



RAFAEL VIEIRA COELHO



GRAFOS

SUMÁRIO

- Estruturas de Dados
- Listas Simplesmente Encadeada
- Listas Duplamente Encadeada
- Filas
- Pilhas
- **Grafos**
- Árvores

ANDRÉ BACKES

Estrutura de dados descomplicada em linguagem

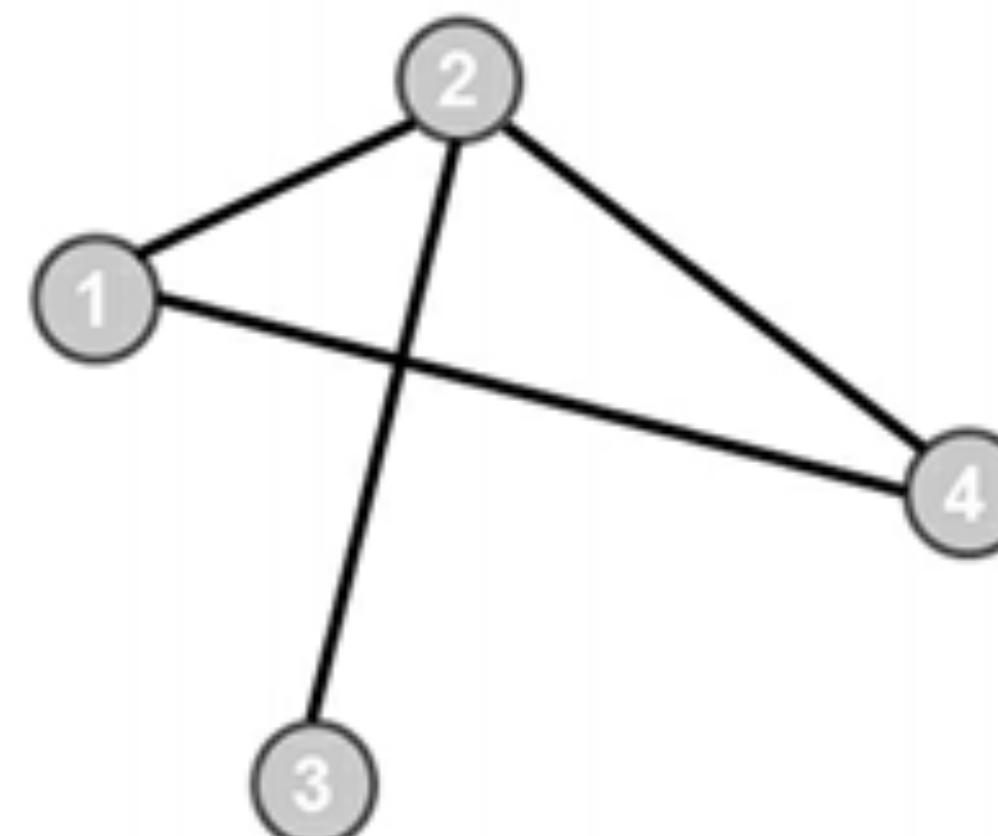
C

DEFINIÇÃO DE GRATOS

Trata-se de um modelo matemático que representa relações entre objetos de determinado conjunto.

Um **grafo $G(V,A)$** é definido em dois termos de dois conjuntos: (V - vértices e A - Areias)

Dois vértices são **adjacentes** se existir uma aresta ligando-os.



