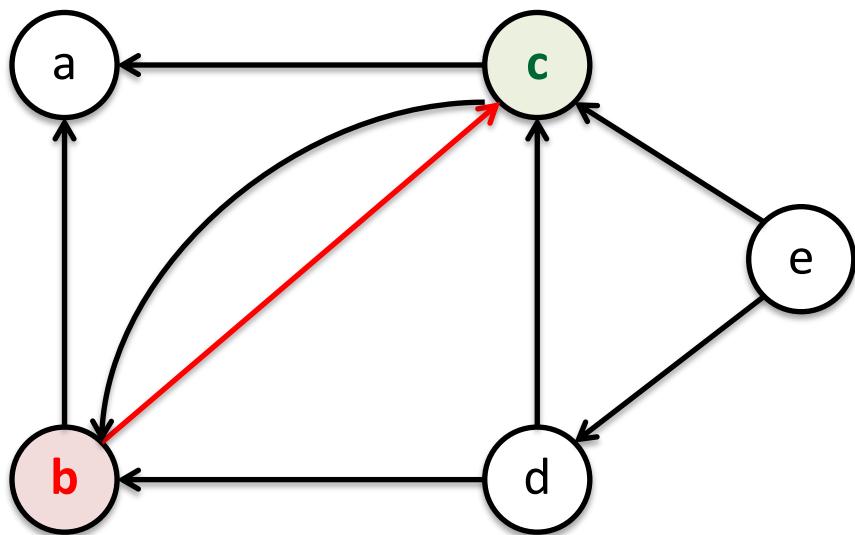
	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Retirada de Arestas do Grafo



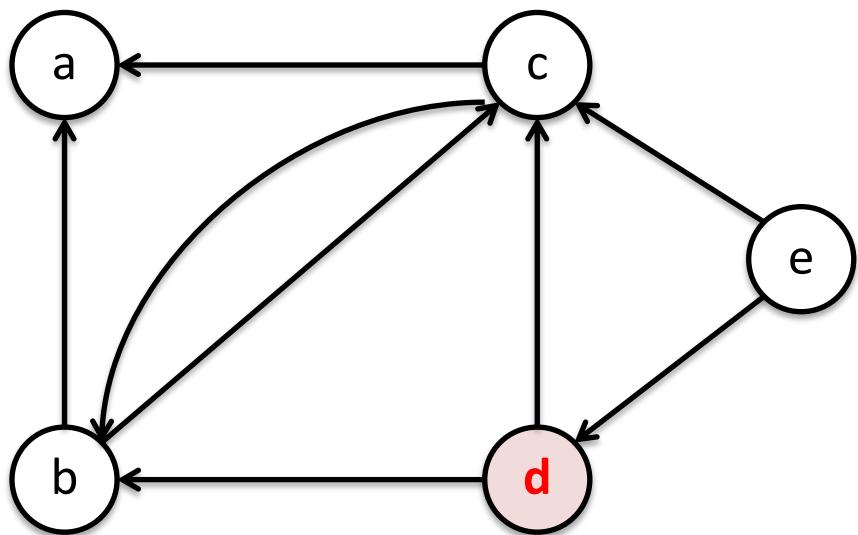
	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Verifica Existência de Aresta



	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

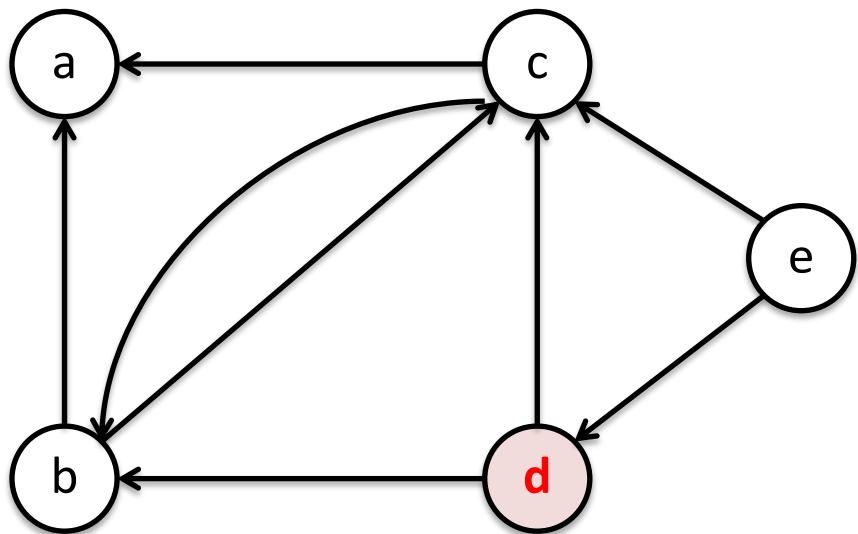
# Lista os Vértices Adjacentes



	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Lista os Vértices Adjacentes

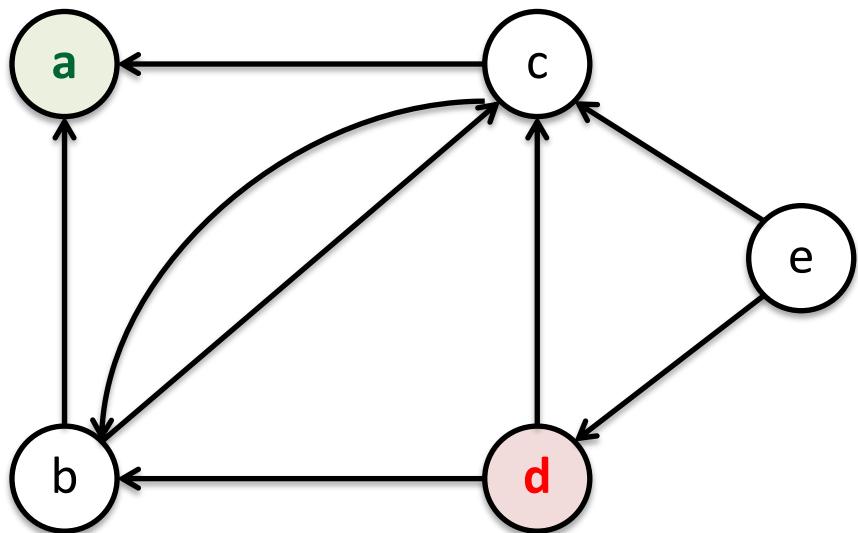
ListaAdj /



	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

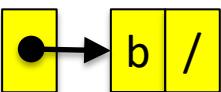
# Lista os Vértices Adjacentes

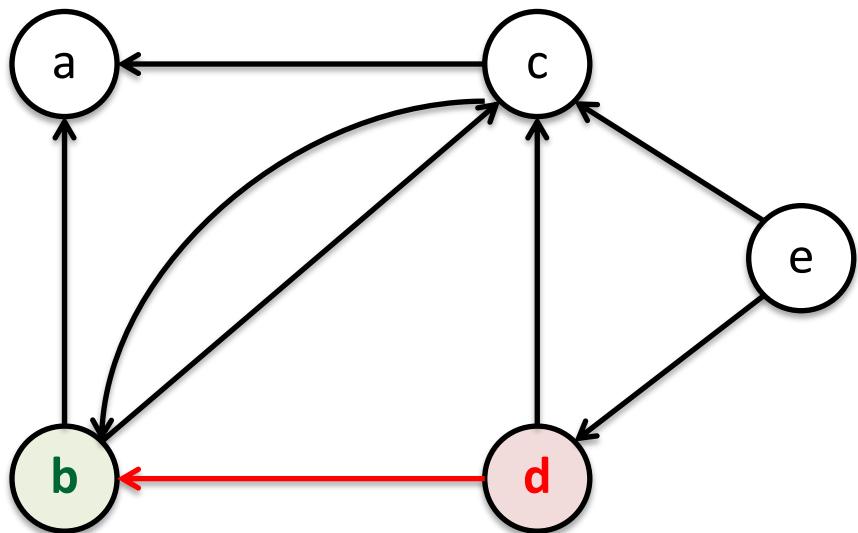
ListaAdj /



	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

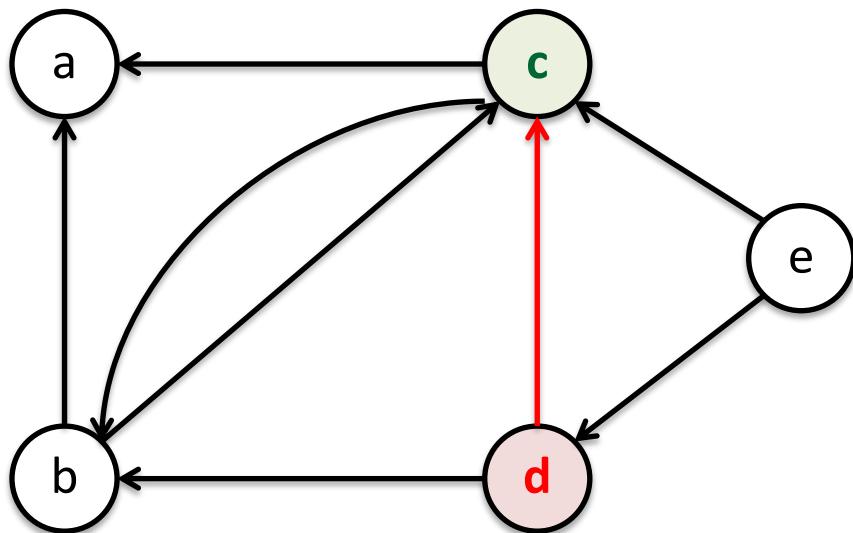
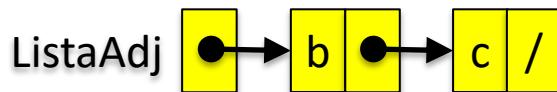
# Lista os Vértices Adjacentes

ListaAdj    



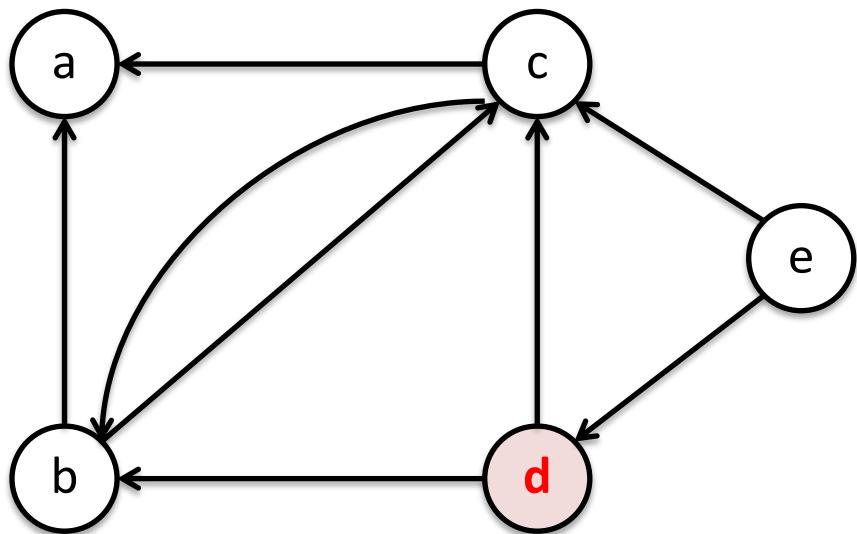
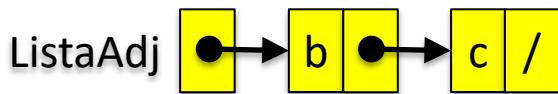
	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Lista os Vértices Adjacentes



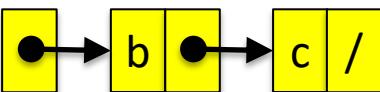
	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

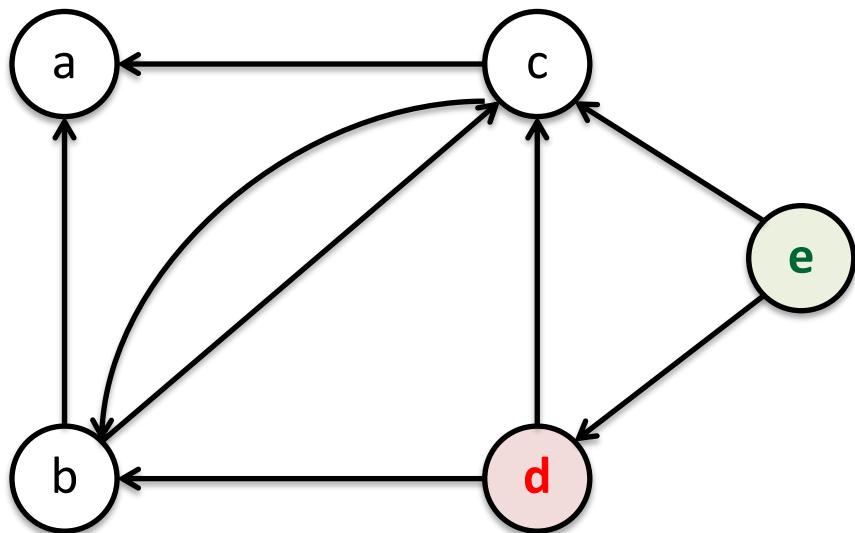
# Lista os Vértices Adjacentes



	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

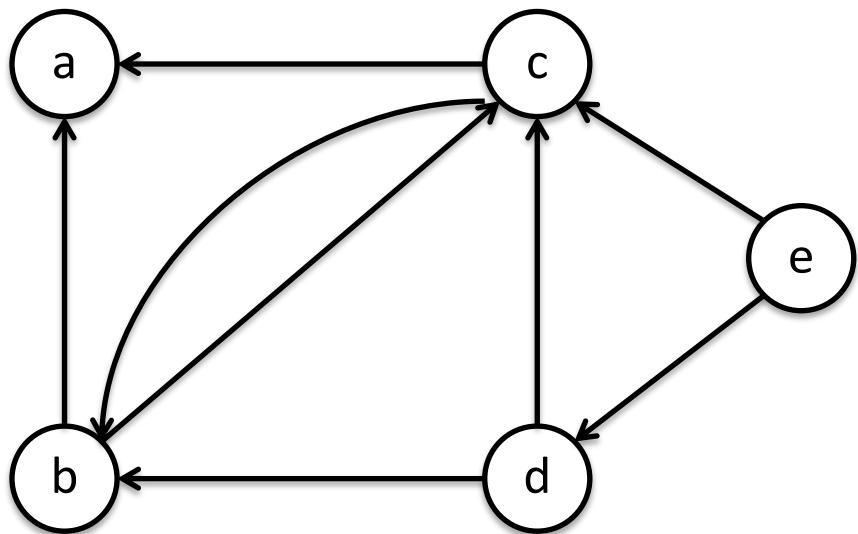
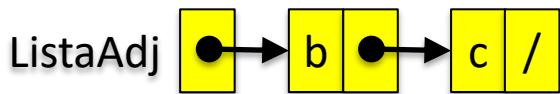
# Lista os Vértices Adjacentes

ListaAdj    



	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Lista os Vértices Adjacentes



	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

# Operações na Matriz de Adjacência

```
/* Inicia as variaveis do grafo */
int TGrafo_Inicia(TGrafo *pGrafo, int NVertices)
{
    TVertice u, v;

    if (NVertices > MAXVERTICES)
        return 0;

    pGrafo->NVertices = NVertices;
    pGrafo->Narestas = 0;
    for (u = 0; u < pGrafo->NVertices; u++)
        for (v = 0; v < pGrafo->NVertices; v++)
            pGrafo->Adj [u] [v].IncideAresta = 0;

    return 1;
}

/* Retorna se existe a aresta (u, v) no grafo */
int TGrafo_ExisteAresta(TGrafo *pGrafo, TVertice u, TVertice v)
{
    return pGrafo->Adj [u] [v].IncideAresta;
}
```

# Operações na Matriz de Adjacência

```
/* Insere a aresta e incidente aos vertices u e v no grafo */
int TGrafo_InsereAresta(TGrafo *pGrafo, TVertice u, TVertice v, TAresta e)
{
    pGrafo->Adj[u][v].IncideAresta = 1;
    pGrafo->Adj[u][v].Aresta = e;
    pGrafo->NArestas++;
    return 1;
}

/* Retira a aresta e incidente aos vertices u e v no grafo */
int TGrafo_RetiraAresta(TGrafo *pGrafo, TVertice u, TVertice v, TAresta *pE)
{
    if (! TGrafo_ExisteAresta(pGrafo, u, v))
        return 0;

    *pE = pGrafo->Adj[u][v].Aresta;
    pGrafo->Adj[u][v].IncideAresta = 0;
    pGrafo->NArestas--;
    return 1;
}
```

# Operações na Matriz de Adjacência

```
/* Retorna a lista de adjacentes do vertice u no grafo */
TLista *TGrafo_ListaAdj(TGrafo *pGrafo, TVertice u)
{
    TLista *pLista;
    TVertice v;

    pLista = (TLista *) malloc(sizeof(TLista));
    TLista_Inicia(pLista);

    for (v = 0; v < pGrafo->NVertices; v++)
        if (TGrafo_ExisteAresta(pGrafo, u, v))
            TLista_Insere(pLista, TLista_Tamanho(pLista), v);

    return pLista;
}

/* Retorna o numero de vertices do grafo */
int TGrafo_NVertices(TGrafo *pGrafo)
{
    return (pGrafo->NVertices);
}

/* Retorna o numero de arestas do grafo */
int TGrafo_NArestas(TGrafo *pGrafo)
{
    return (pGrafo->NArestas);
}
```

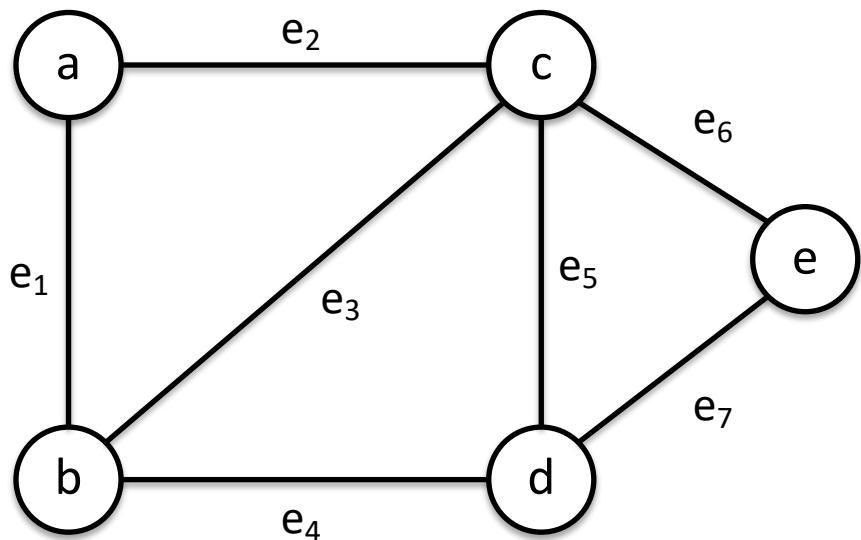
# Matriz de Incidência

- A matriz de incidência de  $\mathbf{G}$  é uma matriz  $\mathbf{A}$  de dimensão  $|V| \times |E|$ , cujas linhas são indexadas pelos vértices em  $V$  e a colunas são indexadas pelas arestas em  $E$ , tal que:

$$A[i, j] = \begin{cases} 1 & \text{se a aresta } j \in E \text{ é incidente ao vértice } i \in V \\ 0 & \text{caso contrário} \end{cases}$$

