Grafos – Introdução

Prof. Renato Pimentel

2025/2

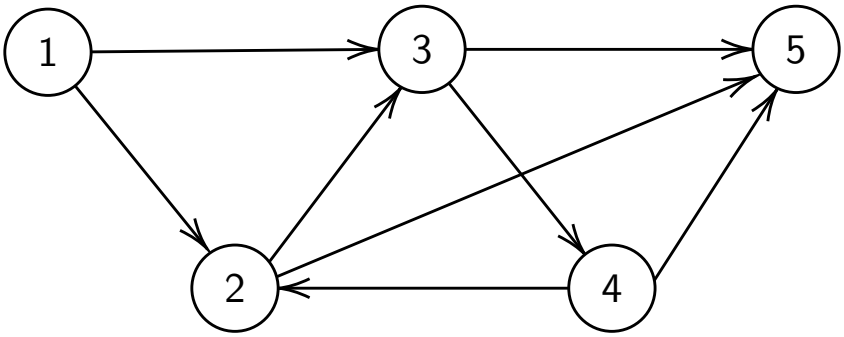
Sumário

1 Introdução a grafos
- Definições básicas
- Representações
- Um tipo abstrato de dados (TAD) para grafo
- Exercícios

Grafo

Um **grafo** ou **rede** é um modelo matemático que representa **relações** entre objetos – **nós** ou **vértices** – de um determinado conjunto. Tais relações, no modelo, são representadas por meio de **arestas** (também chamadas **arcos** ou **ramos**) que interconectam tais vértices.

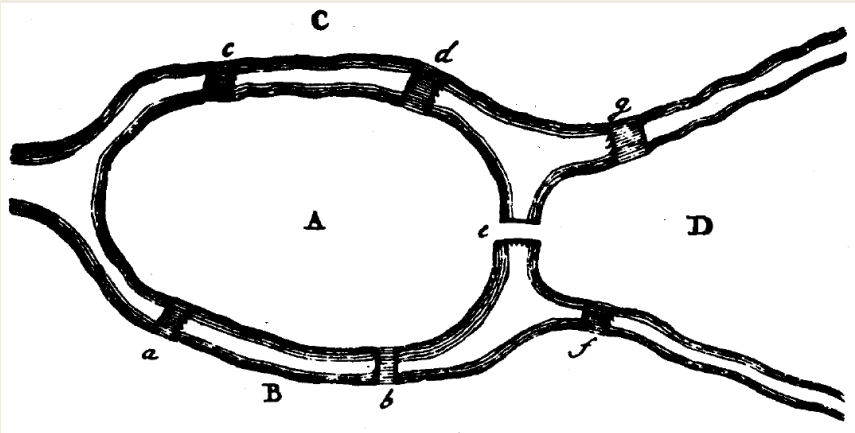
- **Notação**: $G(V, A)$: V é o conjunto – não-vazio – de **vértices**; e A é o conjunto de **arestas**.
- **Exemplo:** $V = \{1, 2, 3, 4, 5\}$,
  $A = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 2), (4, 5)\}$



Exemplo clássico

Pontes de Königsberg, Prússia

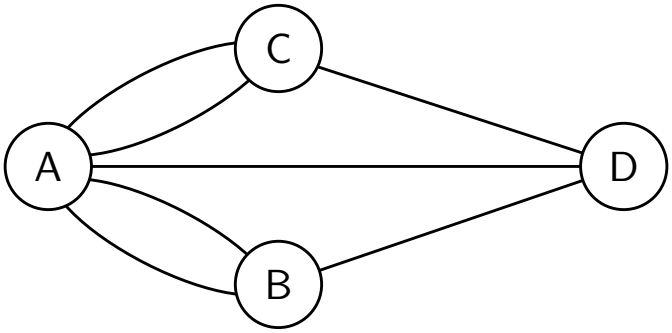
Visitar todas as quatro partes A, B, C e D da cidade (atual Kaliningrado, na Rússia), voltando ao ponto de partida e passando apenas **uma vez** por cada uma das sete pontes (a, b, . . . , g) sobre o Rio Prególia.



Fonte: Leonhard Euler - The World of Mathematics, Volume 1, Public Domain,
https://commons.wikimedia.org/w/index.php?curid=33336521

- Leonhard Euler (séc. XVIII): não há solução possível.