$G(V,A)$

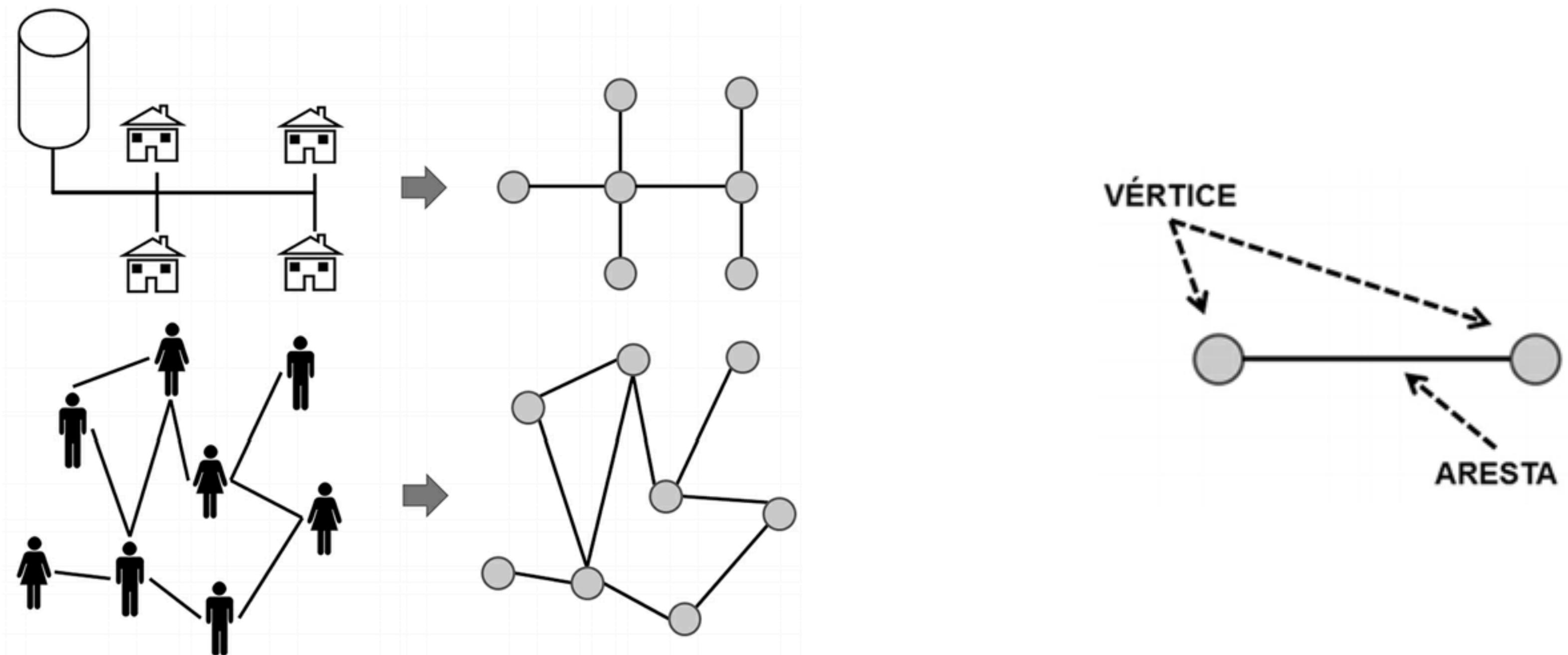
$V = \{1,2,3,4\}$

$A = \{\{1,2\}, \{1,4\}, \{2,3\}, \{2,4\}\}$

VÉRTICES X ARESTAS

Praticamente qualquer objeto (seja ele um pixel de uma imagem, uma pessoa de um grupo de amigos ou mesmo uma cidade em um mapa) pode ser representado como um vértice em um grafo.

Consequentemente, as ligações entre os vértices, isto é, as arestas do grafo, podem ser estabelecidas de acordo com alguma medida que represente adequadamente o relacionamento existente entre os pares de vértices.

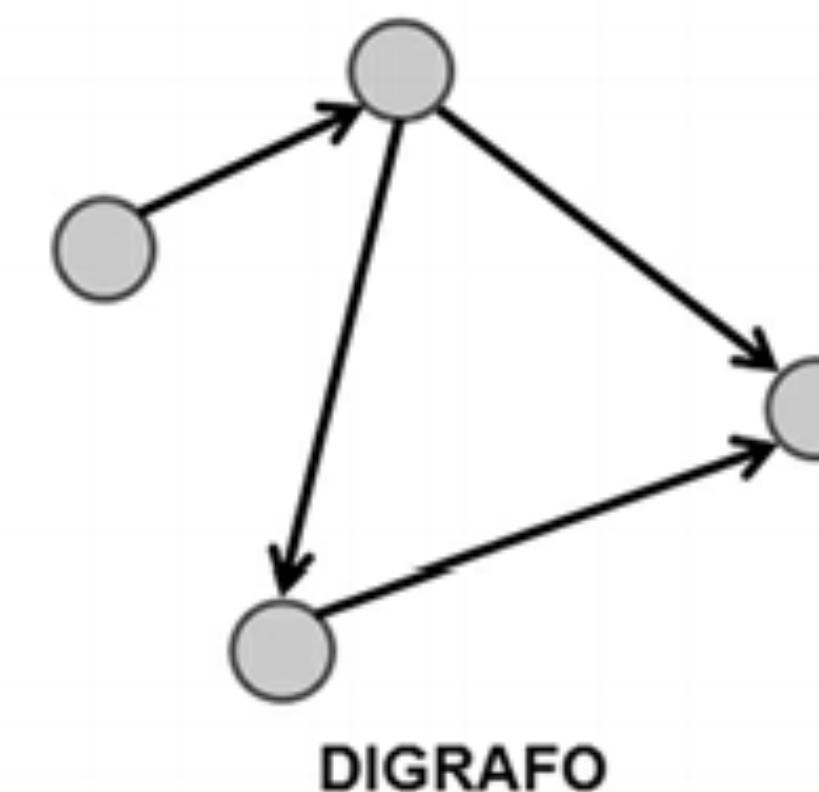
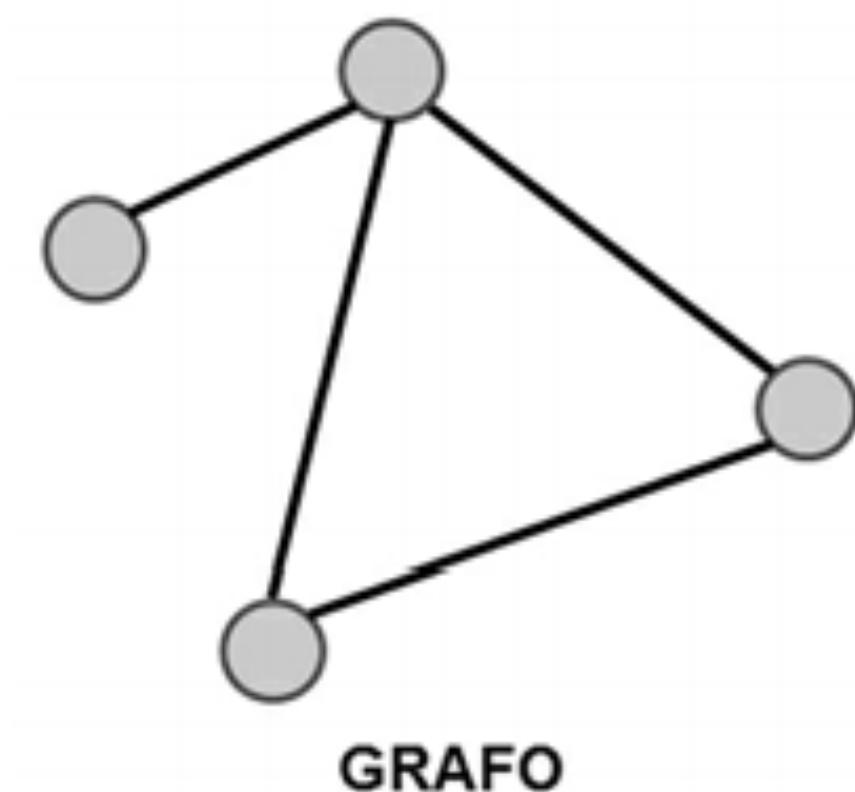