# Matriz de Incidência

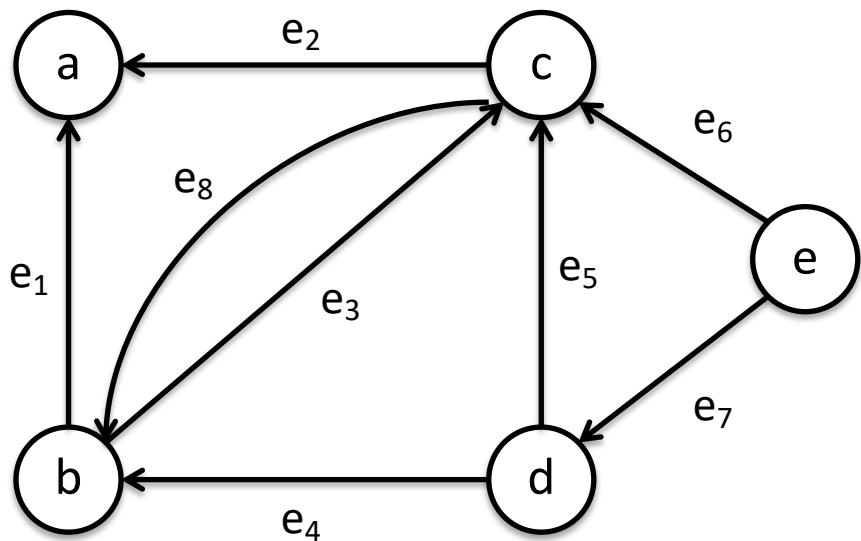
- Exemplo de um grafo **não orientado** e a sua matriz de incidência correspondente



	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	$e_6$	$e_7$
$a$	1	1	0	0	0	0	0
$b$	1	0	1	1	0	0	0
$c$	0	1	1	0	1	1	0
$d$	0	0	0	1	1	0	1
$e$	0	0	0	0	0	1	1

# Matriz de Incidência

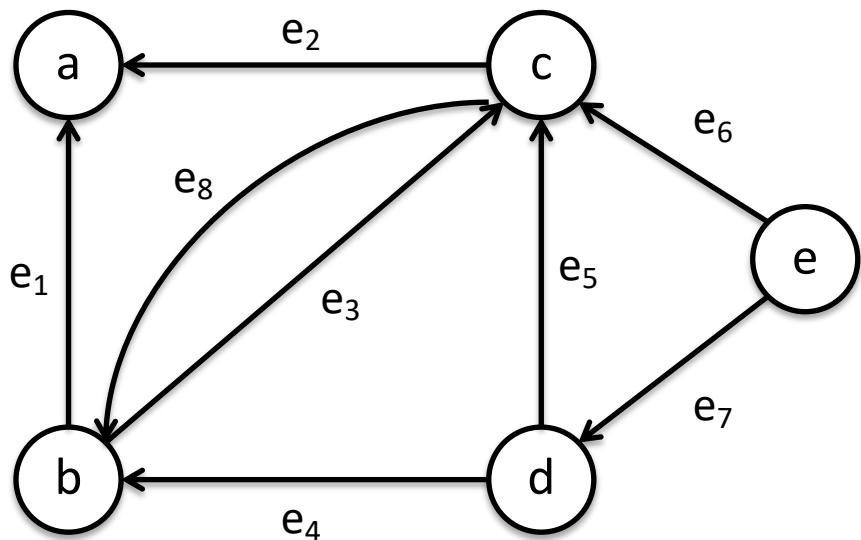
- Exemplo de um grafo **orientado** e a sua matriz de incidência correspondente



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Matriz de Incidência

- Exemplo de um grafo **orientado** e a sua matriz de incidência correspondente



	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	$e_6$	$e_7$	$e_8$
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	0	1	1

Se **G** é **orientado**, então +1 indica o vértice de **saída**  
e -1 indica o vértice de **entrada**

# Estrutura da Matriz de Incidência

```
#define MAXVERTICES 100
#define MAXARESTAS 5000

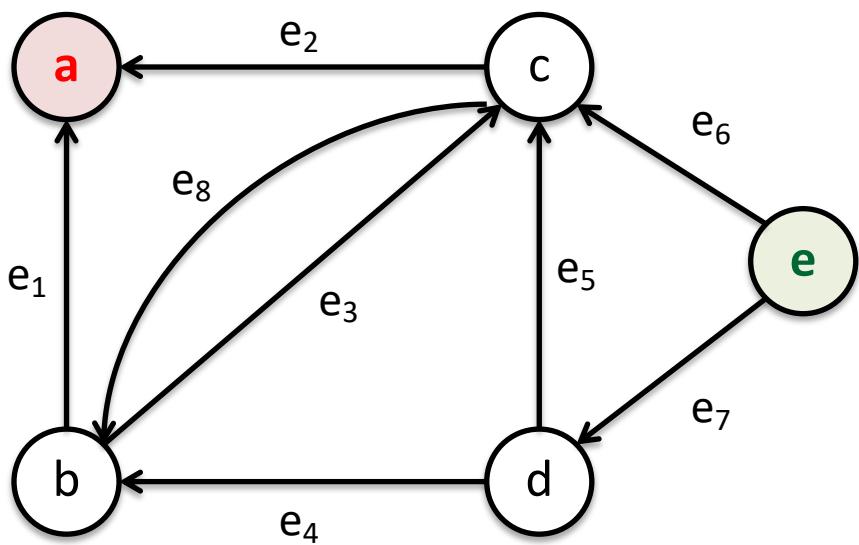
typedef int TVertice;

typedef int Taresta;

typedef struct {
    int IncideAresta;
    Taresta Aresta;
} TAdjacencia;

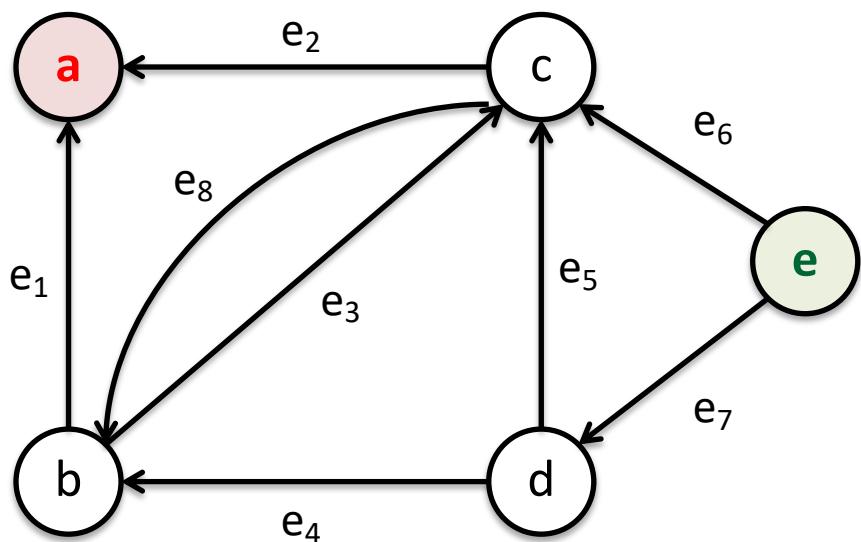
typedef struct {
    TAdjacencia Adj [MAXVERTICES] [MAXARESTAS];
    int NVertices;
    int Narestas;
} TGrafo;
```

# Inserção de Arestas no Grafo



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

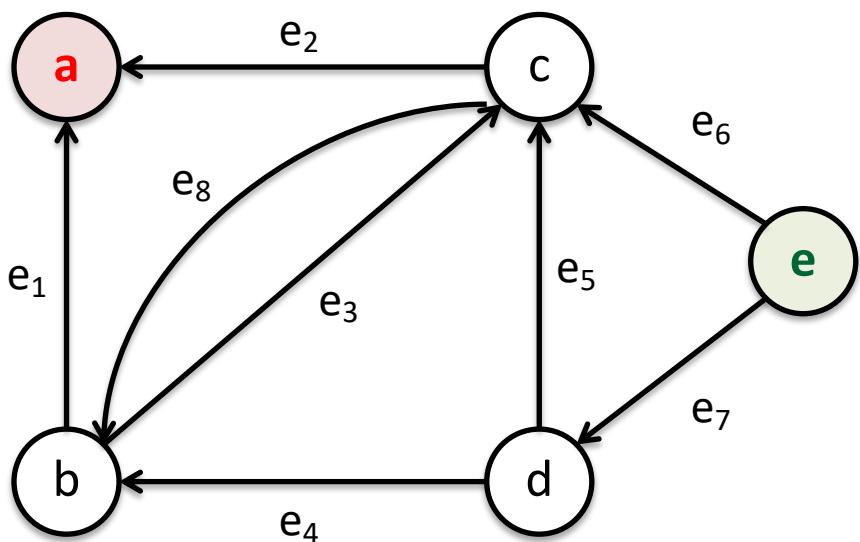
# Inserção de Arestas no Grafo



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

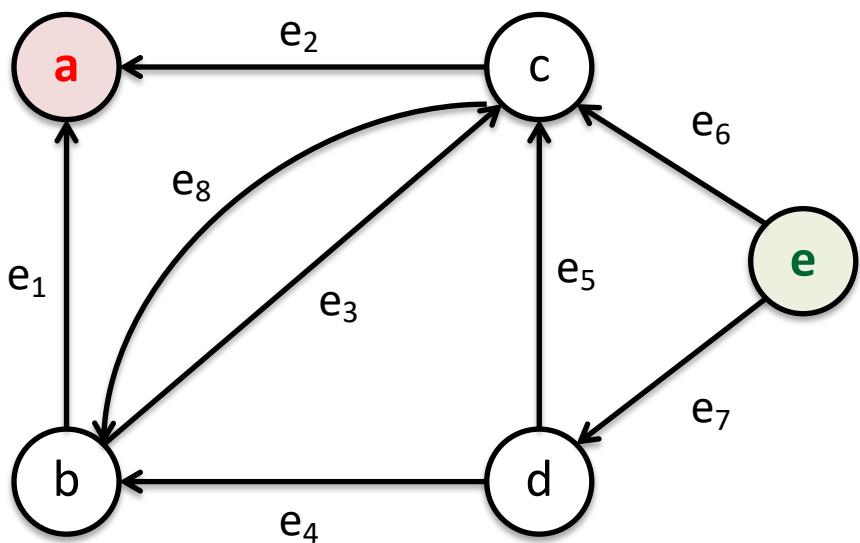
A matriz está cheia?

# Inserção de Arestas no Grafo



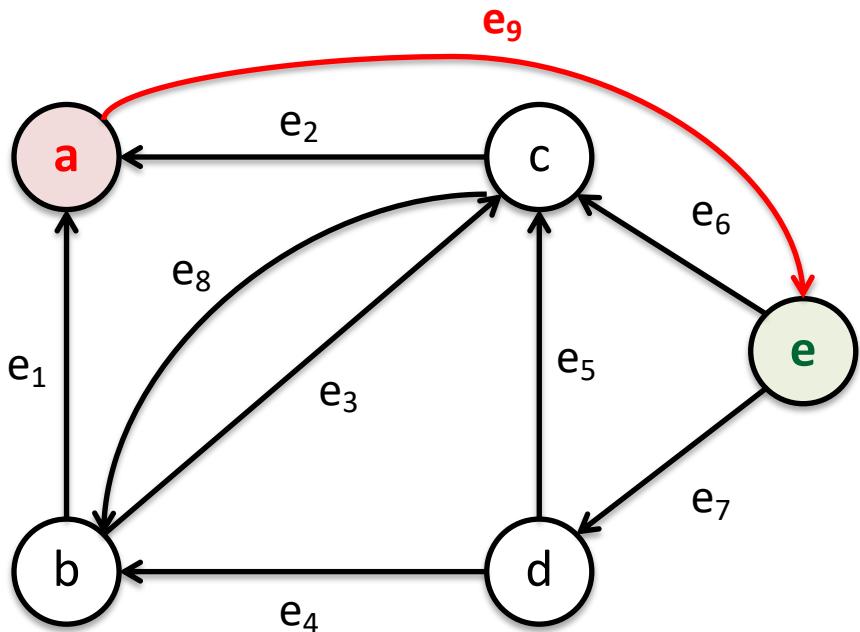
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	
b	1	0	1	-1	0	0	0	-1	
c	0	1	-1	0	-1	-1	0	1	
d	0	0	0	1	1	0	-1	0	
e	0	0	0	0	0	1	1	0	

# Inserção de Arestas no Grafo



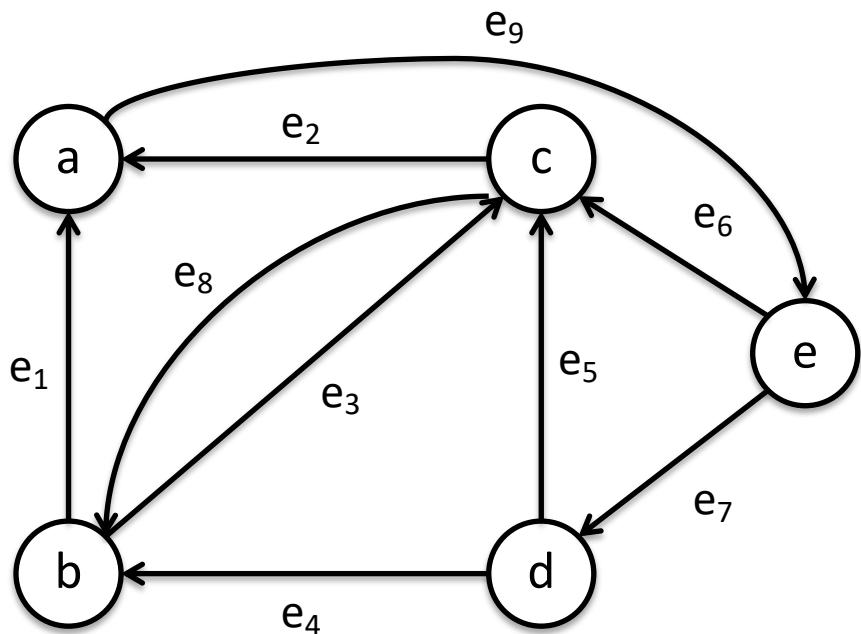
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	0

# Inserção de Arestas no Grafo



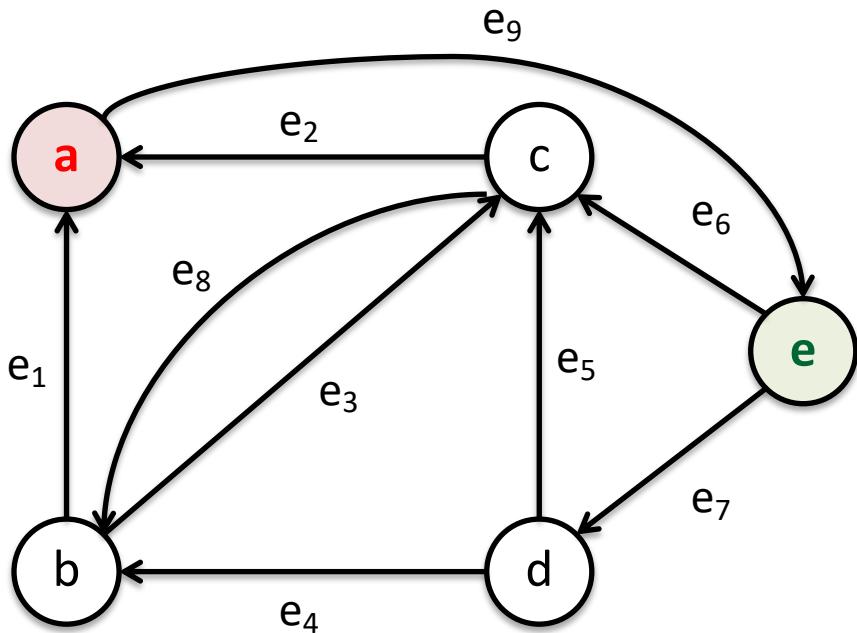
	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	$e_6$	$e_7$	$e_8$	$e_9$
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Inserção de Arestas no Grafo



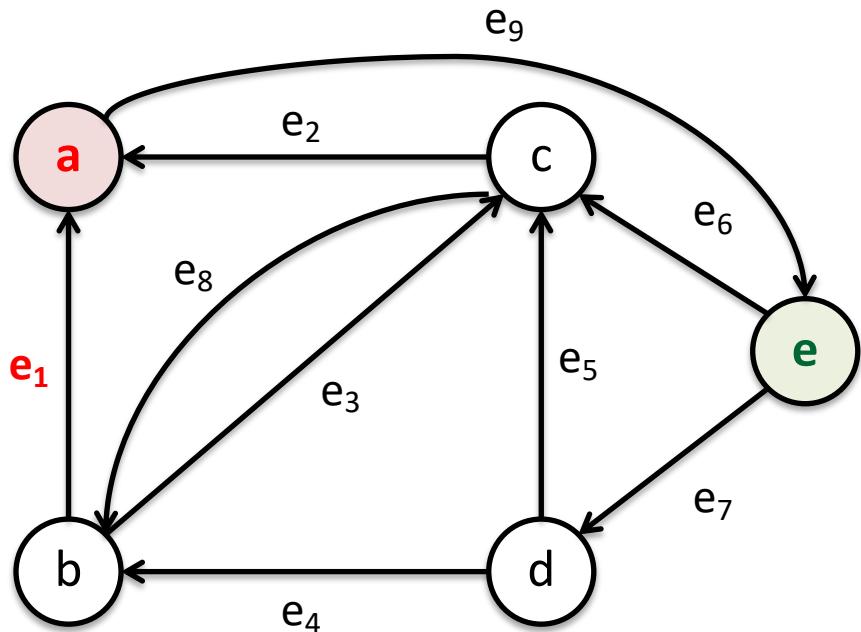
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	0	1	1	0