Representação do problema como um grafo:

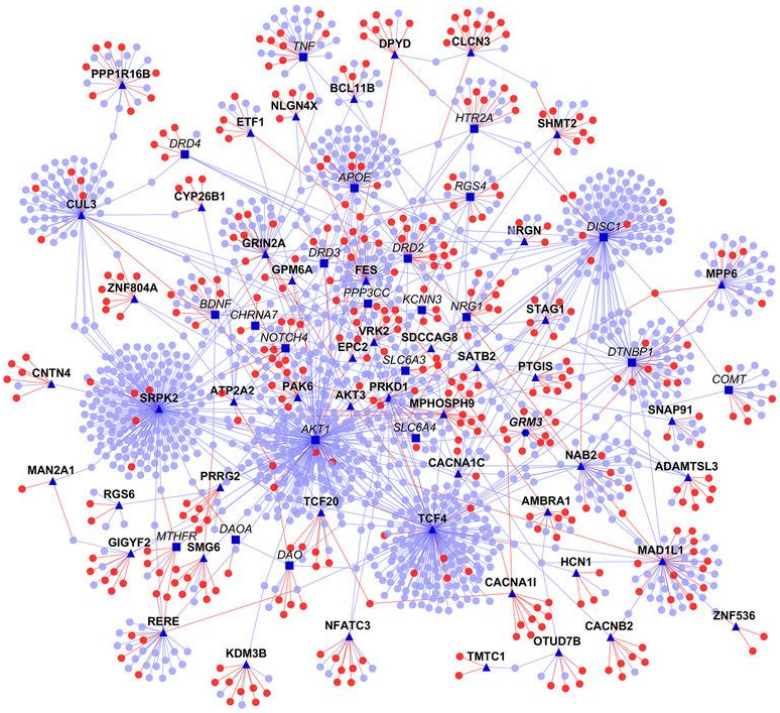


- Vértices são as partes de terra e arestas representam as pontes.
- A partir de qualquer vértice, não é possível encontrar um caminho que passe por todos os outros lugares sem que se passe mais de uma vez por pelo menos uma mesma aresta.

Exemplos de aplicações

1. **Redes sociais**: modelagem de redes como Facebook, Instagram, etc. (vértices são os perfis de usuários; arestas, as relações);
2. **Navegação**: modelagem de sistemas de navegação, como aplicações de GPS e Google Maps;
3. **Biologia**: pesquisas em redes neurais (reais), redes de proteínas, e afins;
4. **Comunicações**: telefonia e fibras óticas, redes de computadores, etc.

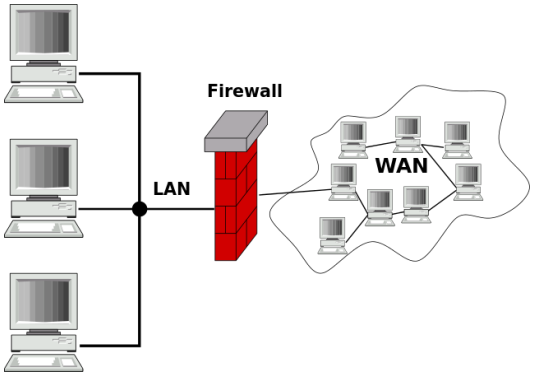
Fonte: https://www.metro.sp.gov.br/



Fonte: Madhavicmu - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=48447204

Conceitos

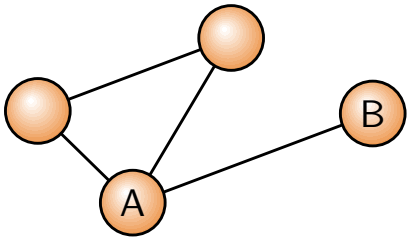
- **Vértice** é cada item do conjunto em estudo representado no grafo.
- A representação do vértice depende do problema estudado:
  - Um indivíduo (perfil) numa rede social;
  - Um lugar em um mapa;
  - Um computador ou servidor numa rede, etc.



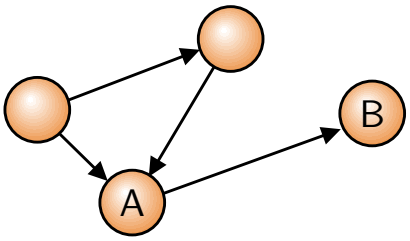
Fonte: Harald Mühlböck - selbst erstellt mithilfe von xfig und der xfig-libaries., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=815692

- Cada **aresta** representa a ligação (conexão) existente entre dois vértices.
  - Modela a **relação** entre ambos;
  - Dois vértices de um grafo são **adjacentes** – ou **vizinhos** – se existe uma aresta interligando-os:
    - ★ Contatos ou seguidores numa rede social;
    - ★ Uma interação biológica entre duas proteínas;
    - ★ Uma ferrovia interligando duas estações em um mapa, etc.