DIREÇÃO DAS ARESTAS

Dependendo da aplicação na qual um grafo é usado, as arestas podem ou não ter uma direção associada a cada uma delas.

Em um **grafo direcionado ou digrafo**, existe uma orientação quanto ao sentido da aresta, ou seja, se uma aresta liga os vértices A a B, isso significa que podemos ir de A para B, mas não o contrário.

Já em um **grafo não direcionado**, não existe nenhuma orientação quanto ao sentido da aresta, ou seja, se uma aresta liga os vértices A a B, isso significa que podemos ir de A para B ou de B para A.



GRAU DE UM VÉRTICE

O **grau** de um vértice corresponde ao número de arestas que conectam aquele vértice a outro vértice do grafo.

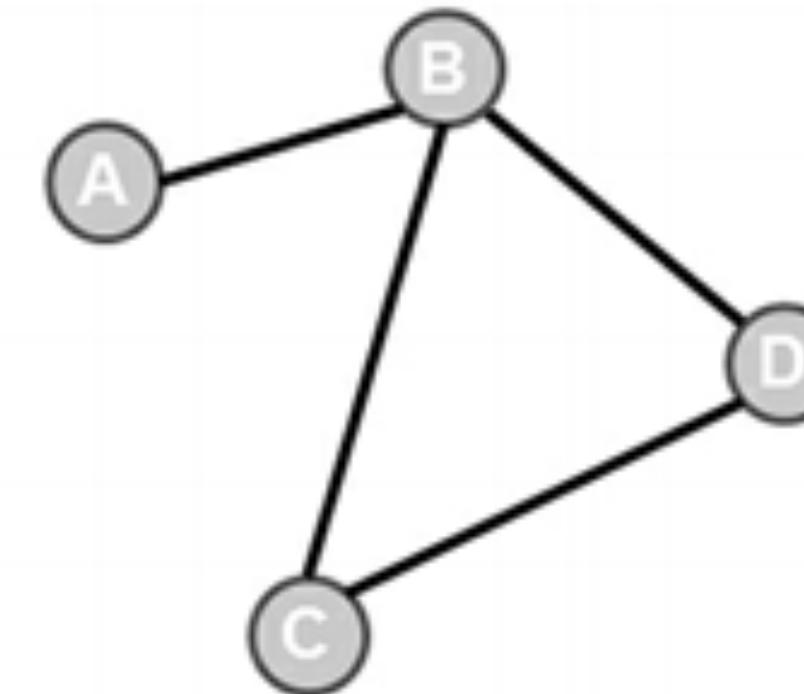
Em outras palavras, trata-se do número de vizinhos que aquele vértice possui no grafo.

No caso dos dígrafos, temos dois tipos de grau:

- **Grau de entrada:** corresponde ao número de arestas que chegam ao vértice partindo de outro.
- **Grau de saída:** corresponde ao número de arestas que partem do vértice em direção a outro.