# Retirada de Arestas do Grafo



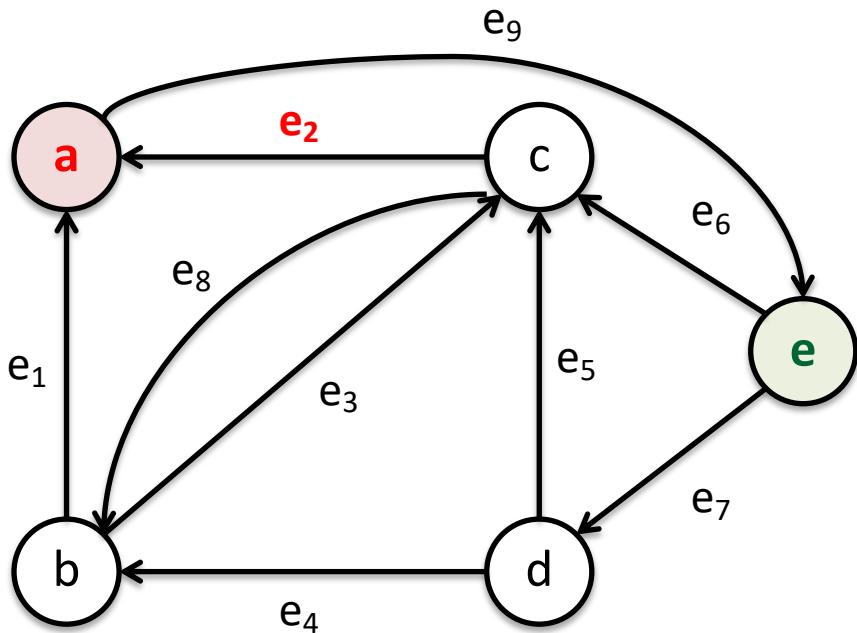
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Retirada de Arestas do Grafo



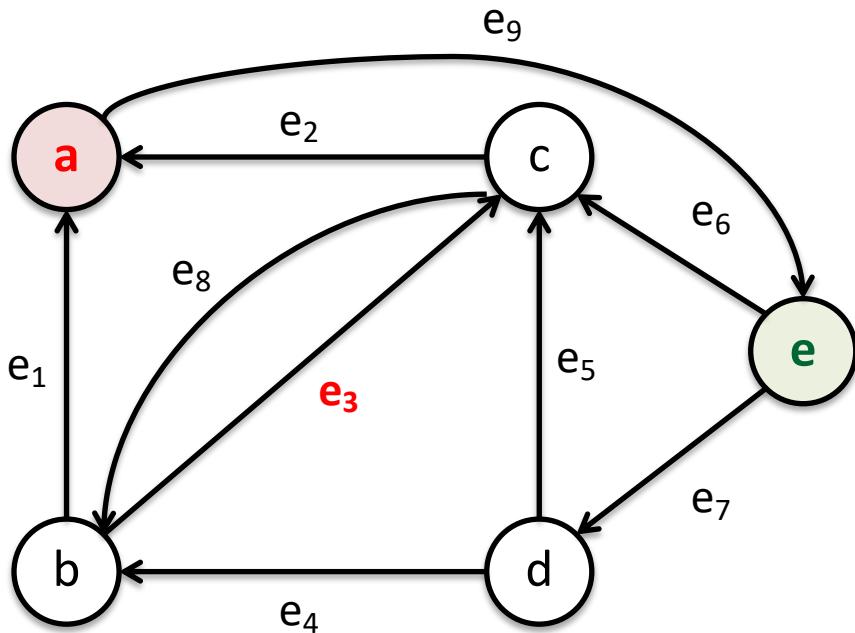
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Retirada de Arestas do Grafo



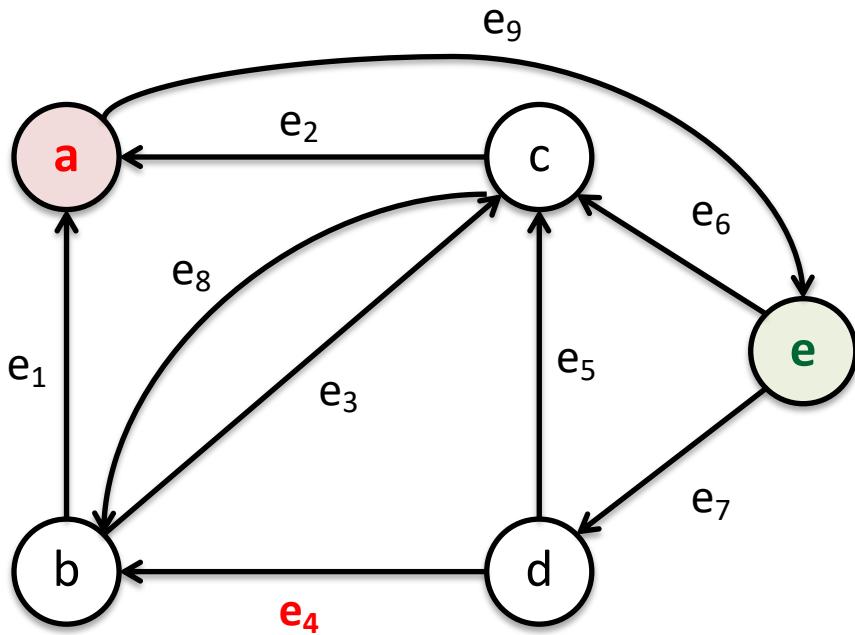
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Retirada de Arestas do Grafo



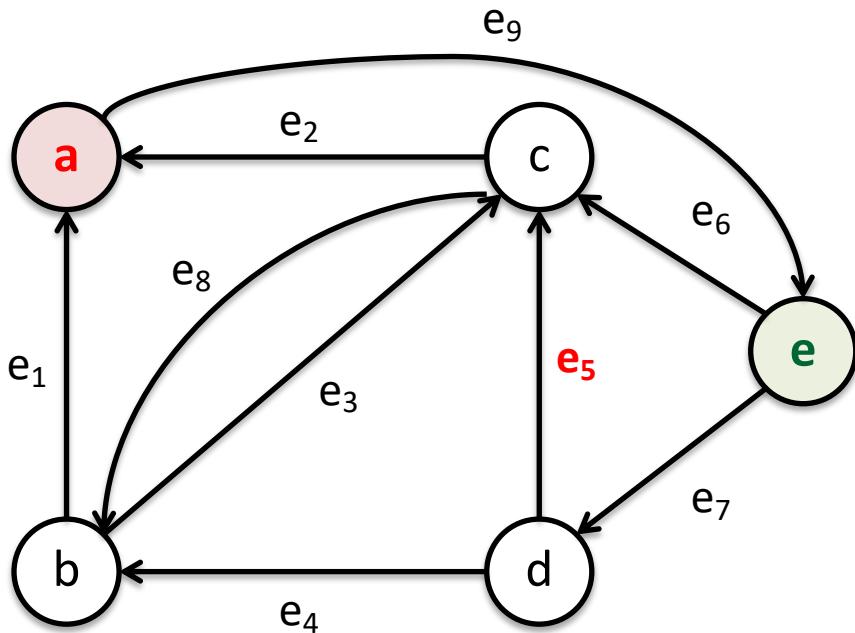
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Retirada de Arestas do Grafo



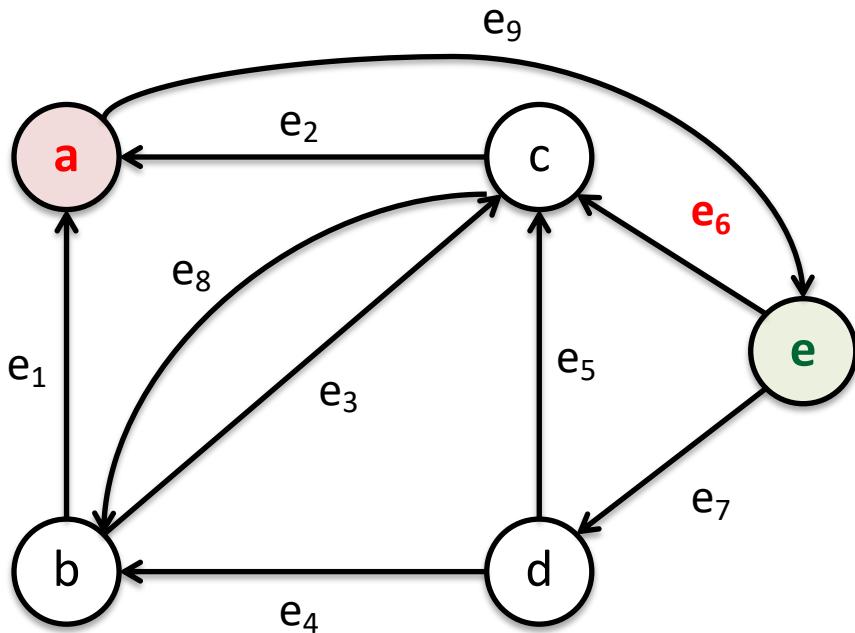
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Retirada de Arestas do Grafo



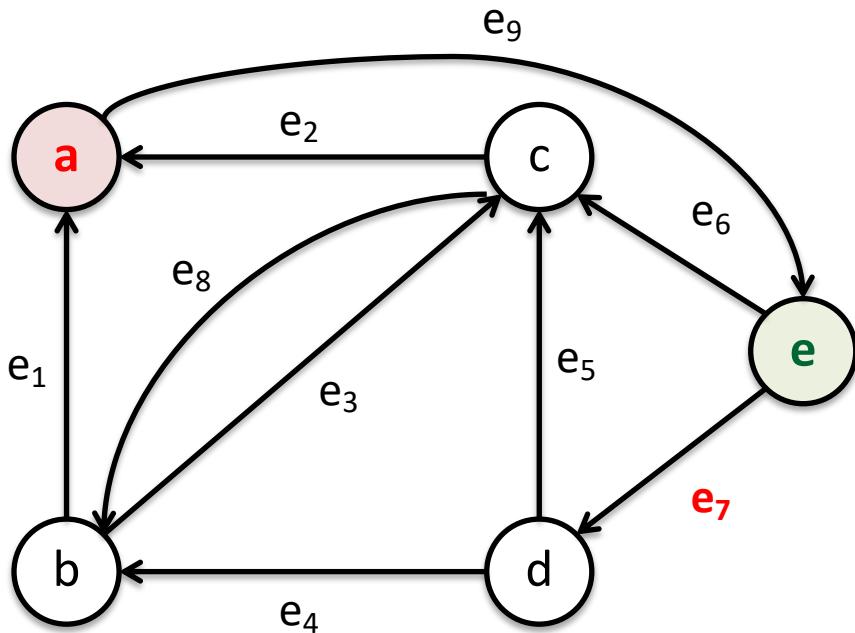
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Retirada de Arestas do Grafo



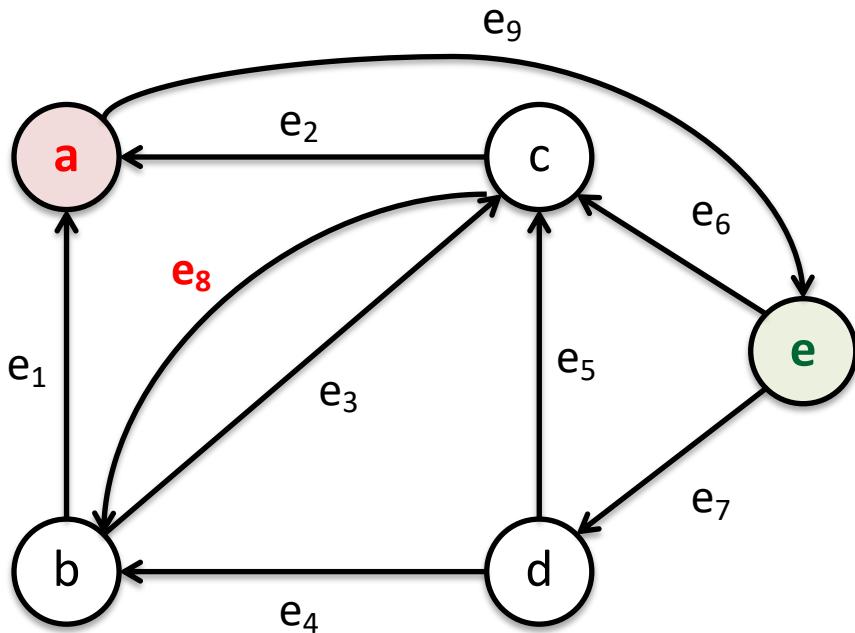
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	0	1	0	-1

# Retirada de Arestas do Grafo



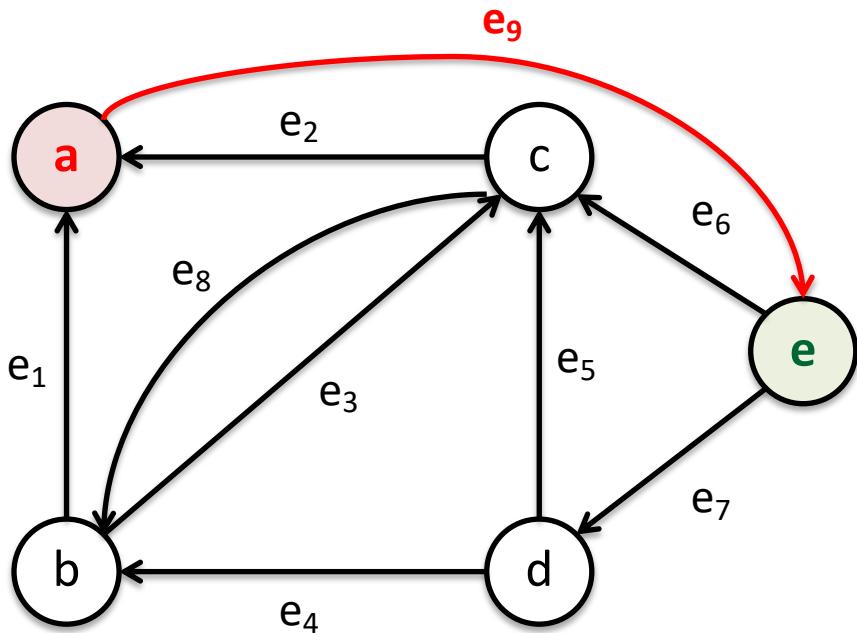
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Retirada de Arestas do Grafo



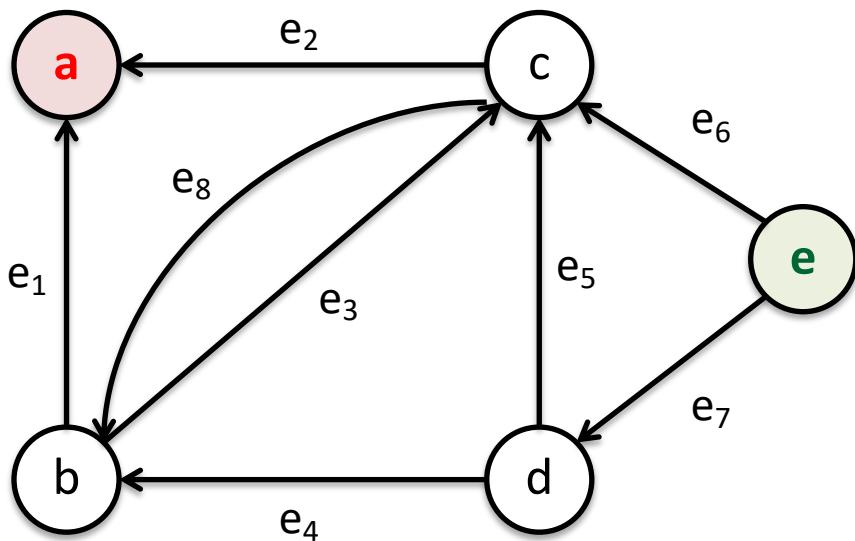
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Retirada de Arestas do Grafo



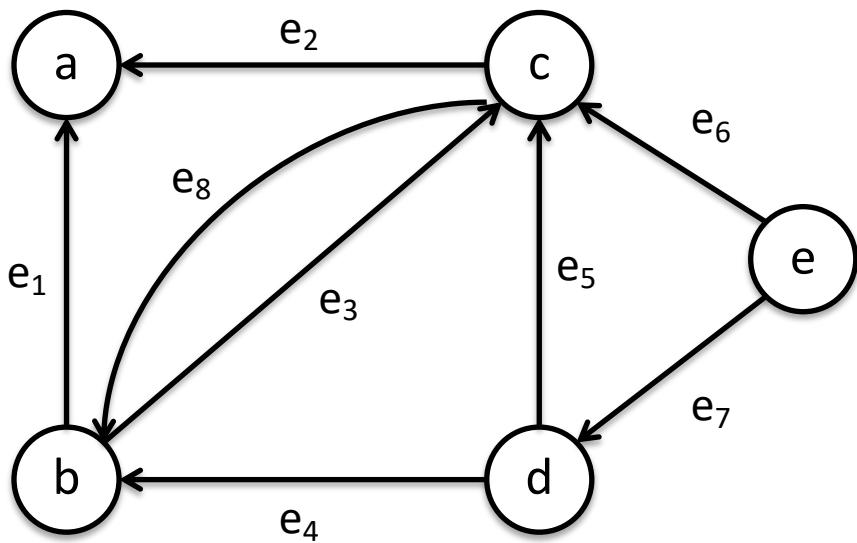
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>
a	-1	-1	0	0	0	0	0	0	1
b	1	0	1	-1	0	0	0	-1	0
c	0	1	-1	0	-1	-1	0	1	0
d	0	0	0	1	1	0	-1	0	0
e	0	0	0	0	0	1	1	0	-1

# Retirada de Arestas do Grafo



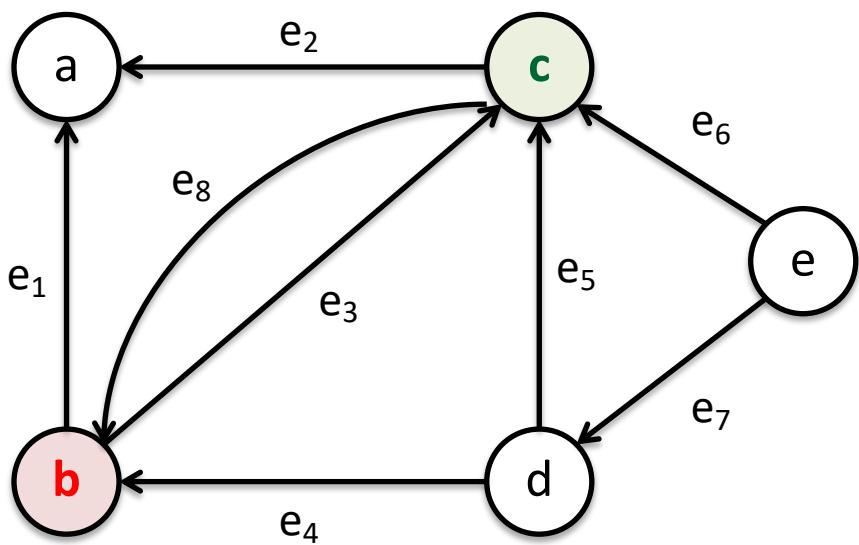
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Retirada de Arestas do Grafo