- A aresta é **dirigida** ou **orientada** se a relação entre dois elementos ocorre somente numa direção.
  - Depende da aplicação sendo modelada.
- O grafo é um **grafo dirigido** (**dígrafo**) se todas as suas arestas forem dirigidas.
  - Num dígrafo, é possível haver uma conexão a partir de um vértice A para outro B, sem existir outra conexão no sentido oposto.

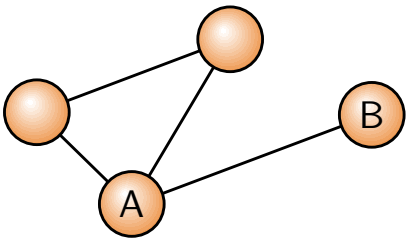


grafo

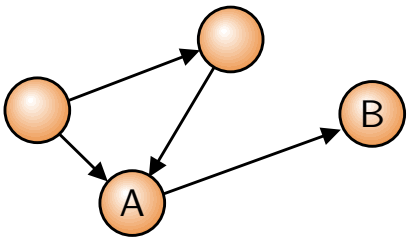


dígrafo

- **Grau**: o número de vizinhos que um vértice possui.
  - Quantidade de arestas que chegam ao vértice, ou partem do mesmo.
- No caso de um **dígrafo**, há dois tipos:
  - **Grau de entrada**: número de arestas que chegam ao vértice;
  - **Grau de saída**: número de arestas que partem do vértice;

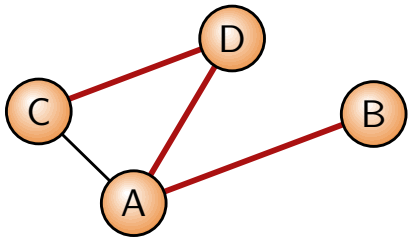


$G(A) = 3$
$G(B) = 1$

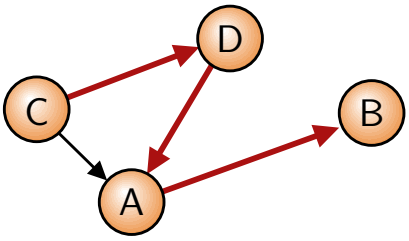


$G_E(A) = 2;\ G_S(A) = 1$
$G_E(B) = 1;\ G_S(B) = 0$

- **Caminho**: sequência de vértices do grafo, onde cada vértice está conectado ao seguinte por meio de uma aresta.
  - **Comprimento**: quantidade de vértices percorridos ao longo do caminho.



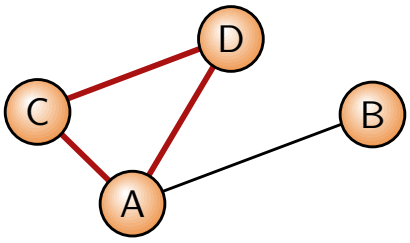
Caminhos: C-D-A-B ou B-A-D-C
(ambos comprimento: 4)



Caminho: C-D-A-B
(comprimento: 4)

- O primeiro vértice – o **vértice inicial** – está **conectado** ao último vértice da sequência – o **vértice final** – se existe um caminho interligando-os.
  - Ex.: no dígrafo do exemplo acima, C é conectado a B.
- No exemplo, todos são **caminhos simples**, pois nenhum vértice aparece mais de uma vez nos mesmos.

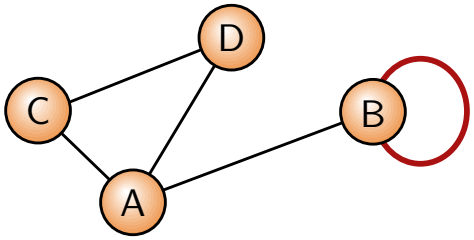
- **Ciclo**: caminho fechado, em que o vértice inicial e o final são o mesmo.
  - O **comprimento do ciclo** é o número de vértices percorridos do inicial até o final, sem contar novamente este último.



Ciclo: C-D-A
(comprimento: 3)

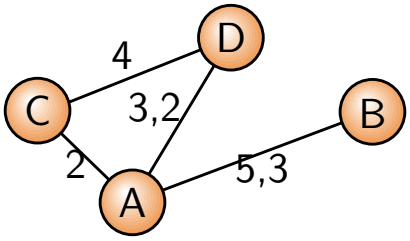
- **Grafo acíclico**: grafo que não possui **ciclos simples** – nos quais cada vértice aparece uma única vez.

- **Laço**: aresta que conecta um vértice a si próprio, i.e. o vértice de origem e o de destino são o mesmo.
  - todo laço é um ciclo de comprimento 1.