GRAU DE UM VÉRTICE

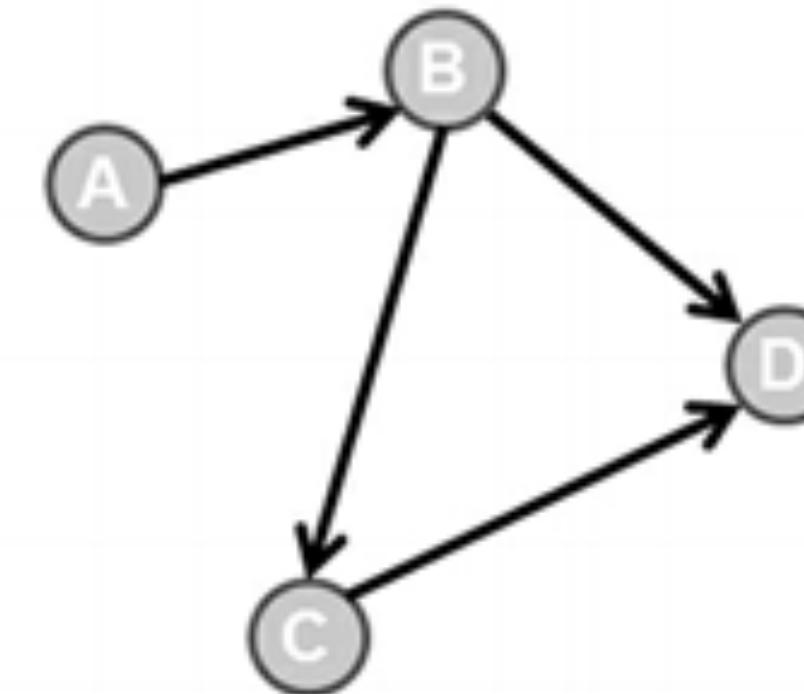
GRAFO



Grau

$G(A) = 1$
$G(B) = 3$
$G(C) = 2$
$G(D) = 2$

DIGRAFO

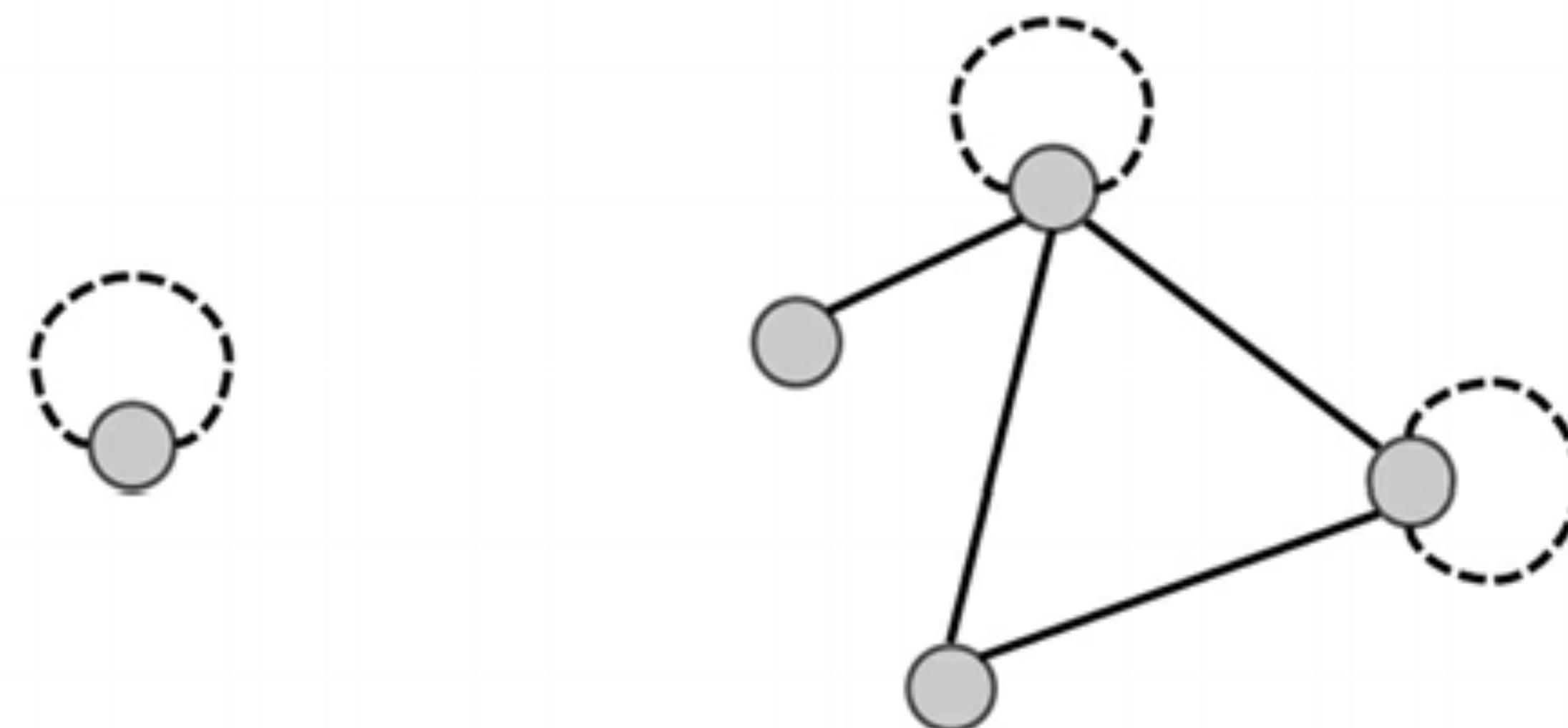


Grau Entrada	Grau Saída
--------------	------------

$G(A) = 0$	$G(A) = 1$
$G(B) = 1$	$G(B) = 2$
$G(C) = 1$	$G(C) = 1$
$G(D) = 2$	$G(D) = 0$

LAÇOS

Uma aresta é chamada laço se seu vértice de partida é o mesmo que o de chegada, ou seja, a aresta conecta o vértice a ele mesmo.