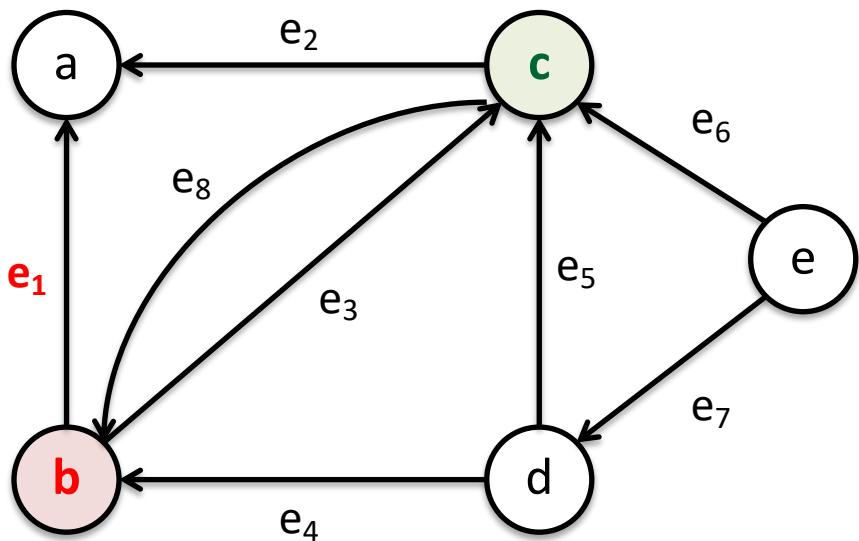
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Verifica Existência de Aresta



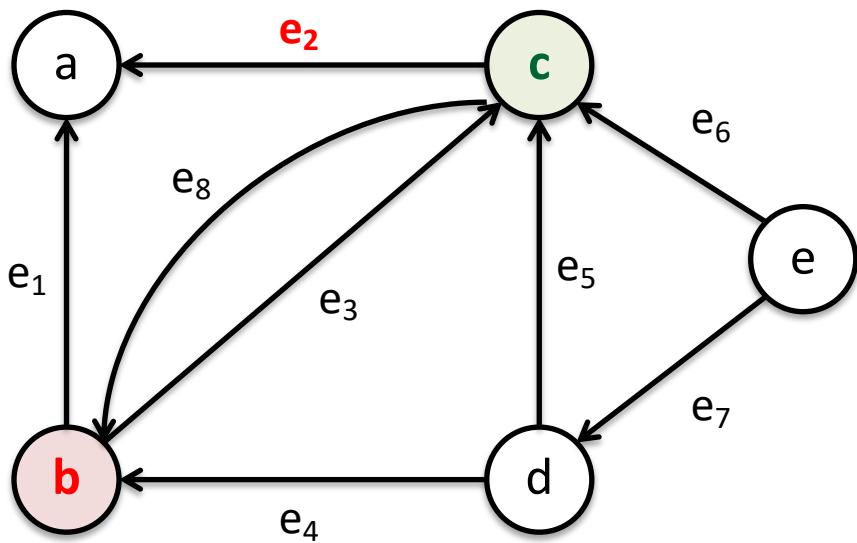
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Verifica Existência de Aresta



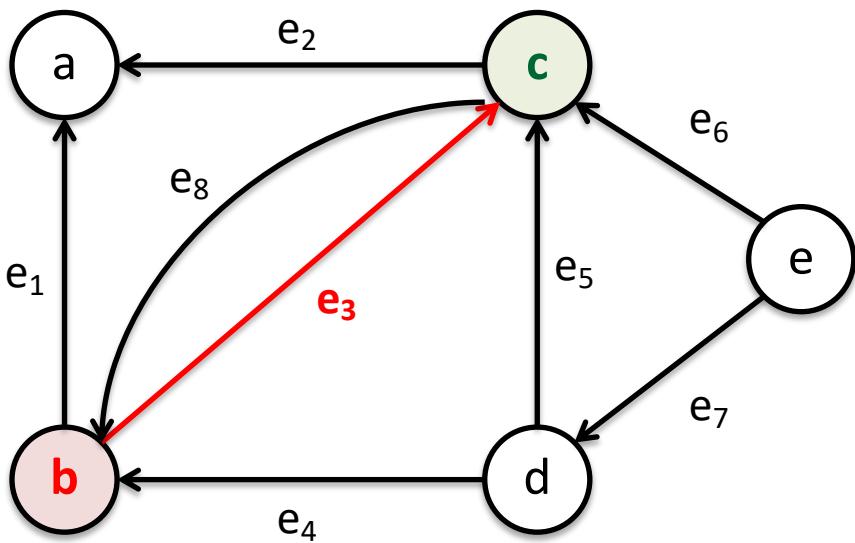
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Verifica Existência de Aresta



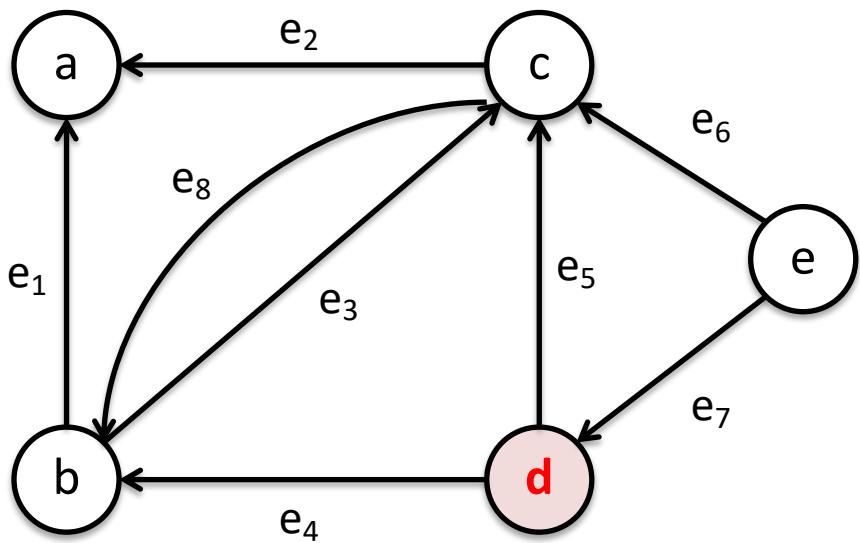
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Verifica Existência de Aresta