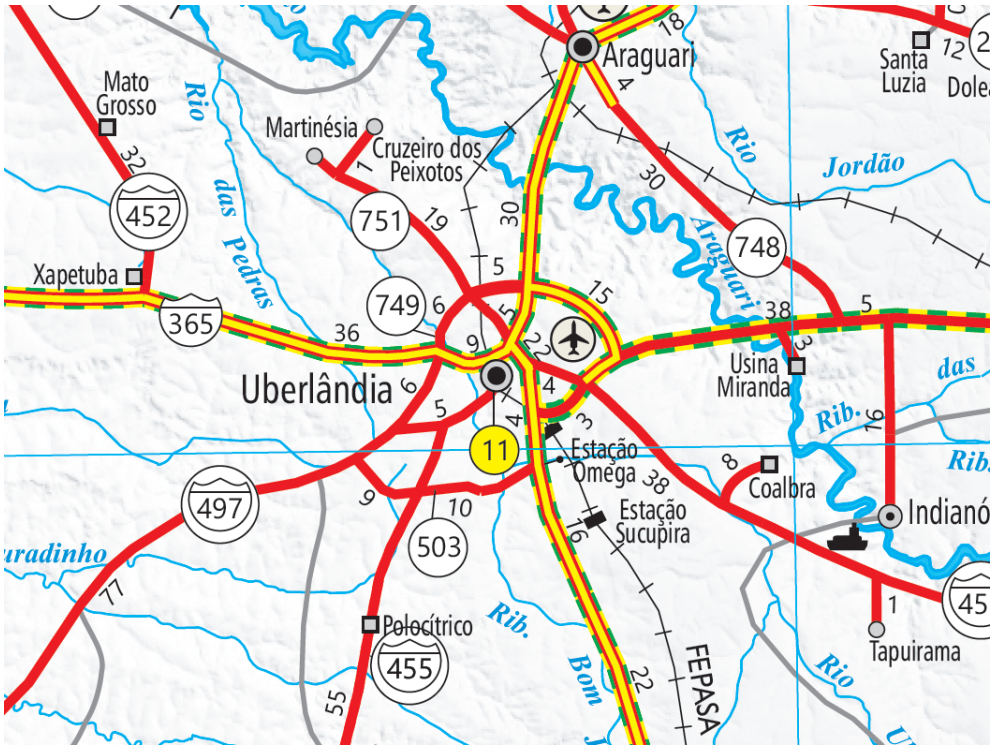


- **Multigrafo**: grafo que permite **arestas múltiplas** – quando pares de vértices adjacentes possuem mais de uma aresta os relacionando.
  - Ex.: O grafo das pontes de Königsberg visto.

- Arestas podem conter um **peso** (valor numérico) associado, representando uma grandeza associada ao relacionamento dos vértices que interliga (ex.: a **distância** entre os mesmos).
- Um grafo é dito **ponderado** quando isto ocorre.



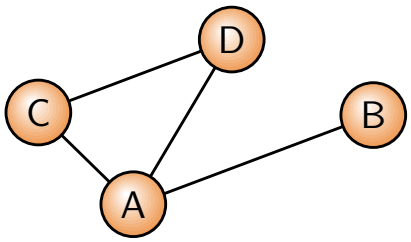
- Exemplo prático de grafo ponderado: um mapa rodoviário.



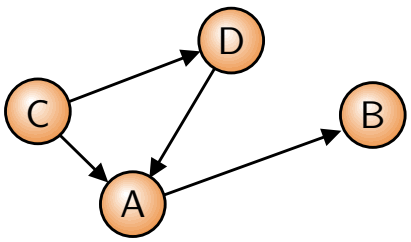
Fonte: DER-MG - https://www.der.mg.gov.br/files/1950/Mapa-Rodoviario/22325/Mapa-Rodoviario-2021.pdf

- O grafo é dito **conexo** se, para *quaisquer* dois vértices u e v, é possível encontrar um caminho de u para v **ou** vice-versa.
  - o grafo é **fortemente conexo** se para qualquer par u e v, há **ambos os caminhos** – de u para v e de v para u.

Todo grafo não-direcionado conexo é, na verdade, **fortemente conexo**.