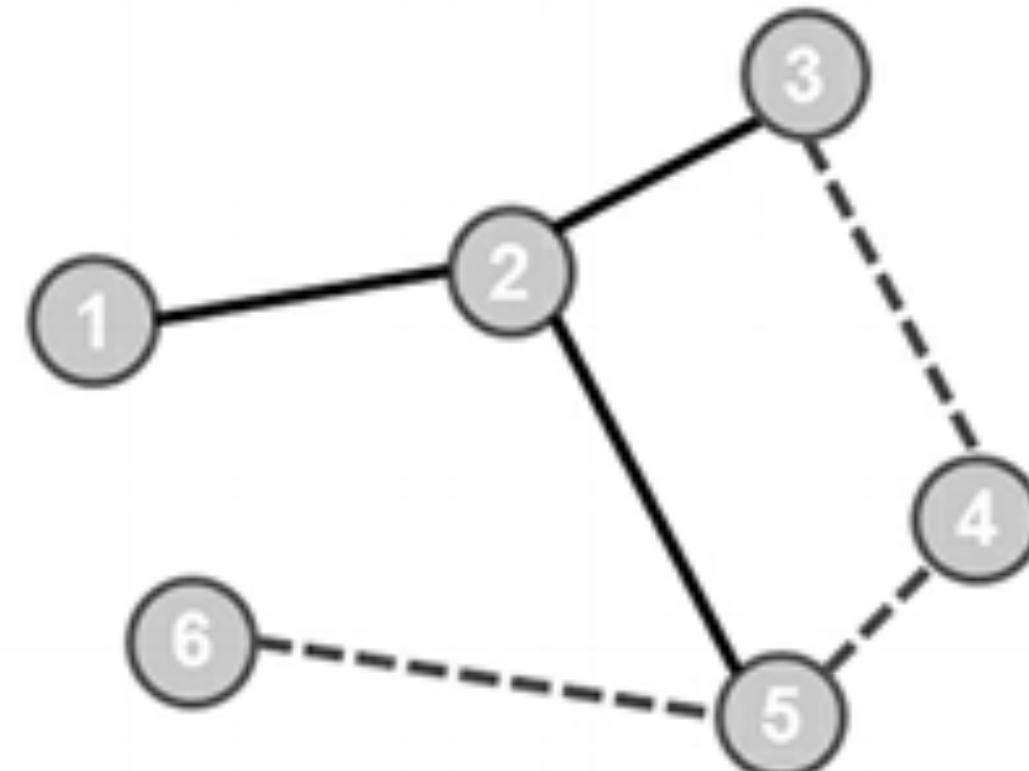


CAMINHOS

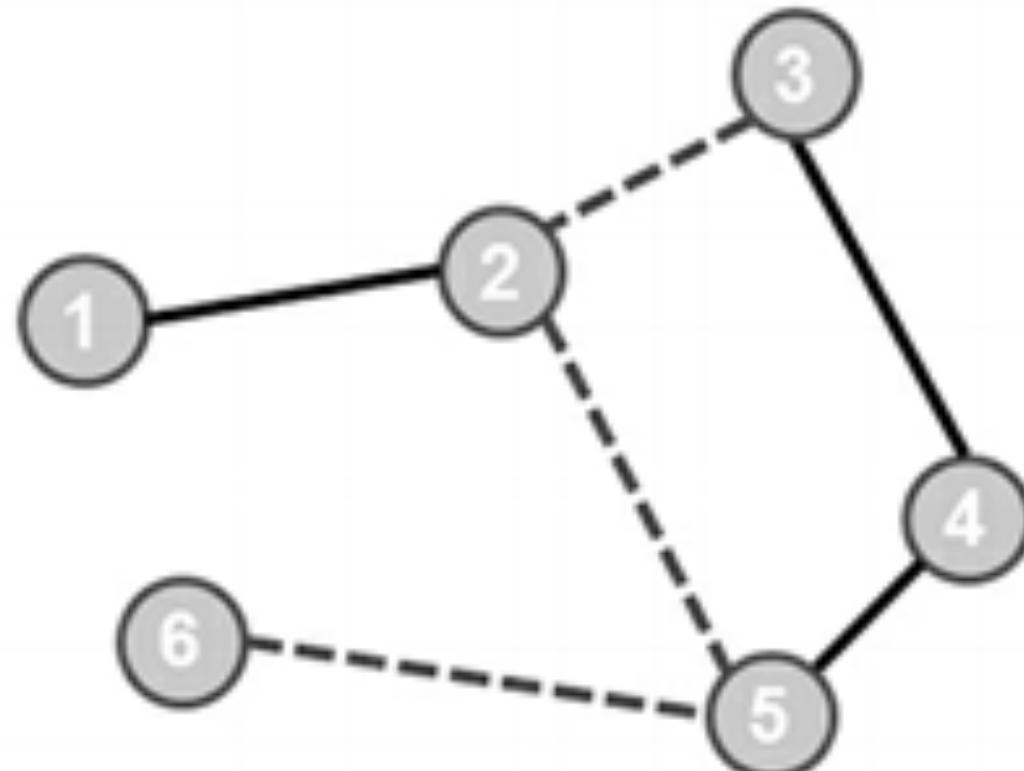
A grande maioria dos grafos são **esparsos**, ou seja, apresentam apenas uma pequena fração de todas as arestas possíveis.

Isso significa que um vértice se conecta com poucos vértices vizinhos.

No entanto, dois vértices não adjacentes (v_1 e v_5) podem ser conectados por uma sequência de arestas, como $(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5)$.



CAMINHO: 3-4-5-6



CAMINHO: 3-2-5-6



Um **caminho** entre dois vértices é uma sequência de vértices em que cada vértice está conectado ao vértice seguinte por meio de uma aresta. Neste caso, o número de vértices que precisamos percorrer de um vértice até o outro é denominado **comprimento do caminho**.



Um **caminho** é chamado **caminho simples** se nenhum dos vértices se repetir ao longo dele.