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

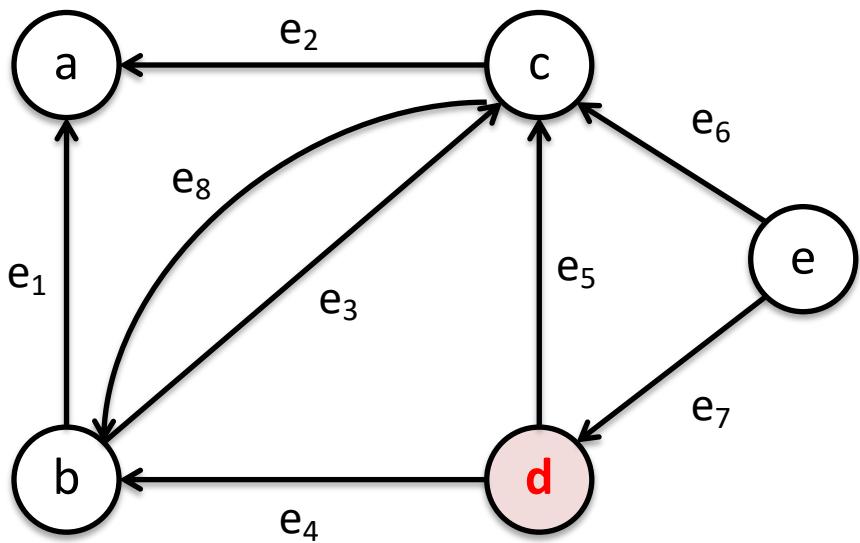
# Lista os Vértices Adjacentes



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes

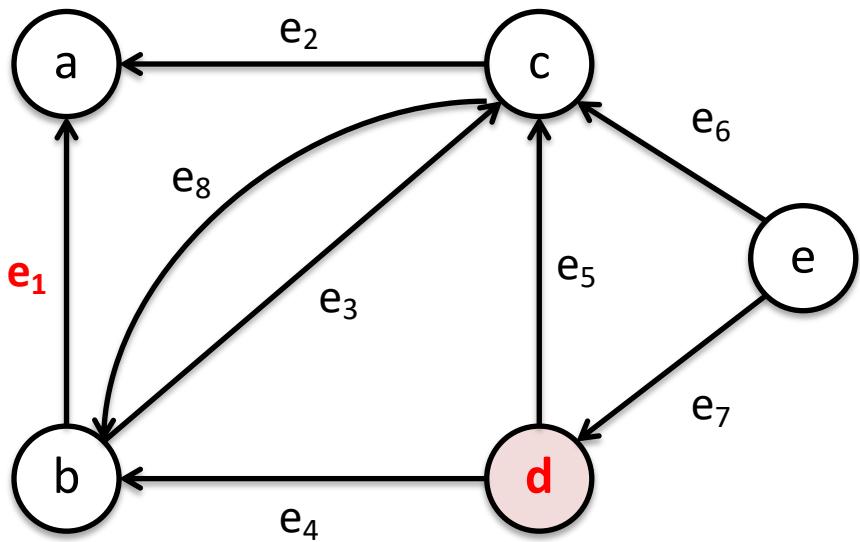
ListaAdj /



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes

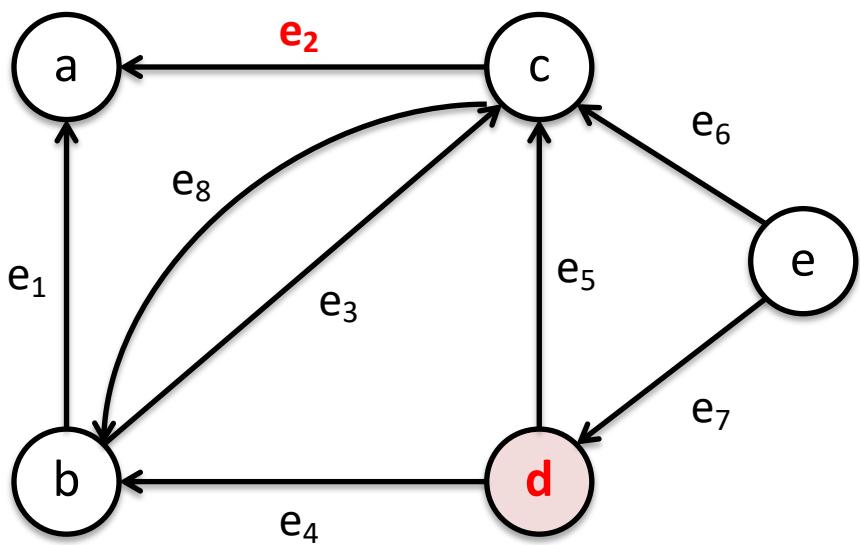
ListaAdj /



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes

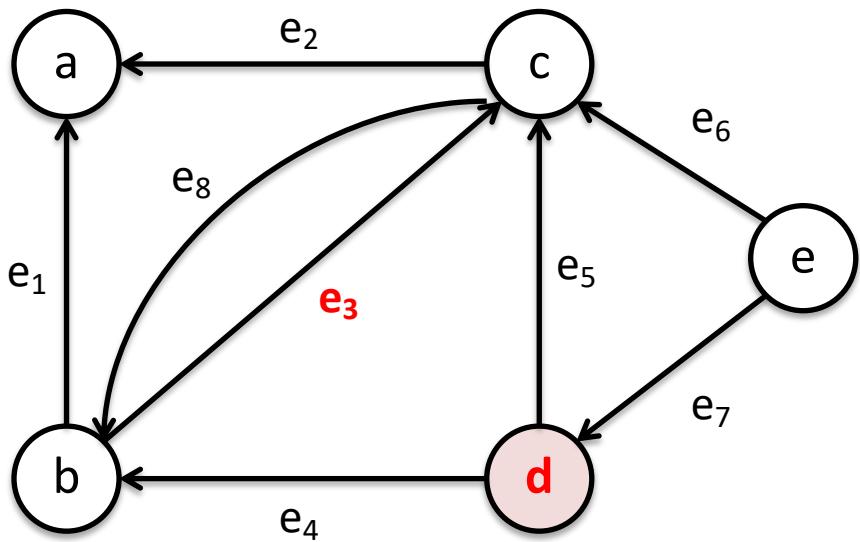
ListaAdj /



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes

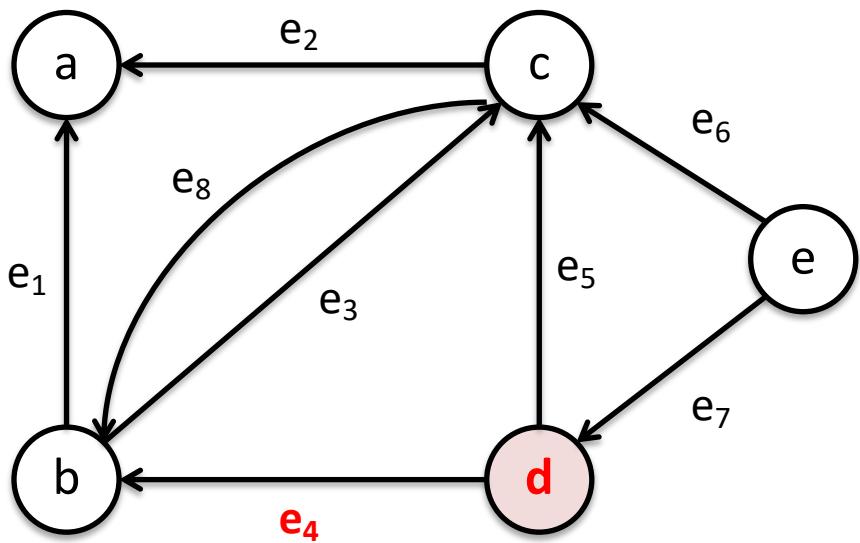
ListaAdj /



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes

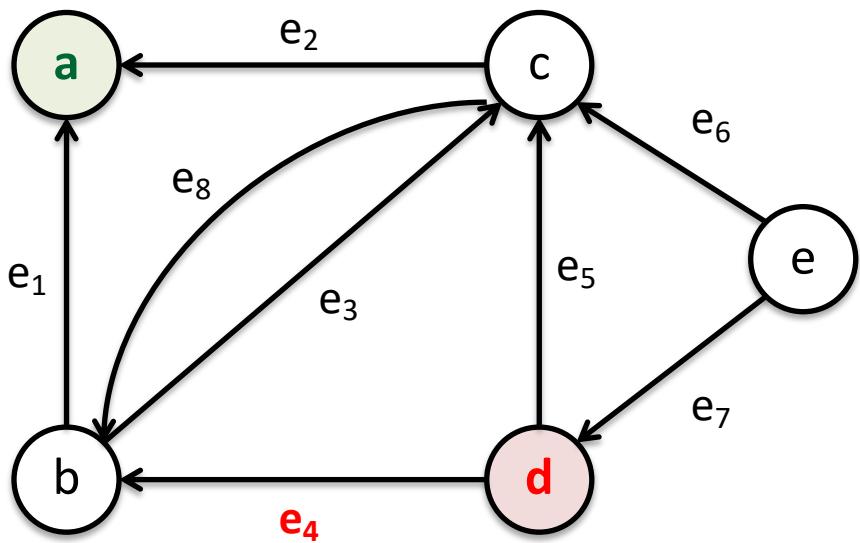
ListaAdj /



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	0	-1	0	0
e	0	0	0	0	0	1	1	0

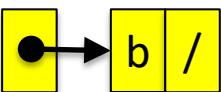
# Lista os Vértices Adjacentes

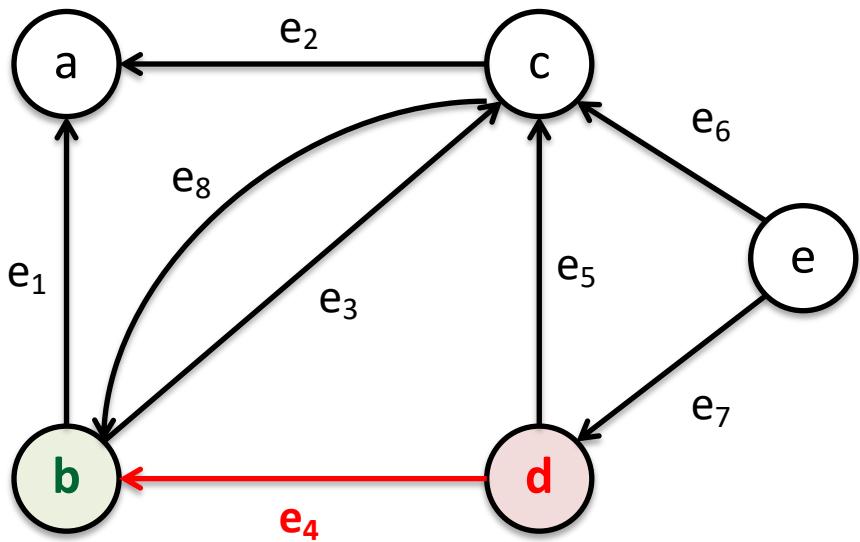
ListaAdj /



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	0	-1	0	0
e	0	0	0	0	0	1	1	0

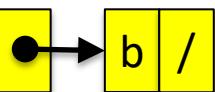
# Lista os Vértices Adjacentes

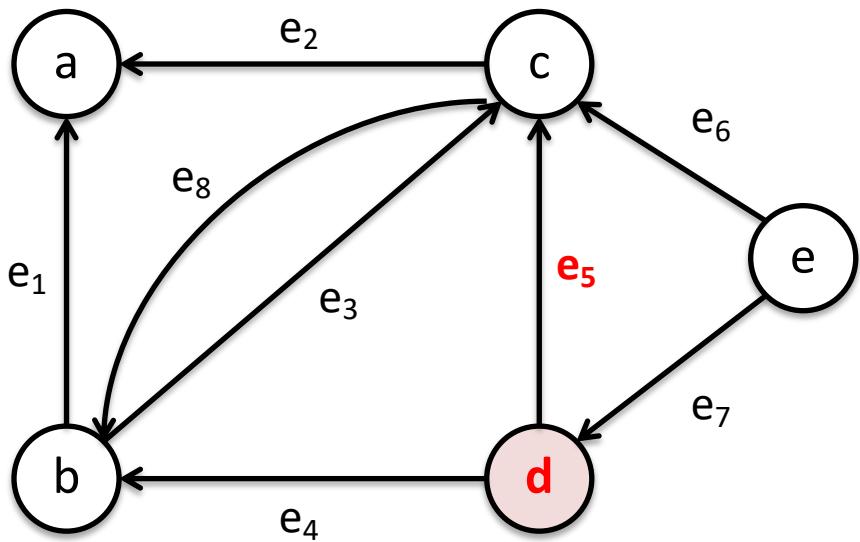
ListaAdj    



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	0	-1	0	0
e	0	0	0	0	0	1	1	0

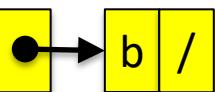
# Lista os Vértices Adjacentes

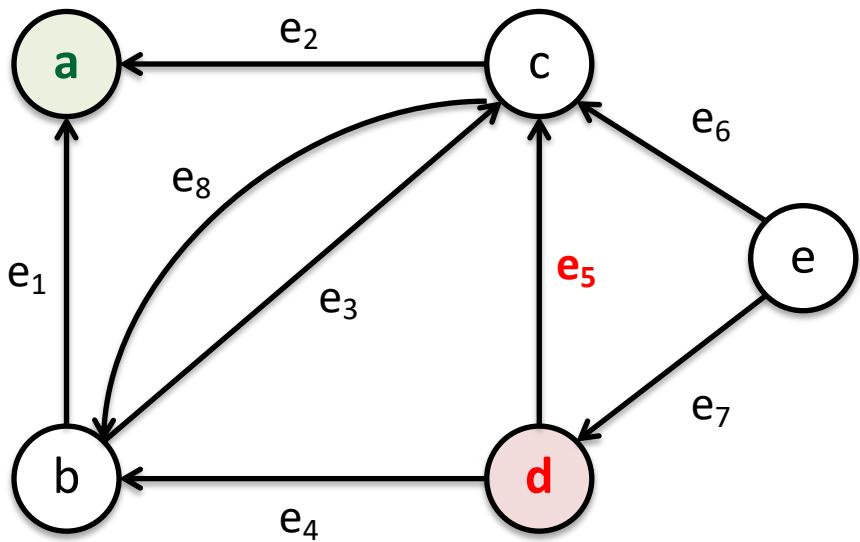
ListaAdj    



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

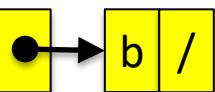
# Lista os Vértices Adjacentes

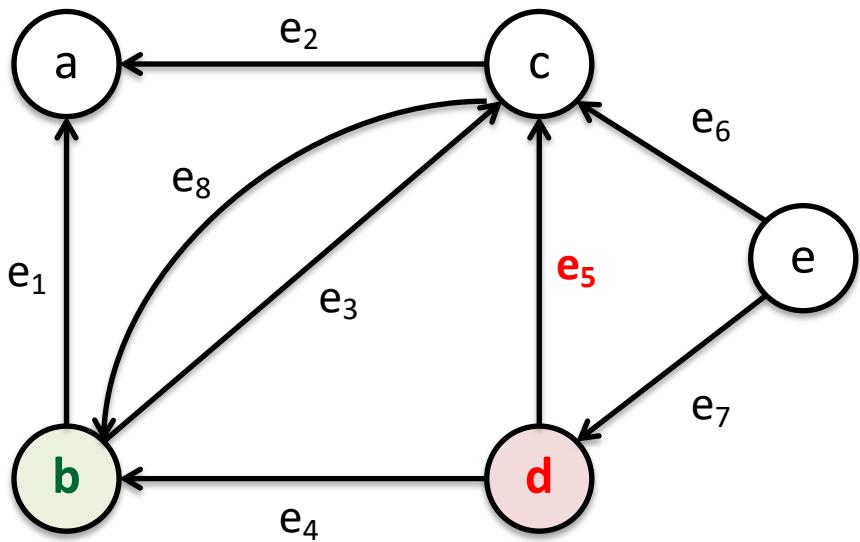
ListaAdj    



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

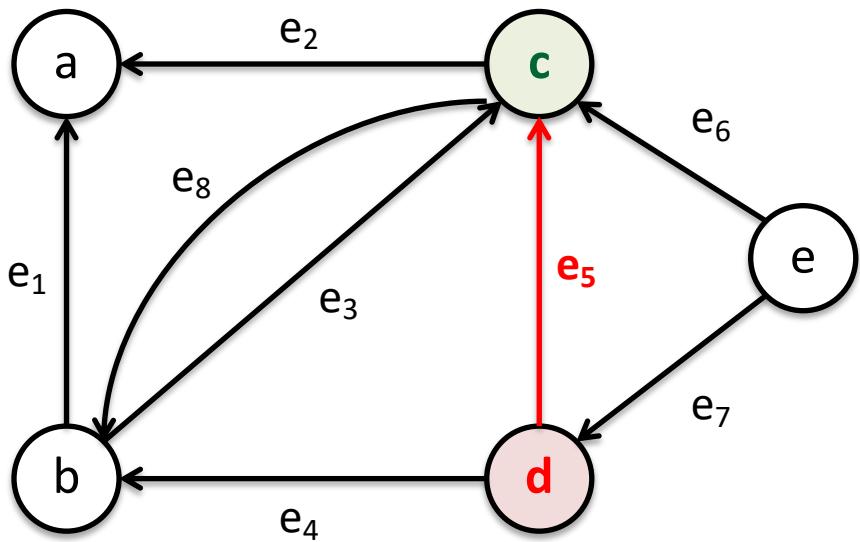
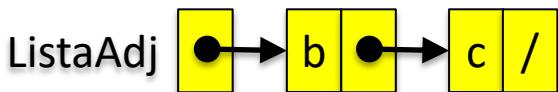
# Lista os Vértices Adjacentes

ListaAdj    



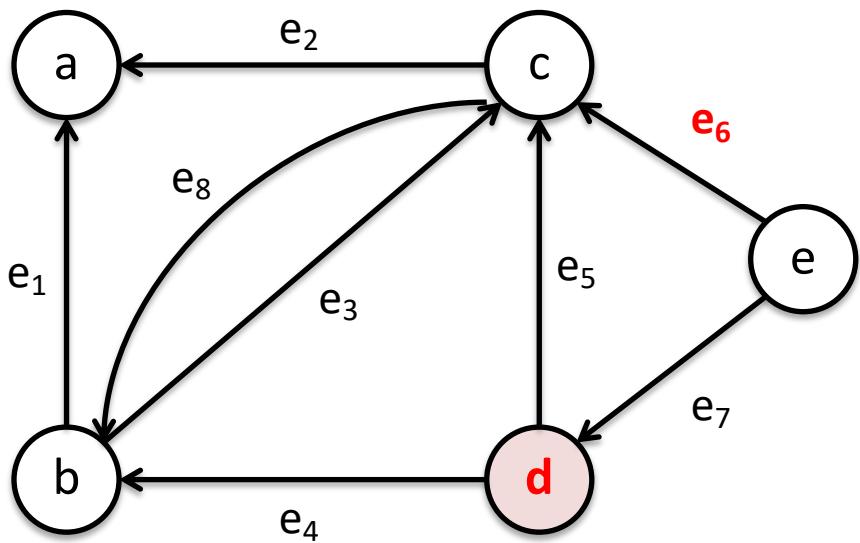
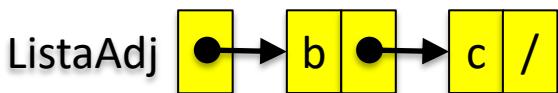
	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes



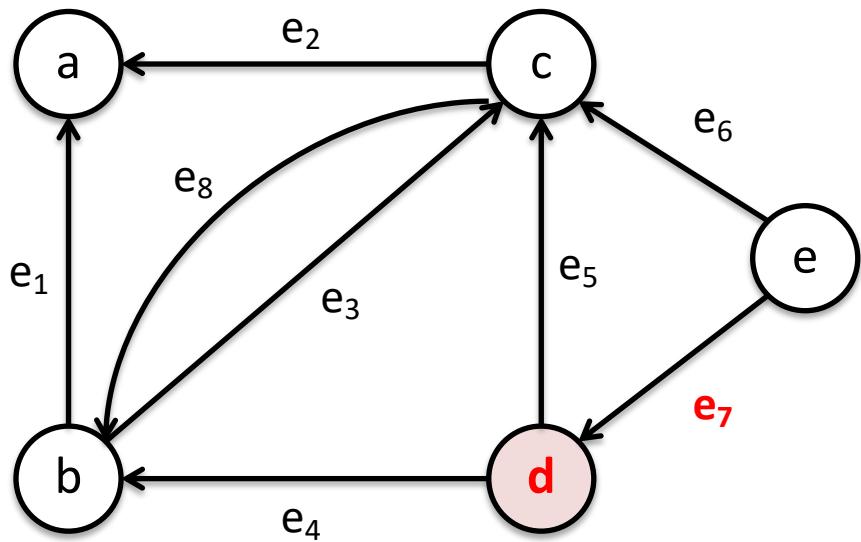
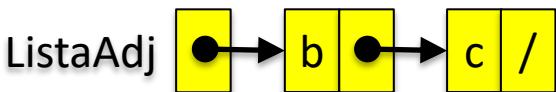
	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	$e_6$	$e_7$	$e_8$
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes



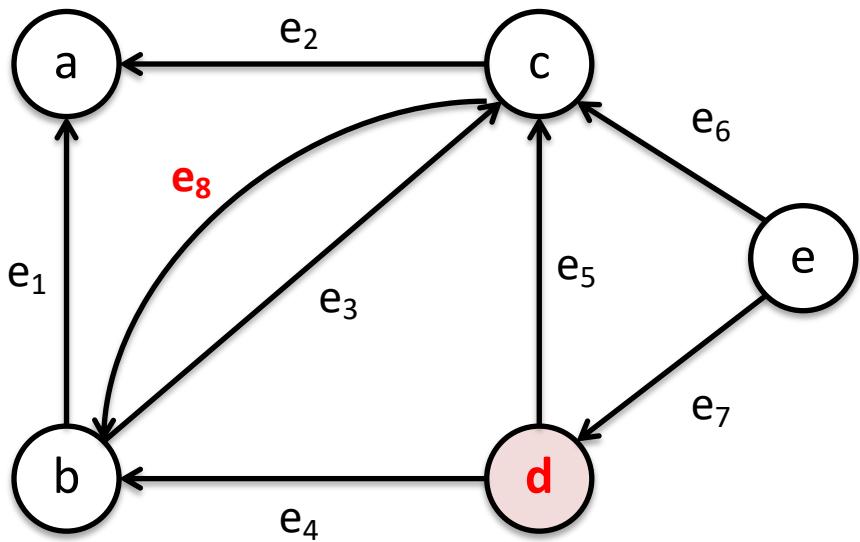
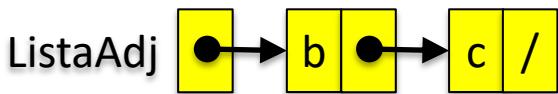
	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	$e_6$	$e_7$	$e_8$
<b>a</b>	-1	-1	0	0	0	0	0	0
<b>b</b>	1	0	1	-1	0	0	0	-1
<b>c</b>	0	1	-1	0	-1	-1	0	1
<b>d</b>	0	0	0	1	1	<b>0</b>	-1	0
<b>e</b>	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes



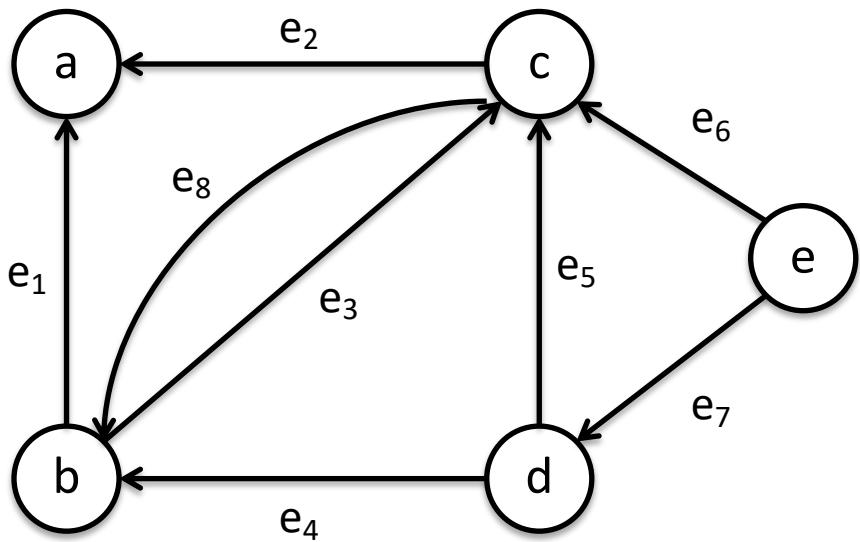
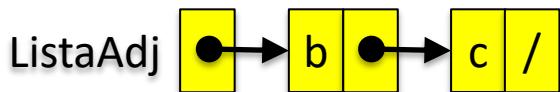
	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	$e_6$	$e_7$	$e_8$
<b>a</b>	-1	-1	0	0	0	0	0	0
<b>b</b>	1	0	1	-1	0	0	0	-1
<b>c</b>	0	1	-1	0	-1	-1	0	1
<b>d</b>	0	0	0	1	1	0	-1	0
<b>e</b>	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes



	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	$e_6$	$e_7$	$e_8$
<b>a</b>	-1	-1	0	0	0	0	0	0
<b>b</b>	1	0	1	-1	0	0	0	-1
<b>c</b>	0	1	-1	0	-1	-1	0	1
<b>d</b>	0	0	0	1	1	0	-1	0
<b>e</b>	0	0	0	0	0	1	1	0

# Lista os Vértices Adjacentes



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>	e <sub>7</sub>	e <sub>8</sub>
a	-1	-1	0	0	0	0	0	0
b	1	0	1	-1	0	0	0	-1
c	0	1	-1	0	-1	-1	0	1
d	0	0	0	1	1	0	-1	0
e	0	0	0	0	0	1	1	0

# Operações na Matriz de Incidência

```
/* Inicia as variaveis do grafo */
int TGrafo_Inicia(TGrafo *pGrafo, int NVertices)
{
    TVertice u, v;

    if (NVertices > MAXVERTICES)
        return 0;

    pGrafo->NVertices = NVertices;
    pGrafo->NArestas = 0;

    return 1;
}

/* Retorna se existe a aresta (u, v) no grafo */
int TGrafo_ExisteAresta(TGrafo *pGrafo, TVertice u, TVertice v)
{
    TAresta e;

    for (e = 0; e < pGrafo->NArestas; e++)
        if ((pGrafo->Adj[u][e].IncideAresta > 0) && (pGrafo->Adj[v][e].IncideAresta < 0))
            return 1;

    return 0;
}
```

# Operações na Matriz de Incidência

```
/* Insere a aresta e incidente aos vertices u e v no grafo */
int TGrafo_InsereAresta(TGrafo *pGrafo, TVertice u, TVertice v, TAresta e)
{
    TVertice i;

    if (pGrafo->NArestas >= MAXARESTAS)
        return 0;

    for (i = 0; i < pGrafo->NVertices; i++)
        pGrafo->Adj[i][pGrafo->NArestas].IncideAresta = 0;

    pGrafo->Adj[u][pGrafo->NArestas].IncideAresta = 1;
    pGrafo->Adj[v][pGrafo->NArestas].IncideAresta = -1;
    pGrafo->Adj[u][pGrafo->NArestas].Aresta = e;
    pGrafo->Adj[v][pGrafo->NArestas].Aresta = e;
    pGrafo->NArestas++;

    return 1;
}
```

# Operações na Matriz de Incidência

```
/* Retira a aresta e incidente aos vertices u e v no grafo */
int TGrafo_RetiraAresta(TGrafo *pGrafo, TVertice u, TVertice v, TAresta *pE)
{
    TVertice i;
    TAresta e;

    if (! TGrafo_ExisteAresta(pGrafo, u, v))
        return 0;

    for (e = 0; e < pGrafo->NAreastas; e++)
        if ((pGrafo->Adj[u][e].IncideAresta > 0) &&
            (pGrafo->Adj[v][e].IncideAresta < 0))
            break;

    *pE = pGrafo->Adj[u][e].Aresta;
    for (i = 0; i < pGrafo->NVertices; i++)
        pGrafo->Adj[i][e] = pGrafo->Adj[i][pGrafo->NAreastas-1];
    pGrafo->NAreastas--;

    return 1;
}
```

# Operações na Matriz de Incidência

```
/* Retorna a lista de adjacentes do vertice u no grafo */
TLista *TGrafo_ListaAdj(TGrafo *pGrafo, TVertice u)
{
    TLista *pLista;
    TVertice v;
    TAresta e;

    pLista = (TLista *) malloc(sizeof(TLista));
    TLista_Inicia(pLista);

    for (e = 0; e < pGrafo->NArestas; e++)
        if (pGrafo->Adj[u][e].IncideAresta > 0)
            for (v = 0; v < pGrafo->NVertices; v++)
                if (pGrafo->Adj[v][e].IncideAresta < 0)
                    { TLista_Insere(pLista, TLista_Tamanho(pLista), v); break; }

    return pLista;
}

/* Retorna o numero de vertices do grafo */
int TGrafo_NVertices(TGrafo *pGrafo)
{
    return (pGrafo->NVertices);
}

/* Retorna o numero de arestas do grafo */
int TGrafo_NArestas(TGrafo *pGrafo)
{
    return (pGrafo->NArestas);
}
```

## ■ Vantagens:

- O tempo necessário para descobrir se dois vértices são adjacentes é independente de  $|V|$  ou  $|E|$
- É muito útil para algoritmos em que é necessário saber com rapidez se existe uma aresta incidente a dois vértices

## ■ Desvantagens:

- O tempo necessário para acessar a lista de adjacentes de um vértice depende de  $|V|$  ou  $|E|$
- Requer uma quantidade de espaço da ordem  $O(|V|^2)$  independente do tamanho do grafo

## ■ Vantagens:

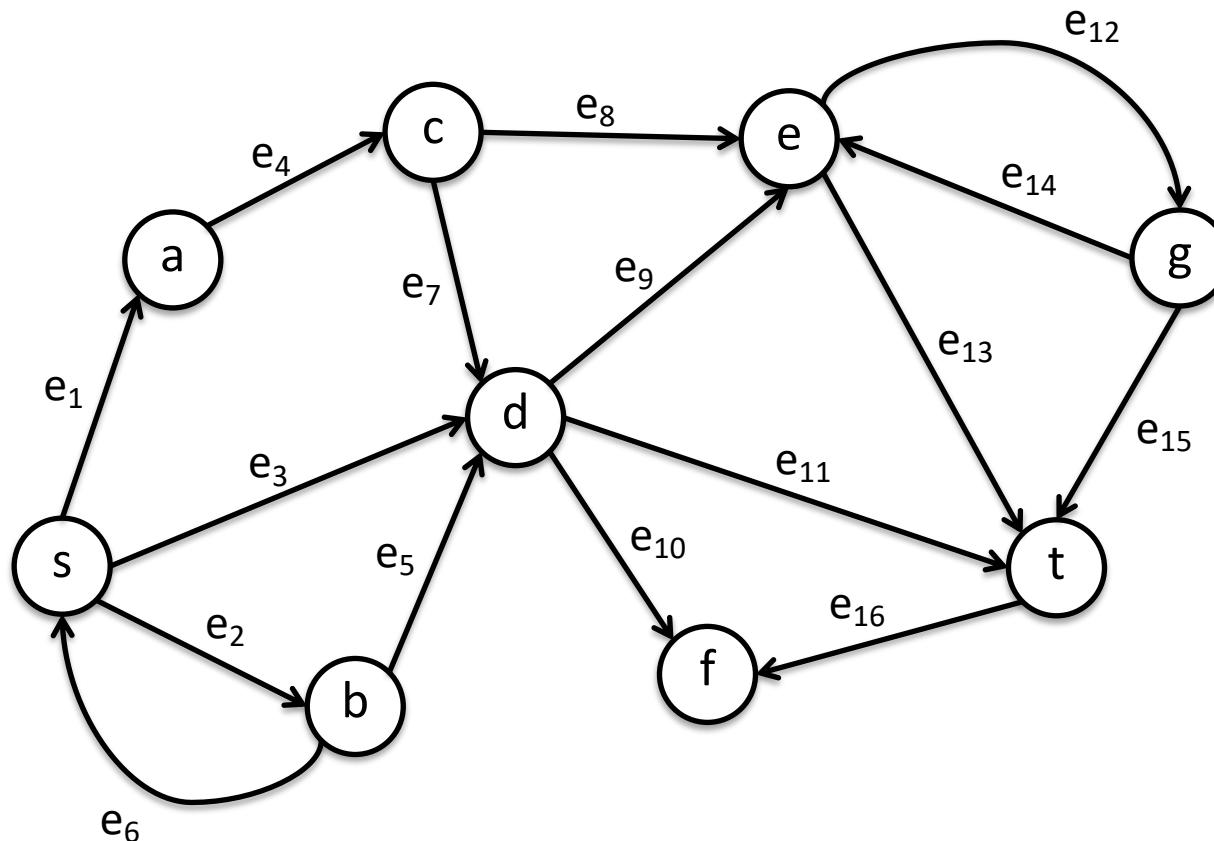
- O tempo necessário para descobrir se dois vértices são adjacentes é independente de  $|V|$  ou  $|E|$
- É muito útil para algoritmos em que é necessário saber com rapidez se existe uma aresta incidente a dois vértices

## ■ Desvantagens:

- O tempo necessário para acessar a lista de adjacentes de um vértice depende de  $|V|$  ou  $|E|$
- Requer uma quantidade de espaço da ordem  $O(|V|^2)$  independente do tamanho do grafo

Recomendada para grafos **densos**, em que  $|E| \approx |V|^2$

- Represente o grafo abaixo por meio de matriz de adjacência e matriz de incidência.