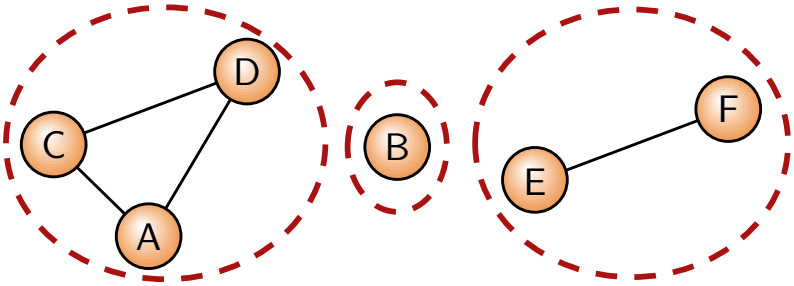
grafo conexo
(fortemente conexo)



dígrafo fracamente conexo

- O dígrafo acima é **fracamente conexo**: por exemplo, dado o par A e D, existe um caminho de D até A, mas não de A até D.

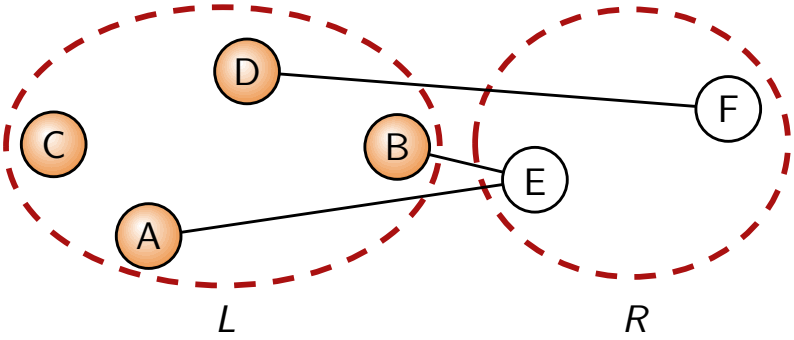
- Se um grafo ou dígrafo não é conexo, então o mesmo é **desconexo**.
  - Cada subgrafo[1] de um grafo desconexo que é, por si, conexo, é chamado de **componente conexa**.



  - Há pelo menos duas componentes conexas num grafo desconexo.

[1] $G_s(V_s, A_s)$ é **subgrafo** de um grafo $G(V, A)$ se $V_s \subseteq V$ e $A_s \subseteq A$

- Um grafo $G(V, A)$ é **bipartido** se seu conjunto de vértices V puder ser dividido em dois subconjuntos **disjuntos** – sem intersecção – L e R tais que toda aresta em E somente conecte vértices de subconjuntos diferentes.
  - i.e. se $(u, v) \in E$, então necessariamente $u \in L$ e $v \in R$, ou vice-versa, onde $L \cup R = V$.



Sumário

1 Introdução a grafos
- Definições básicas
- Representações
- Um tipo abstrato de dados (TAD) para grafo
- Exercícios

Representação computacional

- Duas abordagens principais:
  - **Matriz de adjacências**
  - **Lista de adjacências**
- A melhor representação, dentre as duas, depende da aplicação. Não existe uma ideal para todos os casos.

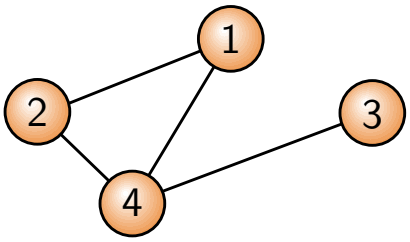
Matriz de adjacências

- Um grafo $G(V, A)$ contendo n vértices, i.e. $|V| \equiv n$, é representado numa matriz $\mathbf{A} = \{a_{ij}\}$, quadrada $n \times n$, tal que:

$$a_{ij} = \begin{cases} 1 & \text{se } (i,j) \in A\,; \\ 0 & \text{caso contrário.} \end{cases}$$

- $n \times n$: custo computacional é $\Theta(n^2)$ – independe do número de arestas $|A|$ do grafo.
  - Não indicada para grafos com n alto, mas poucas arestas (grafo pouco denso).

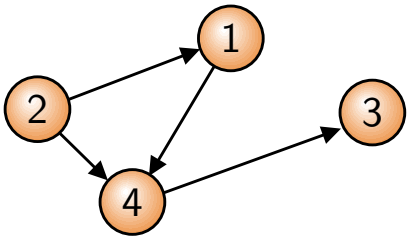
- Matriz de adjacências de um grafo (não-direcionado):



	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0

  - A matriz é sempre **simétrica**.

- Matriz de adjacências de um dígrafo:

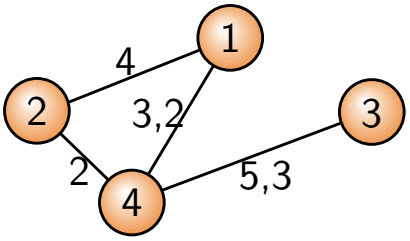


	1	2	3	4
1	0	0	0	1
2	1	0	0	1
3	0	0	0	0
4	0	0	1	0

  - A simetria ocorre somente se para cada aresta $(i, j) \in A$ houver outra correspondente $(j, i) \in A$, para todo $i, j \in V$.