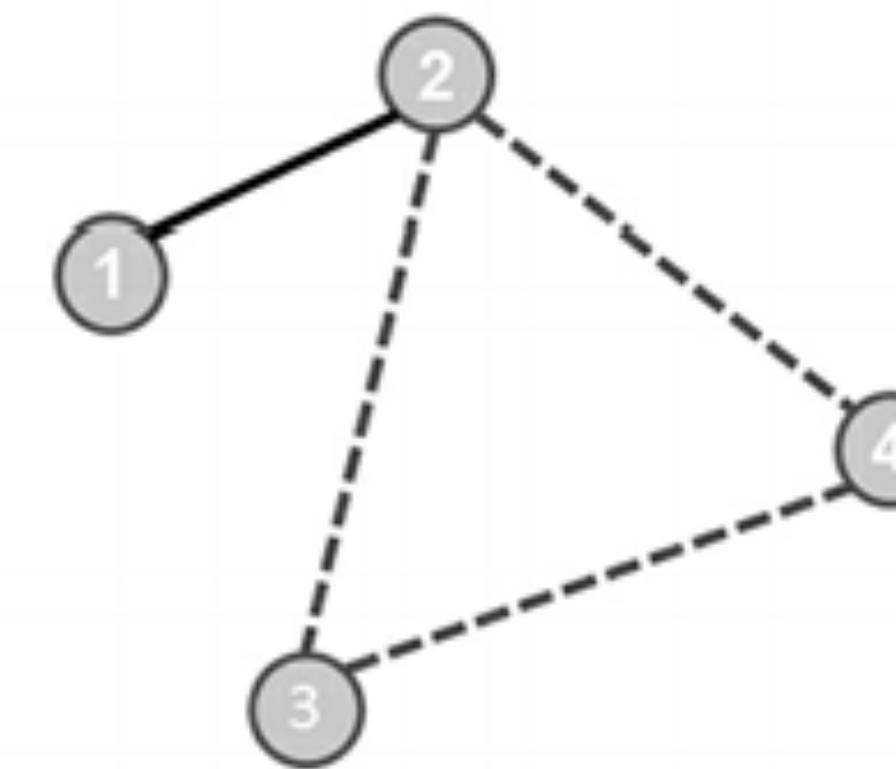
CICLOS

Um ciclo é um caminho em que os vértices inicial e final são o mesmo vértice.

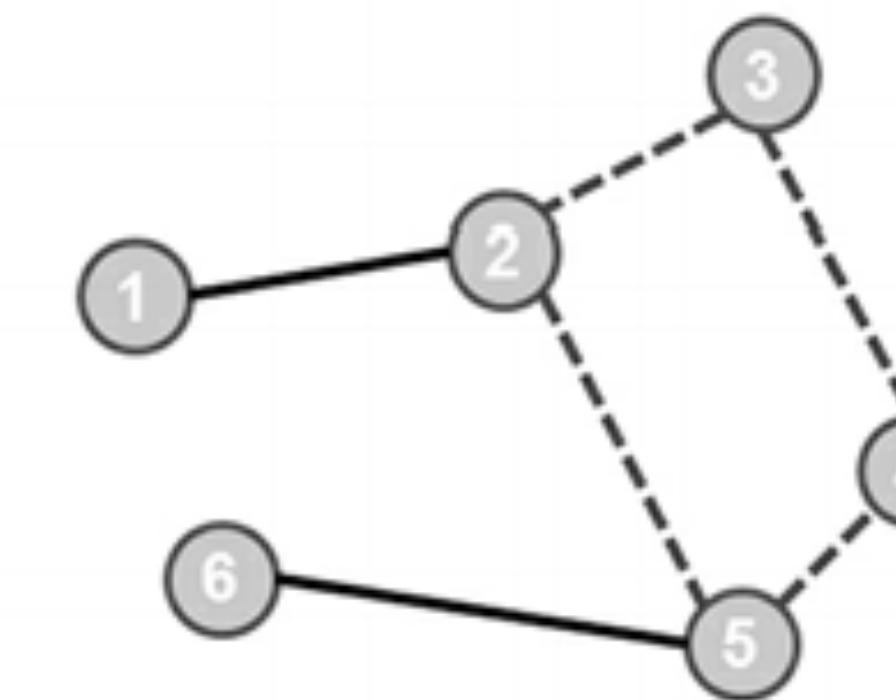
Neste caso, o **comprimento do ciclo** é o número de vértices que precisamos percorrer do vértice inicial até o final, onde o vértice final não é contado pois ele já foi contado como inicial.



Um **grafo acíclico** é um grafo que não contém ciclos simples. Um ciclo é dito simples se cada vértice aparece apenas uma vez no ciclo.