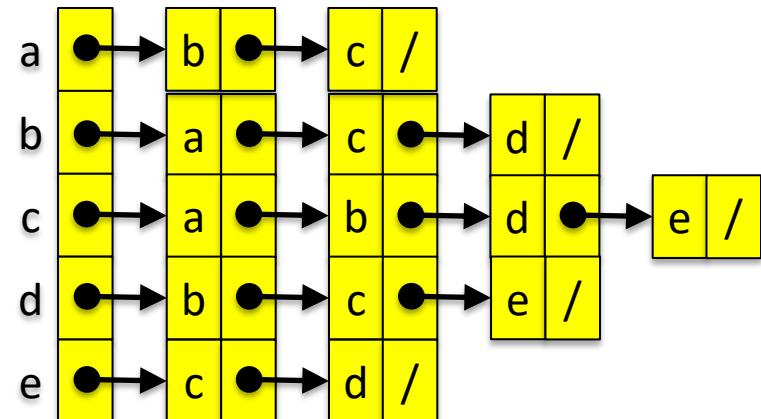
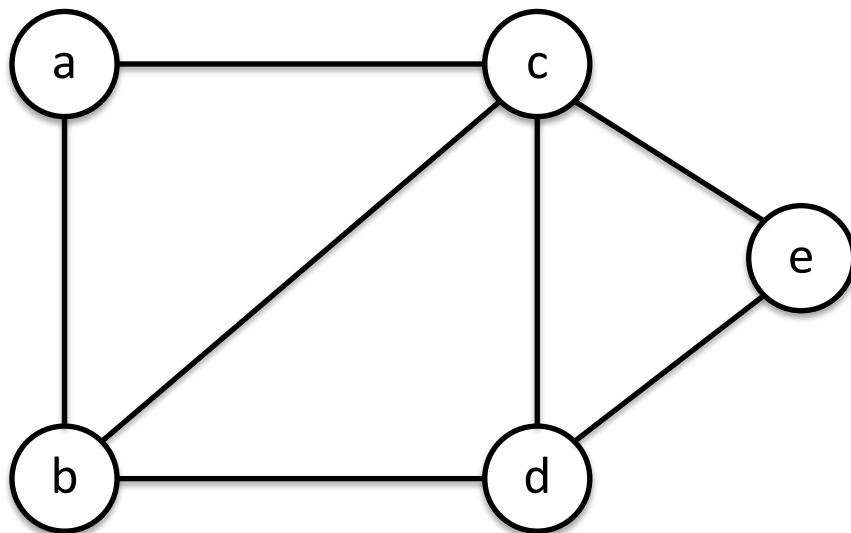


# IMPLEMENTAÇÃO POR LISTAS LINEARES

- Seja  $G=(V, E)$  um grafo simples (orientado ou não)
- Para cada vértice  $u \in V$ , tem-se uma lista linear  $\text{Adj}[u]$  de seus vértices adjacentes, ou seja, o vértice  $v \in V$  aparece em  $\text{Adj}[u]$  se, e somente se,  $(u, v) \in E$ 
  - Os vértices podem estar em qualquer ordem na lista
- Essa estrutura é chamada de **lista de adjacência**

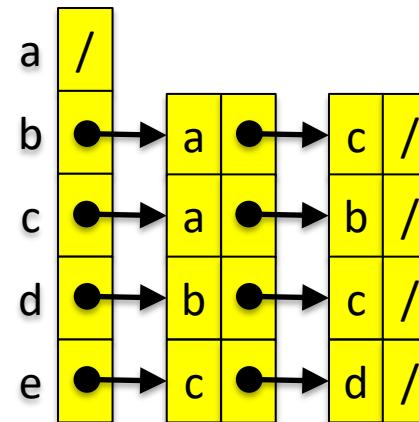
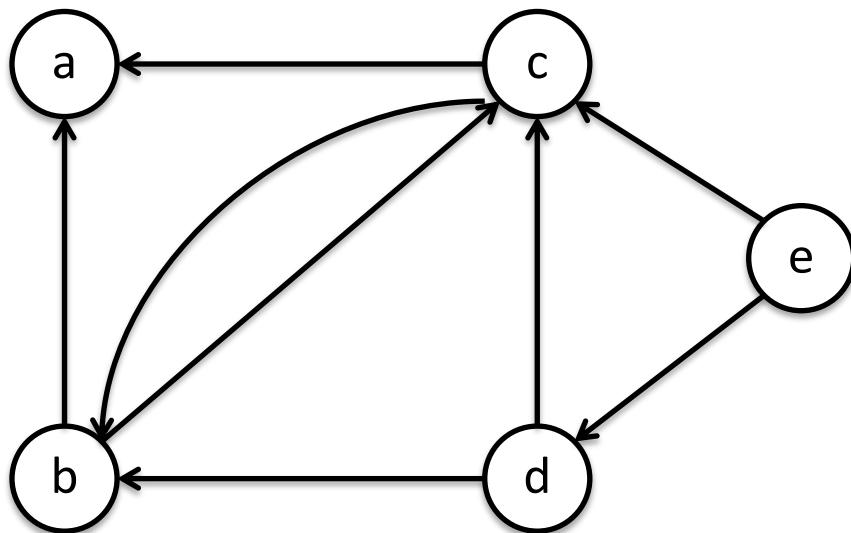
# Implementação por Listas

- Exemplo de um grafo **não orientado** e a sua lista de adjacência correspondente



# Implementação por Listas

- Exemplo de um grafo **orientado** e a sua lista de adjacência correspondente



# Estrutura da Lista de Adjacência

```
#define MAXVERTICES 100

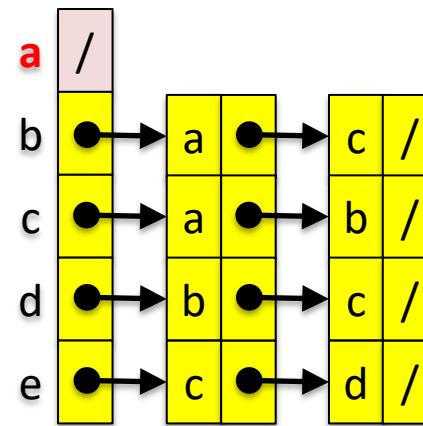
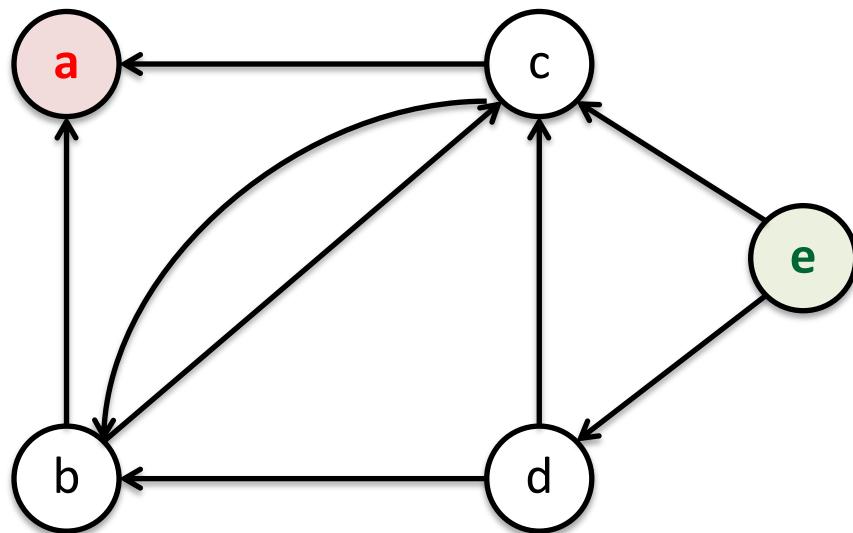
typedef int TVertice;

typedef int TAresta;

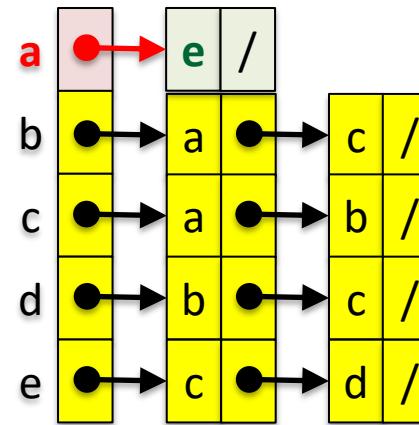
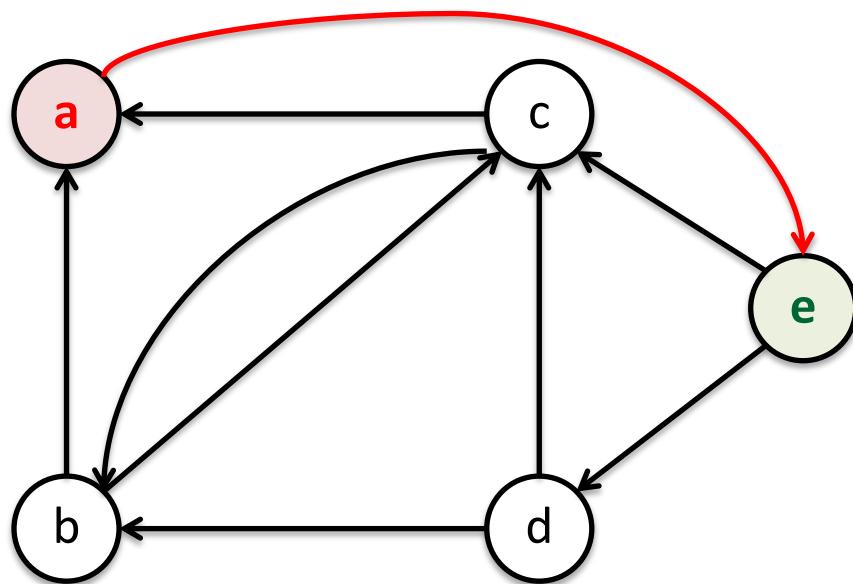
typedef struct {
    TVertice Vertice;
    TAresta Aresta;
} TAdjacencia;

typedef struct {
    TLista Adj [MAXVERTICES];
    int NVertices;
    int NAreastas;
} TGrafo;
```

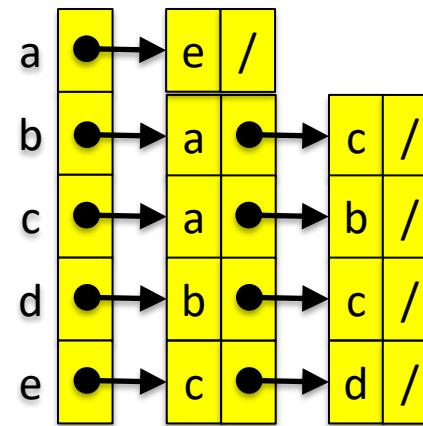
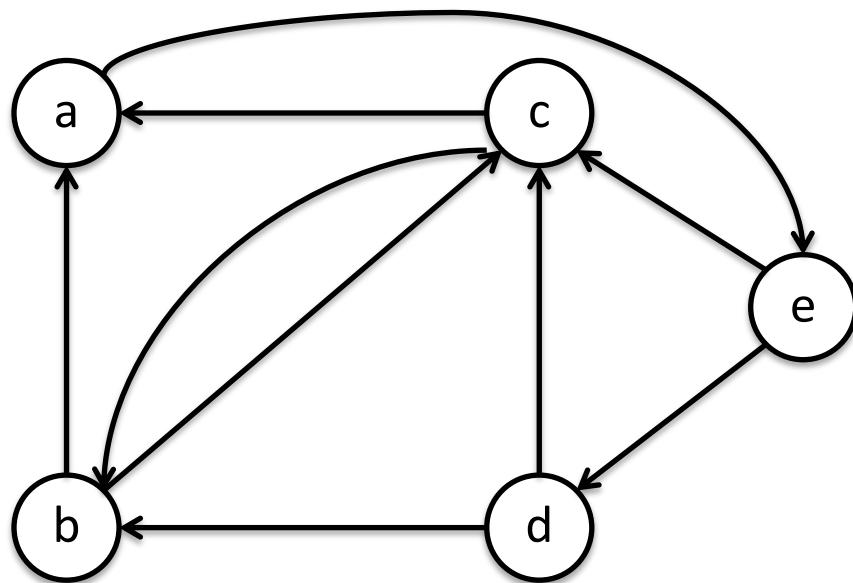
# Inserção de Arestas no Grafo



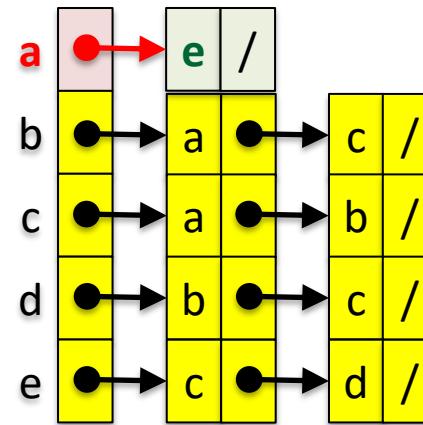
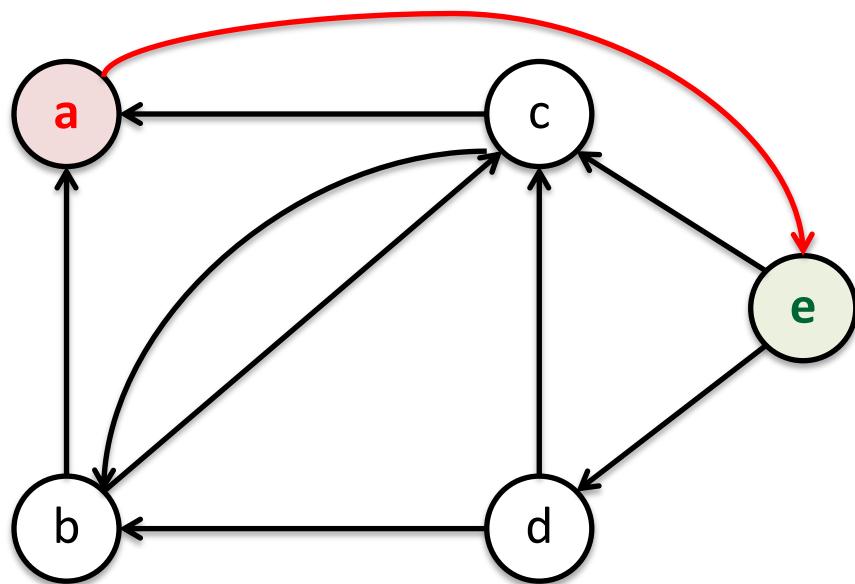
# Inserção de Arestas no Grafo



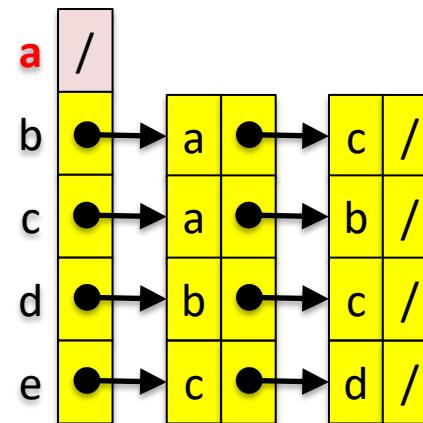
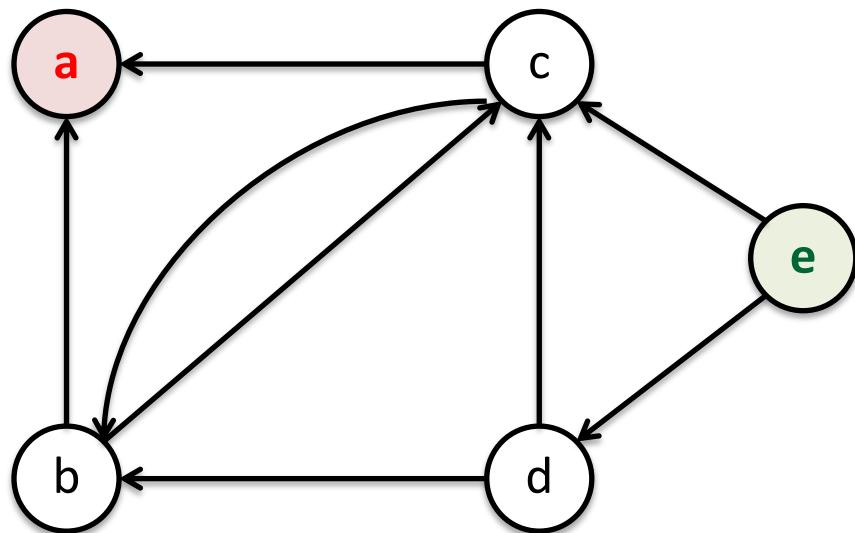
# Inserção de Arestas no Grafo