- Se o grafo ou dígrafo é ponderado: representa-se o **peso** de cada aresta diretamente na matriz **A**, i.e.:

$$a_{ij} = \begin{cases} \omega_{ij} & \text{se } (i, j) \in A \text{ e tem peso } \omega_{ij}\,; \\ 0 & \text{caso contrário.} \end{cases}$$

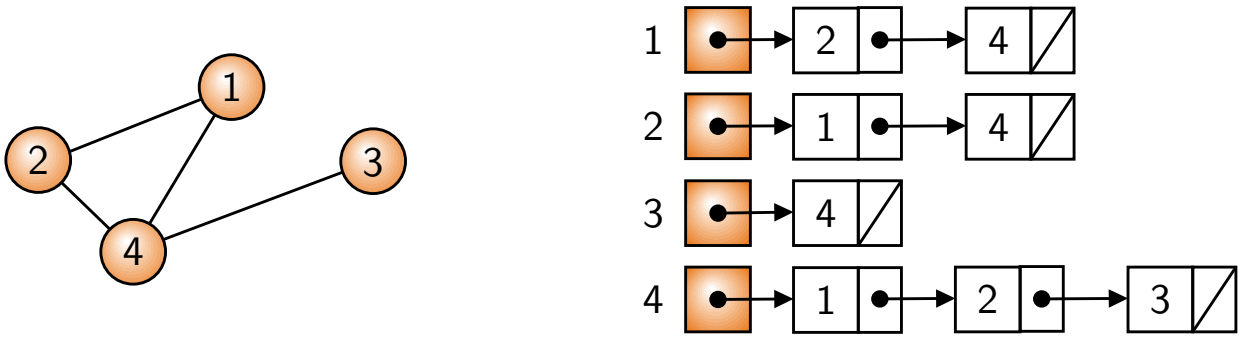


	1	2	3	4
1	0	4	0	3,2
2	4	0	0	2
3	0	0	0	5,3
4	3,2	2	5,3	0

Lista de adjacências

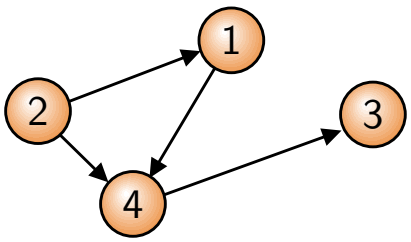
- Um grafo $G(V, A)$ contendo n vértices e m arestas, i.e. $|V| \equiv n$ e $|A| \equiv m$, é representado por um *array* (vetor) Adj com n listas, uma para cada vértice de V.
- Para cada vértice $u \in V$, a lista $Adj[u]$ contém todos os vértices adjacentes a u em G (representados por **índices** ou **ponteiros**).
  - Ou seja, para cada vértice $u \in V$, $Adj[u]$ contém todos os vértices v tais que a aresta (u, v) pertença a A.
- O custo computacional é $\Theta(n + m)$ – em geral menor que o custo no caso de matriz de adjacências.
  - Indicada para grafos com n alto, mas poucas arestas.

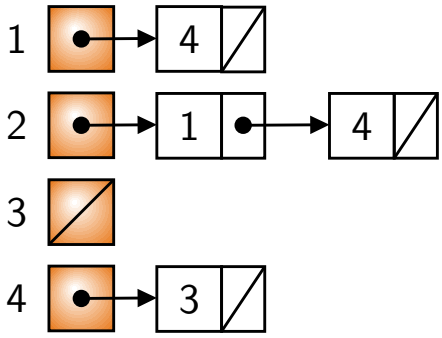
- Lista de adjacências de um grafo (não-direcionado):



  - Cada aresta $(u, v) \in A$ é representada duas vezes: uma na lista de adjacências do vértice u, e outra na lista do vértice v.
  - A soma dos comprimentos das listas de adjacências dos vértices corresponde a $2m \equiv 2|A|$.

- Lista de adjacências de um dígrafo:





  - Cada aresta $(u, v) \in A$ é representada uma única vez.
  - A soma dos comprimentos das listas de adjacências dos vértices corresponde a $m \equiv |A|$.

Sumário

1 Introdução a grafos
- Definições básicas
- Representações
- Um tipo abstrato de dados (TAD) para grafo
- Exercícios

Um tipo abstrato de dados (TAD) para grafo

- Consideramos um tipo abstrato de dados **opaco** em Linguagem C para representação por **lista de adjacências**.