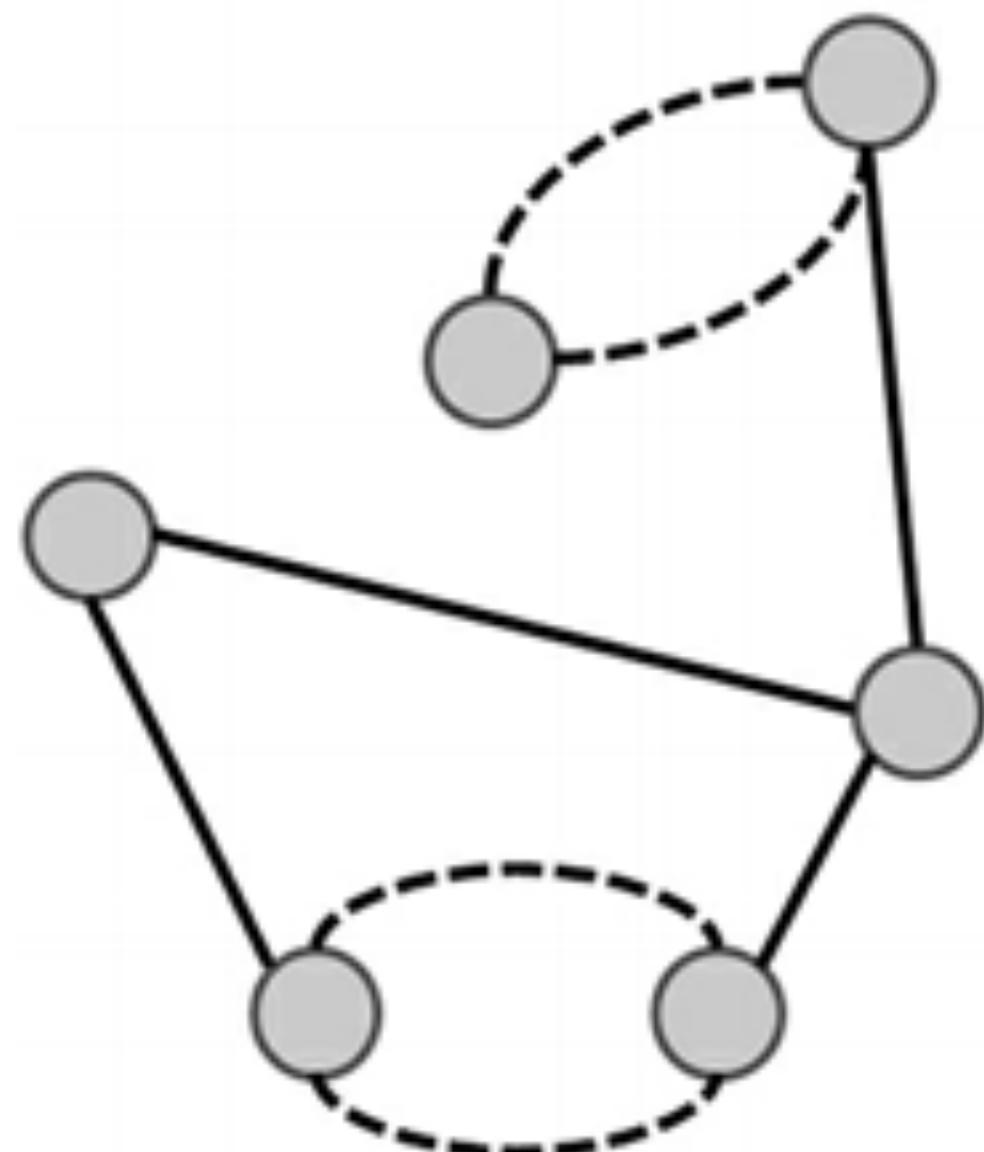
CICLO: 2-3-4



CICLO: 2-3-4-5

ARESTAS MÚLTIPLAS E MULTIGRAFO

Um grafo que possui **arestas múltiplas** é chamado multigrafo. Trata-se de um tipo de grafo especial que permite mais de uma aresta conectando o mesmo par de vértices. Neste caso, as arestas são ditas **paralelas**.