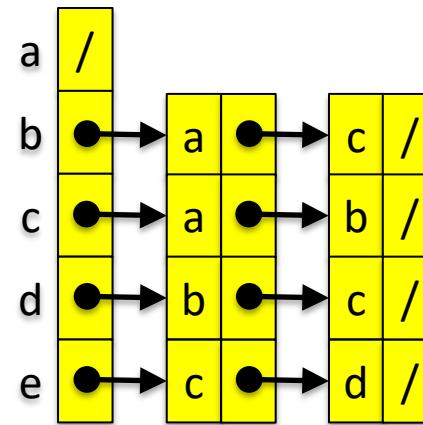
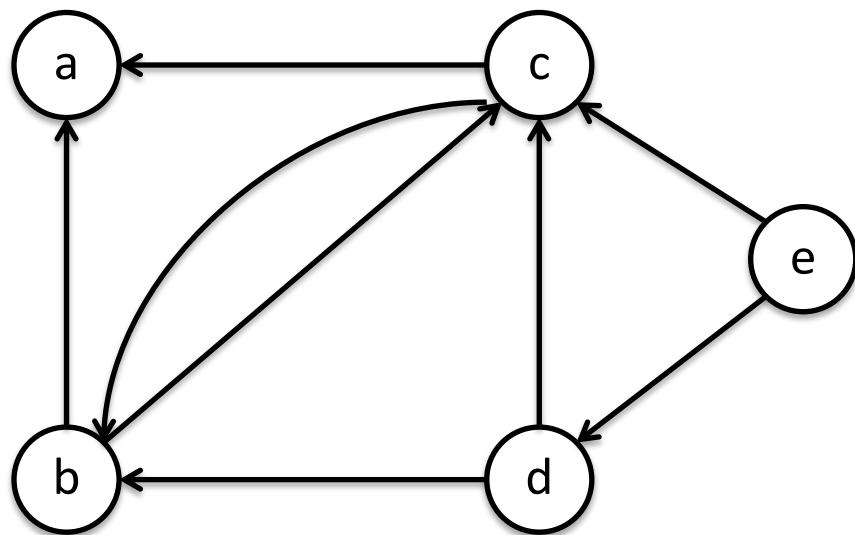
# Retirada de Arestas do Grafo



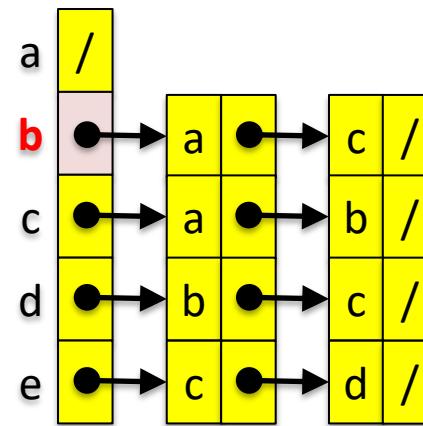
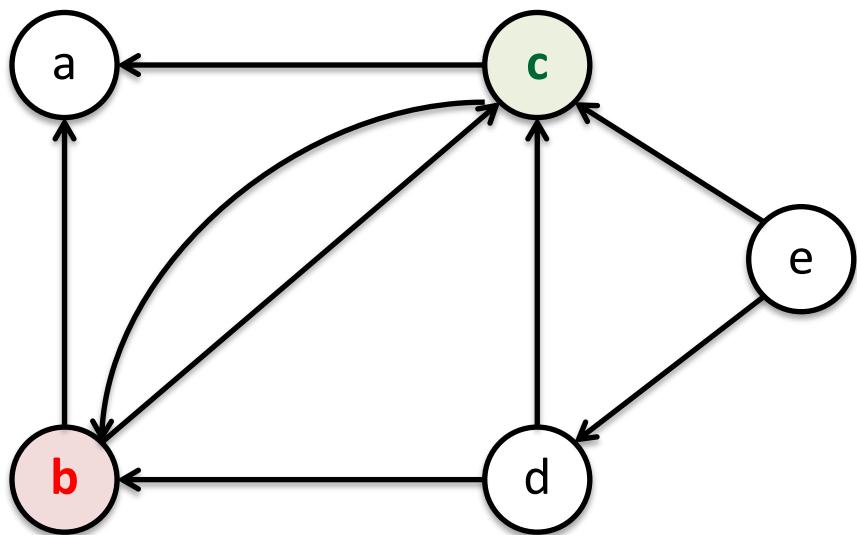
# Retirada de Arestas do Grafo



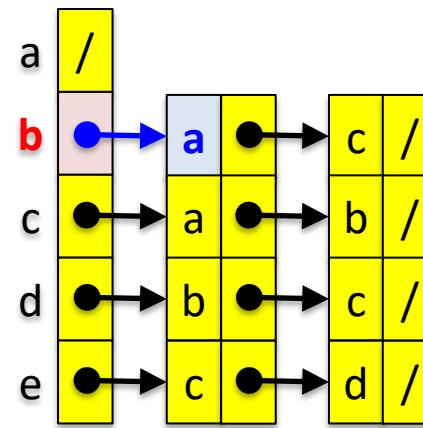
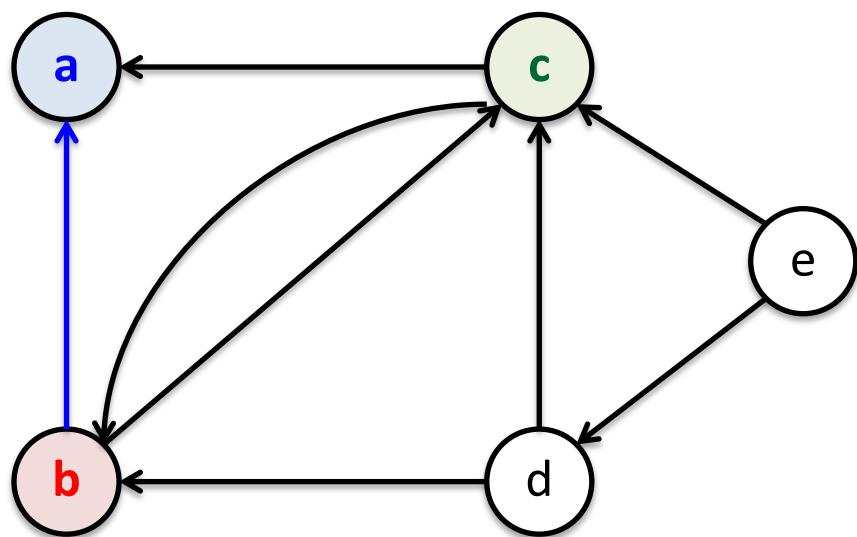
# Retirada de Arestas do Grafo



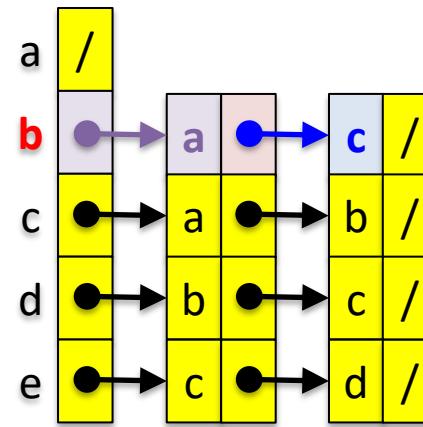
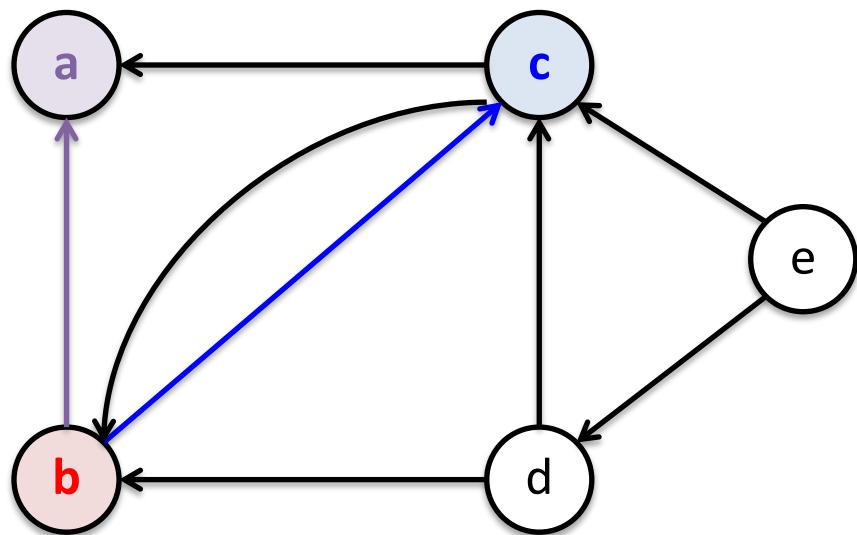
# Verifica Existência de Aresta



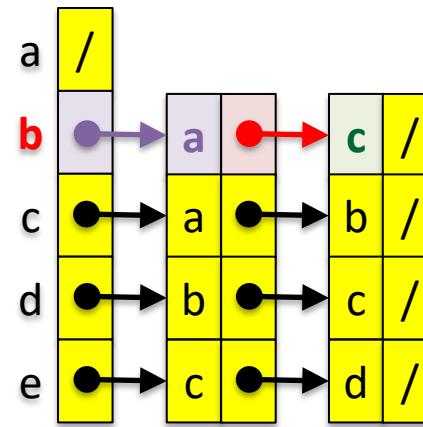
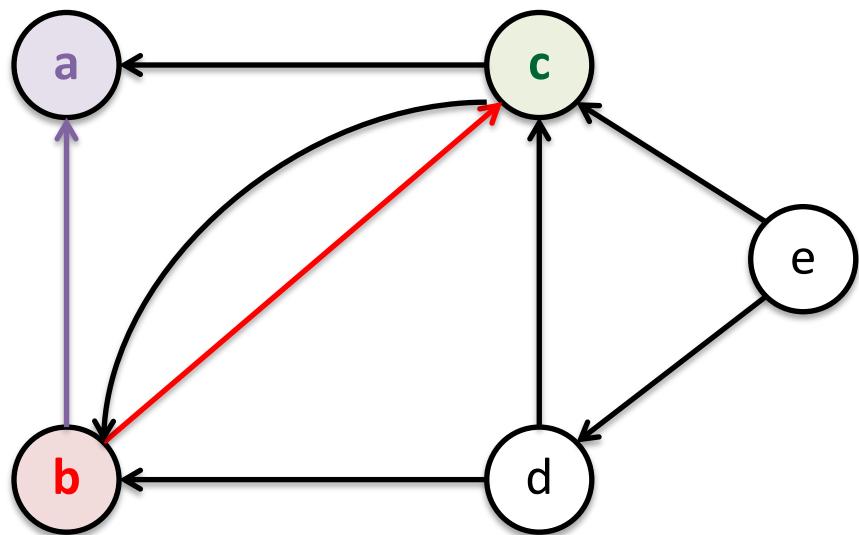
# Verifica Existência de Aresta



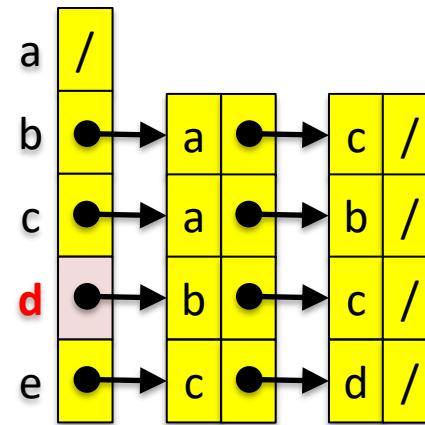
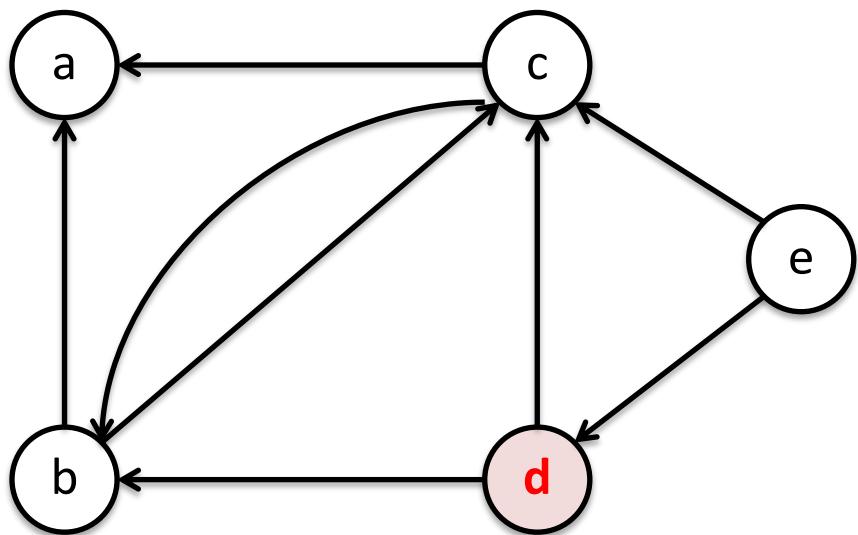
# Verifica Existência de Aresta



# Verifica Existência de Aresta

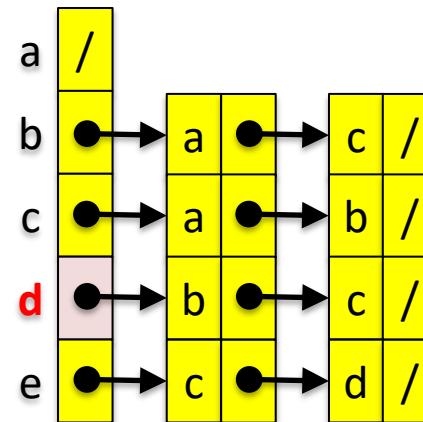
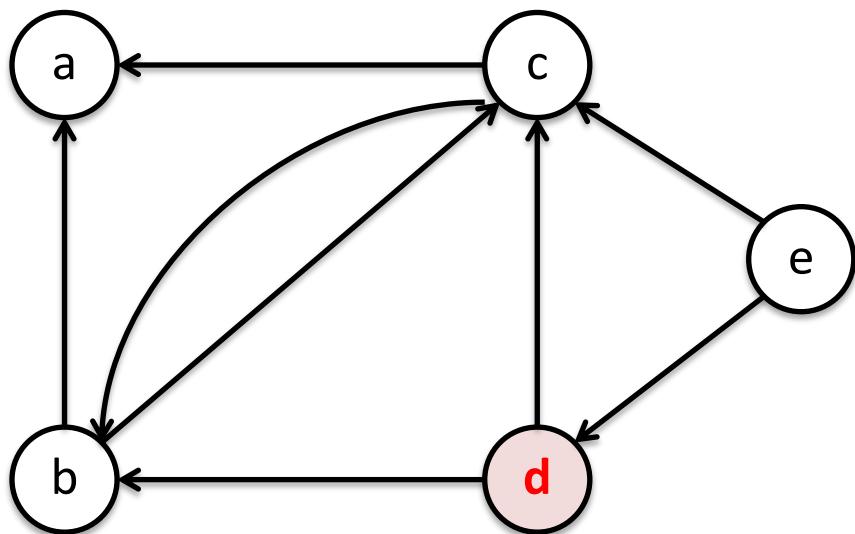


# Lista os Vértices Adjacentes

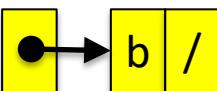


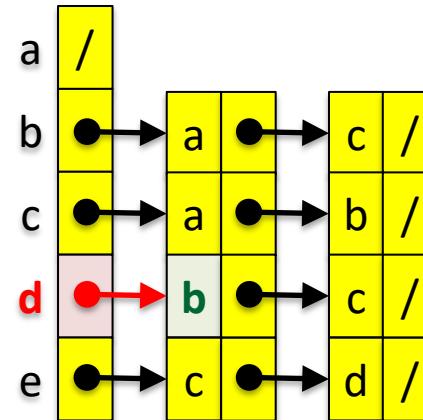
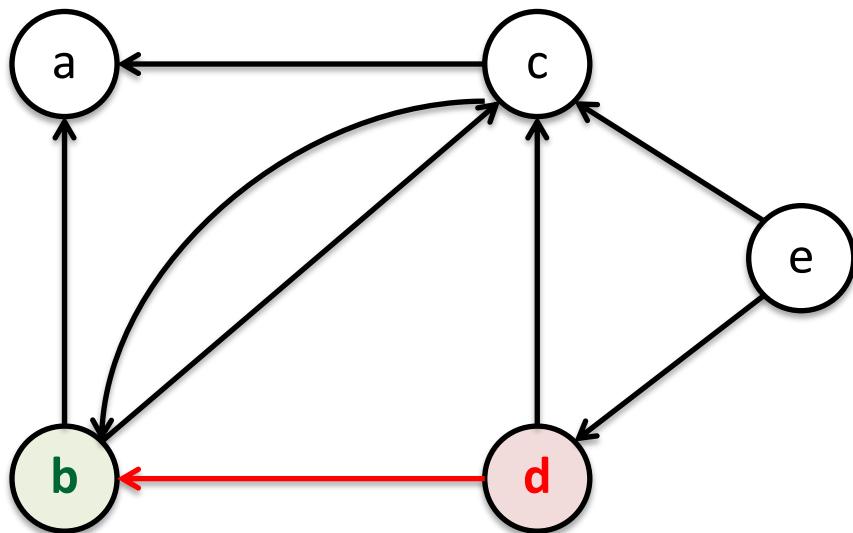
# Lista os Vértices Adjacentes

ListaAdj /

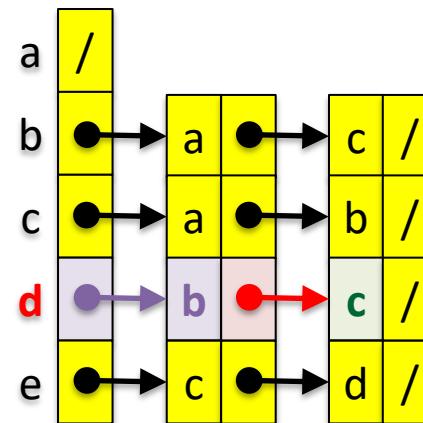
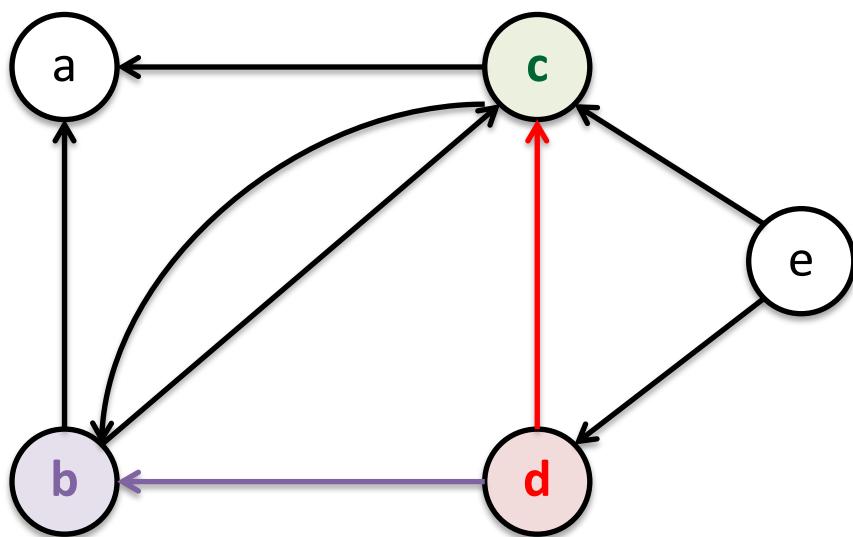
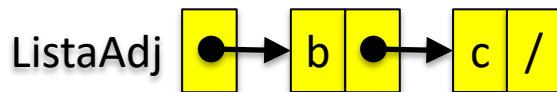