Tipo opaco

- A ED será referenciada mediante um **ponteiro** para a mesma, ao invés de uma variável do tipo `Grafo` diretamente.
- É como ocorre, por exemplo, com o tipo `FILE` em C.
- Isto permite ocultar a implementação, permitindo sua alteração sem interferir no funcionamento de qualquer código que o utilize – desde que a **interface** seja respeitada.

- Modularização do TAD:
  - **grafo.h**: a **interface** em si: definição do tipo opaco; funções disponíveis para manipular o grafo (protótipos).
  - **grafo.c**: o que deve ficar oculto do usuário, além da implementação das funções definidas pela interface. Inclui a definição do **tipo grafo**.
- **Tipo grafo**: estrutura que contém:
  - eh_ponderado:se as arestas têm peso ou não (i.e. grafo ponderado?);
  - `nro_vert`: número de vértices que o grafo – tamanho do conjunto V;
  - `grau_max`: número máximo de arestas a qual um vértice poderá se conectar;
  - `arestas`: ponteiro onde é alocada a matriz de arestas do grafo;
  - `pesos`: ponteiro onde é alocada a matriz de pesos das arestas do grafo – caso o mesmo seja ponderado;
  - `grau`: ponteiro onde é alocado um vetor que armazena o número de arestas já associadas a um vértice.

Interface: grafo.h

grafo.h

```c
typedef struct grafo Grafo;
Grafo* cria_Grafo(int nro_vertices, int grau_max, int
    eh_ponderado);
void libera_Grafo(Grafo* gr);
int insereAresta(Grafo* gr, int orig, int dest, int
    eh_digrafo, float peso);
int removeAresta(Grafo* gr, int orig, int dest, int
    eh_digrafo);
```


Implementação do TAD opaco Grafo: grafo.c

grafo.c: bibliotecas usadas e definição do tipo

```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h" //inclui os protótipos

//Definição do tipo Grafo
struct grafo{
  int eh_ponderado;
  int nro_vertices;
  int grau_max;  // tamanho das listas
  int** arestas; // array de arestas
  float** pesos;
  int* grau; // quantidade de elementos em cada lista
};
```

Criando um grafo I

Exemplo de uso

```
Grafo *gr;
gr = cria_Grafo(10, 7, 0);
```

grafo.c: criação do grafo

```
Grafo* cria_Grafo(int nro_vertices, int grau_max, int eh_ponderado){
  Grafo *gr;
  gr = (Grafo*) malloc(sizeof(struct grafo));
  if(gr != NULL){
    int i;
    gr->nro_vertices = nro_vertices;
    gr->grau_max = grau_max;
    gr->eh_ponderado = (eh_ponderado != 0)?1:0;
    gr->grau = (int*) calloc(nro_vertices,sizeof(int));

```


Criando um grafo II

```
    gr->arestas = (int**) malloc(nro_vertices * sizeof(int*));
    for(i=0; i<nro_vertices; i++)
      gr->arestas[i] = (int*) malloc(grau_max * sizeof(int));

    if(gr->eh_ponderado){
      gr->pesos = (float**) malloc(nro_vertices * sizeof(float*));
      for(i=0; i<nro_vertices; i++)
        gr->pesos[i] = (float*) malloc(grau_max * sizeof(float));
    }
  }
  return gr;
}
```

```
gr = cria_Grafo(10, 7, 0);
```

- Cria um grafo com 10 vértices;
- Cada um pode se conectar com até outros 7;
- Matriz 10×7 para as arestas;
- Vetor grau indica o número de conexões para cada vértice

arestas

	0	1	2	3	4	5	6
0							
1							
2							
3							
4							
5							
6							
7							
8							
9							

grau

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Liberando o grafo da memória I

Exemplo de uso

```
Grafo *gr;
gr = cria_Grafo(10, 7, 0);
.
.
libera_Grafo(gr);
```

grafo.c: liberando o grafo

```
38 void libera_Grafo(Grafo* gr){
39   if(gr != NULL){
40     int i;
41
42     // libera a matriz de arestas
43     for(i=0; i<gr->nro_vertices; i++)
44       free(gr->arestas[i]);
45     free(gr->arestas);
```

Liberando o grafo da memória II

```
46
47      // libera a matriz de pesos
48      if(gr->eh_ponderado){
49        for(i=0; i<gr->nro_vertices; i++)
50          free(gr->pesos[i]);
51        free(gr->pesos);
52      }
53      free(gr->grau);
54      free(gr);
55    }
56  }
```


Inserindo uma aresta I

Exemplo de uso

```
Grafo *gr;
gr = cria_Grafo(10, 7, 0);
insereAresta(gr, 0, 1, 0, 0);
insereAresta(gr, 1, 3, 0, 0);
```

grafo.c: inserindo uma aresta no grafo

```
58  int insereAresta(Grafo* gr, int orig, int dest, int
      eh_digrafo, float peso){
59    if(gr == NULL)
60      return 0;
61
62    // verifica a existência dos vértices de origem e
      destino
63    if(orig < 0 || orig >= gr->nro_vertices)
64      return 0;
```

Inserindo uma aresta II

```
  if(dest < 0 || dest >= gr->nro_vertices)
    return 0;

  // insere aresta ao final da linha
  gr->arestas[orig][gr->grau[orig]] = dest;
  if(gr->eh_ponderado)
    gr->pesos[orig][gr->grau[orig]] = peso;
  gr->grau[orig]++;

  // insere a outra aresta se não for dígrafo
  if (eh_digrafo == 0)
    insereAresta(gr,dest,orig,1,peso);
  return 1;
}
```


`insereAresta(gr, 0, 1, 0, 0);`

arestas

	0	1	2	3	4	5	6
0	1						
1	0						
2							
3							
4							
5							
6							
7							
8							
9							

grau

0	1
1	1
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

`insereAresta(gr, 1, 3, 0, 0);`

arestas

	0	1	2	3	4	5	6
0	1						
1	0	3					
2							
3	1						
4							
5							
6							
7							
8							
9							

grau

0	1
1	2
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

Removendo uma aresta

Exemplo de uso

```
Grafo *gr;
gr = cria_Grafo(10, 7, 0);
insereAresta(gr, 0, 1, 0, 0);
insereAresta(gr, 1, 3, 0, 0);
.
.
removeAresta(gr, 0, 1, 0, 0);
```

- Caso o ponteiro para grafo não seja nulo, é preciso **buscar** pela aresta a ser removida.
- A implementação não é encadeada: é necessário, nas listas de adjacência dos vértices envolvidos, ocupar as respectivas posições sendo disponibilizadas pela exclusão.
- Como `insereAresta()`, retorna 1 se exclusão for bem-sucedida, e 0 caso contrário.

grafo.c: removendo uma aresta do grafo

```
80 int removeAresta(Grafo* gr, int orig, int dest, int
     eh_digrafo){
81   if(gr == NULL)
82     return 0;
83   if(orig < 0 || orig >= gr->nro_vertices)
84     return 0;
85   if(dest < 0 || dest >= gr->nro_vertices)
86     return 0;
87
88   // procura pela aresta
89   int i = 0;
90   while(i < gr->grau[orig] && gr->arestas[orig][i] !=
     dest)
91     i++;
92   if(i == gr->grau[orig]) // elemento não encontrado
93     return 0;
94   gr->grau[orig]--;
```

```
  gr->arestas[orig][i] = gr->arestas[orig][gr->grau[orig]];
  if(gr->eh_ponderado)
    gr->pesos[orig][i] = gr->pesos[orig][gr->grau[orig]];
  if(eh_digrafo == 0)
    removeAresta(gr,dest,orig,1);
  return 1;
}
```



```
insereAresta(gr, 0, 1, 0, 0);
insereAresta(gr, 1, 3, 0, 0);
```

arestas

	0	1	2	3	4	5	6
0	1						
1	0	3					
2							
3	1						
4							
5							
6							
7							
8							
9							

grau

0	1
1	2
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

```
removeAresta(gr, 0, 1, 0, 0);
```

arestas

	0	1	2	3	4	5	6
0	~~1~~						
1	3 ←						
2							
3	1						
4							
5							
6							
7							
8							
9							

grau

0	0
1	1
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

Sumário

1. Introdução a grafos
   - Definições básicas
   - Representações
   - Um tipo abstrato de dados (TAD) para grafo
   - Exercícios

Exercícios I

1. Qual é o custo computacional da exclusão (ou mesmo a consulta) de uma aresta no código visto?
2. Este custo seria o mesmo para a implementação por matriz de adjacência? Justifique.
3. A partir da implementação vista, adapte o código para a versão com lista encadeada.

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.
2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por André R. Backes (UFSCar); Bruno Travençolo e Christiane Brasil (FACOM/UFU).
TAD grafo: disponível em:
https://www.facom.ufu.br/~backes/wordpress/ProjGrafo.zip.
Acesso em: 22/jan/2025.
Adaptações: Renato Pimentel, FACOM/UFU