# Lista os Vértices Adjacentes

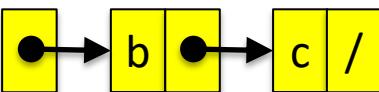
ListaAdj    

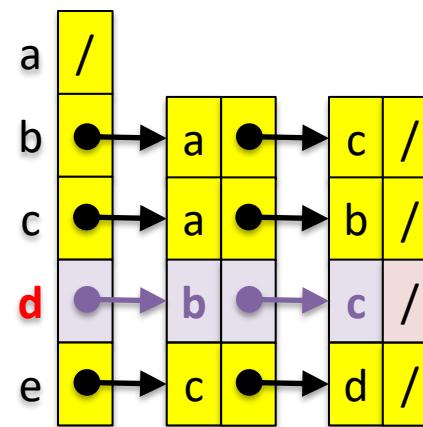
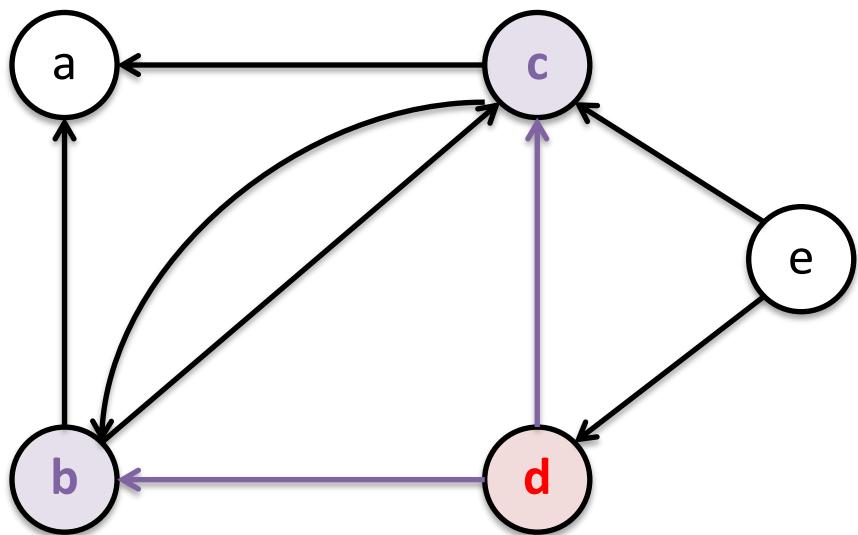


# Lista os Vértices Adjacentes

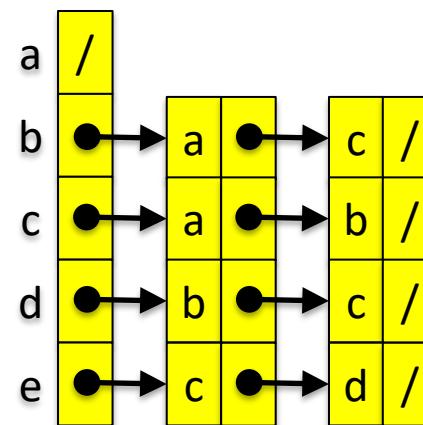
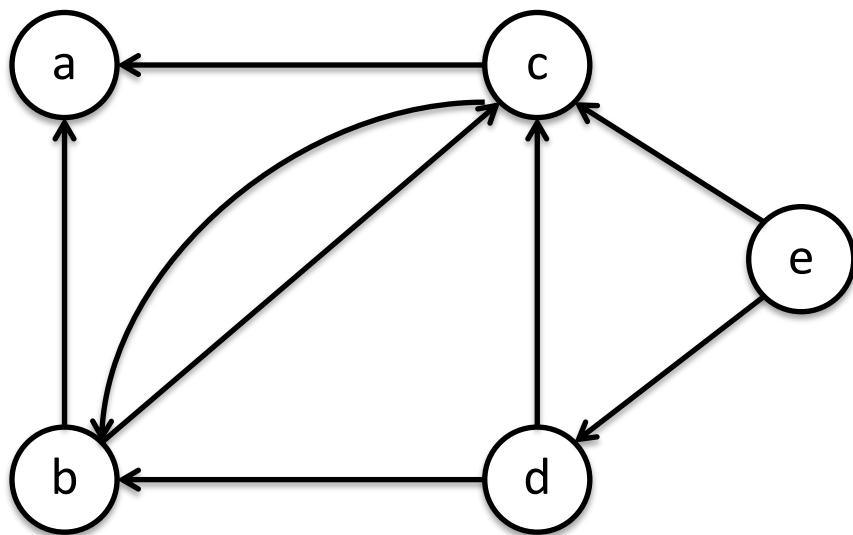
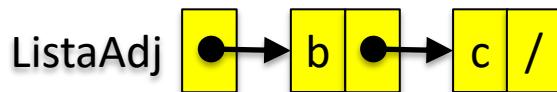


# Lista os Vértices Adjacentes

ListaAdj    



# Lista os Vértices Adjacentes



# Operações na Lista de Adjacência

```
/* Inicia as variaveis do grafo */
int TGrafo_Inicia(TGrafo *pGrafo, int NVertices)
{
    TVertice u;

    if (NVertices > MAXVERTICES)
        return 0;

    pGrafo->NVertices = NVertices;
    pGrafo->NAreastas = 0;
    for (u = 0; u < pGrafo->NVertices; u++)
        TLista_Inicia(&pGrafo->Adj[u]);

    return 1;
}
```

# Operações na Lista de Adjacência

```
/* Retorna se existe a aresta (u, v) no grafo */
int TGrafo_ExisteAresta(TGrafo *pGrafo, TVertice u, TVertice v)
{
    TAdjacencia Adj;
    TLista ListaAdj;
    int Existe;

    Existe = 0;
    TLista_Inicia(&ListaAdj);
    while (TLista_Retira(&pGrafo->Adj[u], TLista_Tamanho(&pGrafo->Adj[u])-1, &Adj)) {
        TLista_Insere(&ListaAdj, TLista_Tamanho(&ListaAdj), Adj);
        Existe = (Adj.Vertice == v);
        if (Existe)
            break;
    }

    while (TLista_Retira(&ListaAdj, TLista_Tamanho(&ListaAdj)-1, &Adj))
        TLista_Insere(&pGrafo->Adj[u], TLista_Tamanho(&pGrafo->Adj[u]), Adj);

    return (Existe);
}
```

# Operações na Lista de Adjacência

```
/* Insere a aresta e incidente aos vertices u e v no grafo */
int TGrafo_InsereAresta(TGrafo *pGrafo, TVertice u, TVertice v, Taresta e)
{
    TAdjacencia Adj;

    Adj.Vertice = v;
    Adj.Aresta = e;
    if (TLista_Insere(&pGrafo->Adj[u], TLista_Tamanho(&pGrafo->Adj[u]), Adj)) {
        pGrafo->NAreastas++;
        return 1;
    }
    else
        return 0;
}
```

# Operações na Lista de Adjacência

```
/* Retira a aresta e incidente aos vertices u e v no grafo */
int TGrafo_RetiraAresta(TGrafo *pGrafo, TVertice u, TVertice v, TAresta *pE)
{
    TLista ListaAdj;
    TAdjacencia Adj;
    int IncideAresta;

    IncideAresta = 0;
    TLista_Inicia(&ListaAdj);
    while (!IncideAresta &&
           TLista_Retira(&pGrafo->Adj[u], TLista_Tamanho(&pGrafo->Adj[u])-1, &Adj))
        if (Adj.Vertice == v) {
            *pE = Adj.Aresta;
            pGrafo->NAreastas--;
            IncideAresta = 1;
        }
    else
        TLista_Insere(&ListaAdj, TLista_Tamanho(&ListaAdj), Adj);

    while (TLista_Retira(&ListaAdj, TLista_Tamanho(&ListaAdj)-1, &Adj))
        TLista_Insere(&pGrafo->Adj[u], TLista_Tamanho(&pGrafo->Adj[u]), Adj);

    return IncideAresta;
}
```

# Operações na Lista de Adjacência

```
/* Retorna a lista de adjacentes do vertice u no grafo */
TLista *TGrafo_ListaAdj (TGrafo *pGrafo, TVertice u)
{
    TLista *pLista, ListaAdj;
    TAdjacencia Adj;

    pLista = (TLista *) malloc(sizeof(TLista));
    TLista_Inicia(pLista);

    TLista_Inicia(&ListaAdj);
    while (TLista_Retira(&pGrafo->Adj[u], TLista_Tamanho(&pGrafo->Adj[u])-1, &Adj))
        TLista_Insere(&ListaAdj, TLista_Tamanho(&ListaAdj), Adj);

    while (TLista_Retira(&ListaAdj, TLista_Tamanho(&ListaAdj)-1, &Adj)) {
        TLista_Insere(&pGrafo->Adj[u], TLista_Tamanho(&pGrafo->Adj[u]), Adj);
        TLista_Insere(pLista, TLista_Tamanho(pLista), Adj);
    }

    return pLista;
}
```

# Operações na Lista de Adjacência

```
/* Retorna o numero de vertices do grafo */
int TGrafo_NVertices(TGrafo *pGrafo)
{
    return (pGrafo->NVertices);
}

/* Retorna o numero de arestas do grafo */
int TGrafo_NArestas(TGrafo *pGrafo)
{
    return (pGrafo->NArestas);
}
```

## ■ Vantagens:

- O tempo necessário para acessar a lista de adjacentes de um vértice é independente de  $|V|$  ou  $|E|$
- É muito útil para algoritmos em que é necessário saber com rapidez os vértices adjacentes a um dado vértice
- Requer uma quantidade de espaço da ordem  $O(|V| + |E|)$  ou seja, exatamente igual ao tamanho do grafo

## ■ Desvantagens:

- O tempo necessário para descobrir se dois vértices são adjacentes depende de  $|V|$  ou  $|E|$

## ■ Vantagens:

- O tempo necessário para acessar a lista de adjacentes de um vértice é independente de  $|V|$  ou  $|E|$
- É muito útil para algoritmos em que é necessário obter com frequência (rapidez) os vértices adjacentes a um dado vértice
- Requer uma quantidade de espaço da ordem  $O(|V| + |E|)$  ou seja, exatamente igual ao tamanho do grafo

## ■ Desvantagens:

- O tempo necessário para descobrir se dois vértices são adjacentes depende de  $|V|$  ou  $|E|$

Recomendada para grafos **esparsos**, em que  $|E| \ll |V|^2$

# Comparação de Desempenho

Característica	Matriz de Adjacência	Matriz de Incidência	Lista de Adjacência
Espaço de Memória	$O( V ^2)$	$O( V  \times  E )$	$O( V  +  E )$
Inserir uma aresta	$O(1)$	$O( V  \times  E )$	$O(1)$
Retirar uma aresta	$O(1)$	$O( V  \times  E )$	$O( E )$
Verificar se existe uma dada aresta	$O(1)$	$O( E )$	$O( V )$
Obter a lista de vértices adjacentes	$O( V )$	$O( V  \times  E )$	$O(1)$

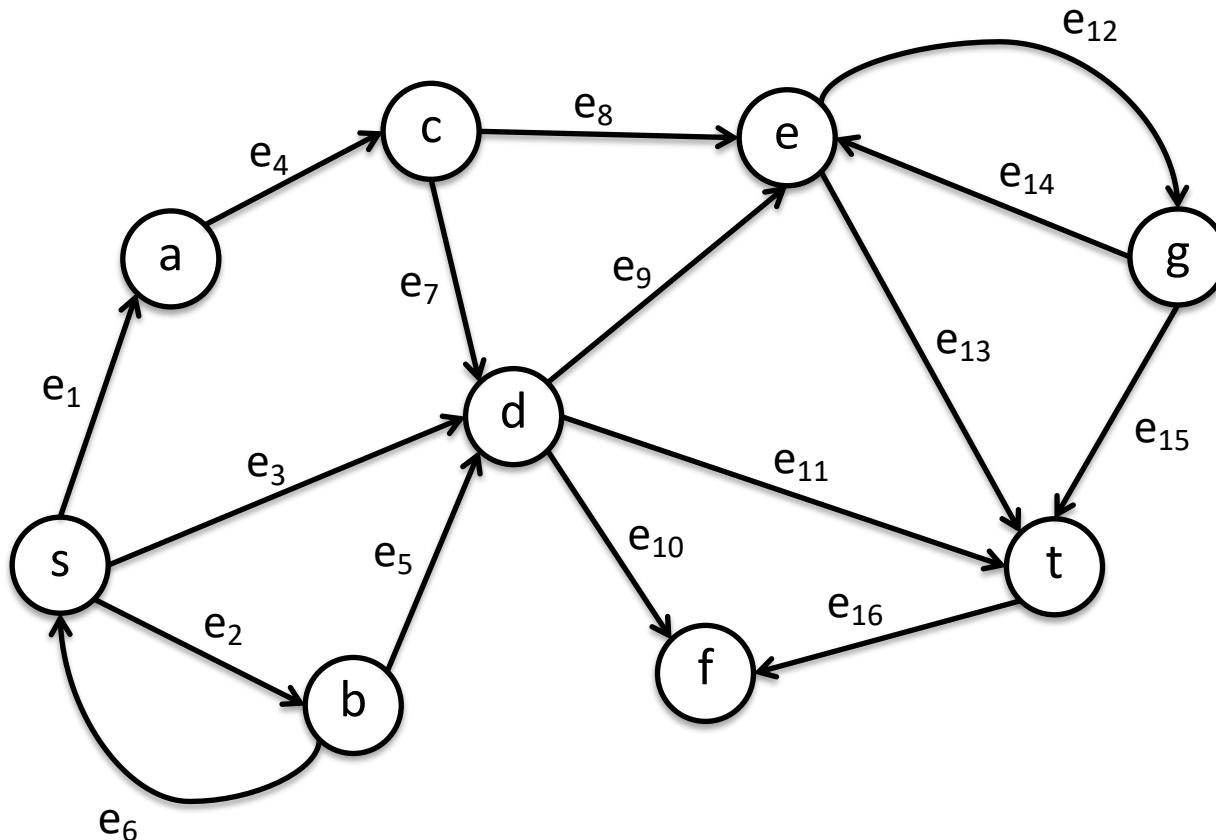
# Comparação de Desempenho

Característica	Matriz de Adjacência	Matriz de Incidência	Lista de Adjacência
Espaço de Memória	$O( V ^2)$	$O( V  \times  E )$	$O( V  +  E )$
Inserir uma aresta	$O(1)$	$O( V  \times  E )$	$O(1)$
Retirar uma aresta	$O(1)$	$O( V  \times  E )$	$O( E )$
Verificar se existe uma dada aresta	$O(1)$	$O( E )$	$O( V )$
Obter a lista de vértices adjacentes	$O( V )$	$O( V  \times  E )$	$O(1)$

Teoricamente, obter a lista de vértices adjacentes de um vértice  $v$  usando Lista de Adjacência é  $O(1)$ . Mas isso ocorre na implementação apresentada? Analise e fundamente sua resposta.

# Exercício

- Represente o grafo abaixo por meio de matriz de adjacência, matriz de incidência e lista de adjacência.



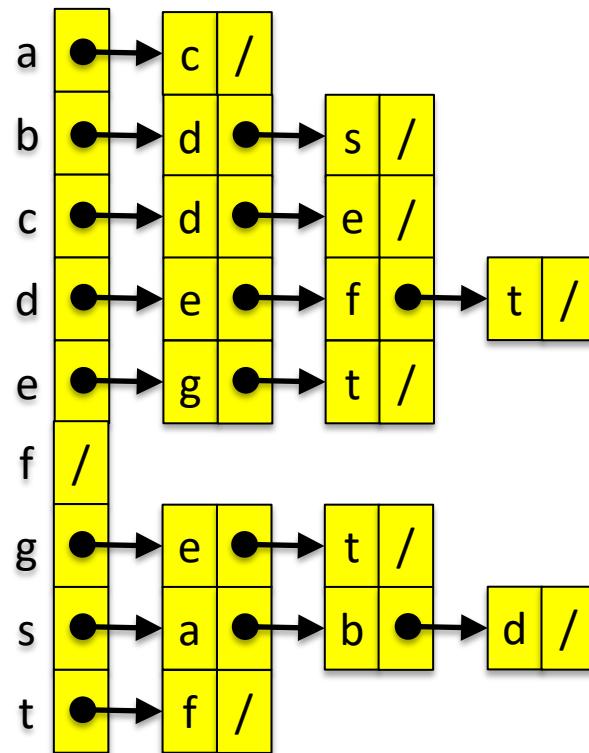
- Matriz de Adjacência

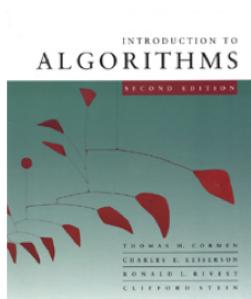
	a	b	c	d	e	f	g	s	t
a	0	0	1	0	0	0	0	0	0
b	0	0	0	1	0	0	0	1	0
c	0	0	0	1	1	0	0	0	0
d	0	0	0	0	1	1	0	0	1
e	0	0	0	0	0	0	1	0	1
f	0	0	0	0	0	0	0	0	0
g	0	0	0	0	1	0	0	0	1
s	1	1	0	1	0	0	0	0	0
t	0	0	0	0	0	1	0	0	0

- Matriz de Incidência

	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$	$e_6$	$e_7$	$e_8$	$e_9$	$e_{10}$	$e_{11}$	$e_{12}$	$e_{13}$	$e_{14}$	$e_{15}$	$e_{16}$
$a$	-1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	
$b$	0	-1	0	0	1	1	0	0	0	0	0	0	0	0	0	
$c$	0	0	0	-1	0	0	1	1	0	0	0	0	0	0	0	
$d$	0	0	-1	0	-1	0	-1	0	1	1	1	0	0	0	0	
$e$	0	0	0	0	0	0	0	-1	-1	0	0	1	1	-1	0	
$f$	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	-1	
$g$	0	0	0	0	0	0	0	0	0	0	-1	0	1	1	0	
$s$	1	1	1	0	0	-1	0	0	0	0	0	0	0	0	0	
$t$	0	0	0	0	0	0	0	0	0	0	-1	0	-1	-1	1	

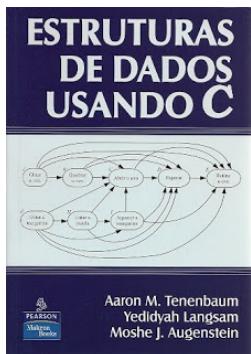
- Lista de Adjacência





CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. 3<sup>a</sup> Edição. MIT Press, 2009. **Seção 22.1**

ZIVIANI, N. **Projeto de Algoritmos com Implementações em Pascal e C**. 3<sup>a</sup> Edição. Cengage Learning, 2010. **Seções 7.1 e 7.2**



TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estruturas de Dados usando C**. Pearson Makron Books, 2008.  
**Capítulo 8**