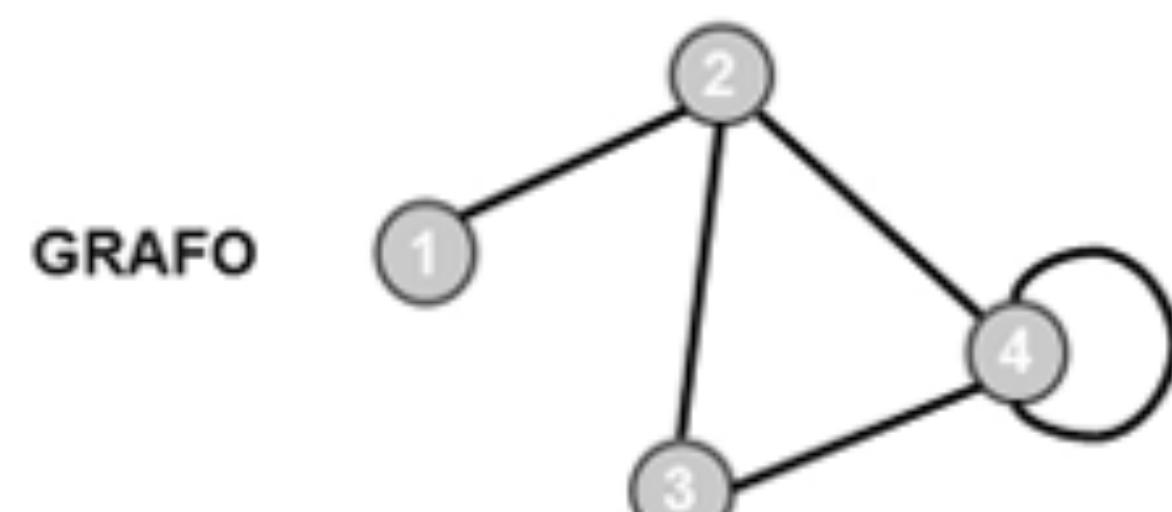


REPRESENTAÇÃO DE GRAFOS

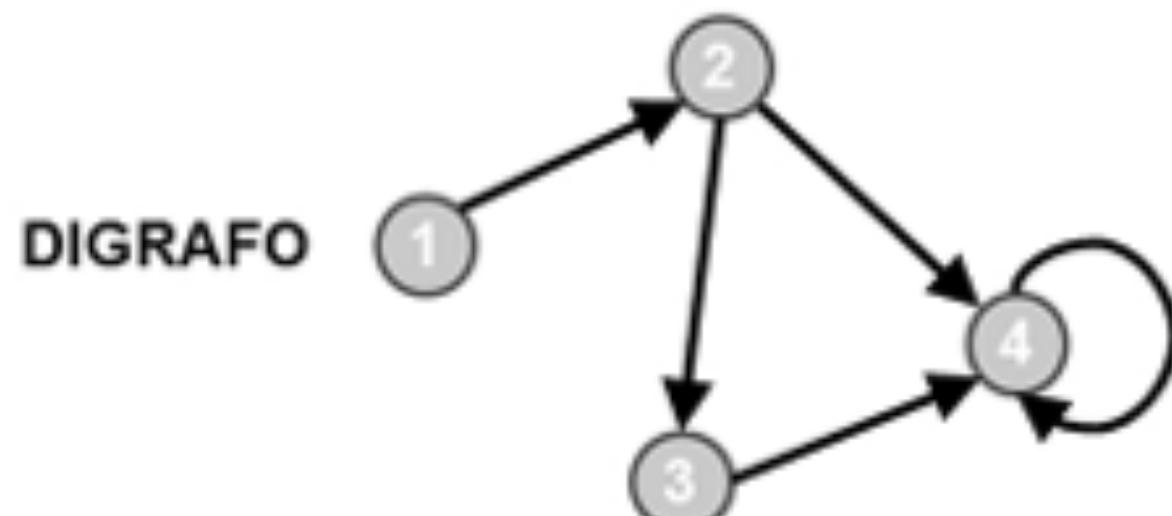
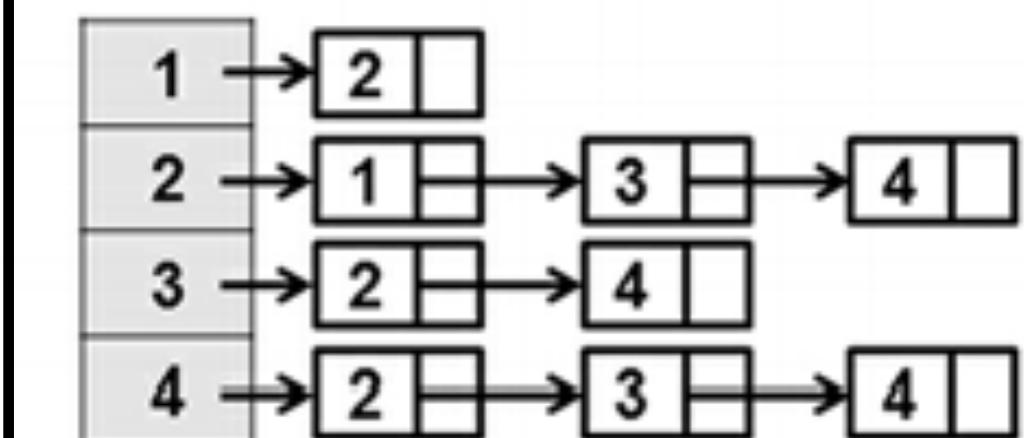
Um grafo pode ser representado de duas formas:

- Matriz de adjacência.
- Lista de adjacência.

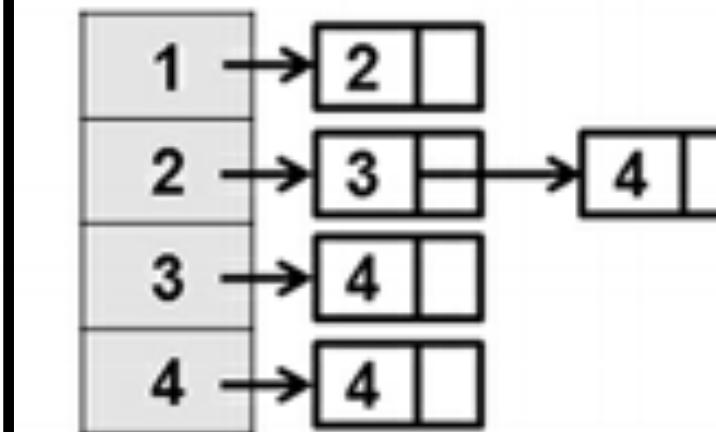
A representação escolhida depende da aplicação que está sendo desenvolvida.



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



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



TAD GRAFO

Note que o nosso tipo grafo é uma estrutura contendo vários campos:

- **eh_ponderado**: define se as arestas têm ou não peso (grafo ponderado ou não).
- **nro_vert**: define o número de vértices que o grafo irá ter.
- **Gmax**: define o número de arestas com as quais um vértice vértice poderá se conectar.
- **Arestas**: ponteiro no qual será alocada a matriz de arestas do grafo.
- **Pesos**: ponteiro no qual será alocada a matriz de pesos das arestas do grafo, se o grafo for ponderado.
- **Grau**: ponteiro no qual será alocado um vetor que irá armazenar o número de arestas já associadas a um vértice.

Arquivo Grafo.h

```
01 typedef struct grafo Grafo;
02 Grafo* cria_Grafo(int nro_vert,int Gmax,int eh_ponderado);
03 void libera_Grafo(Grafo* gr);
04 int insereAresta(Grafo* gr,int orig,int dest,
                     int digrafo,float peso);
05 int removeAresta(Grafo* gr,int orig,int dest,int digrafo);
```

Arquivo Grafo.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "Grafo.h" //inclui os Protótipos
04 //Definição do tipo Grafo
05 struct grafo{
06     int eh_ponderado;
07     int nro_vert;
08     int Gmax;
09     int** arestas;
10     float** pesos;
11     int* grau;
12 };
```

Criando um grafo

```
01 Grafo* cria_Grafo(int nro_vert, int Gmax, int eh_ponderado) {
02     Grafo *gr;
03     gr = (Grafo*) malloc(sizeof(struct grafo));
04     if(gr != NULL) {
05         int i;
06         gr->nro_vert = nro_vert;
07         gr->Gmax = Gmax;
08         gr->eh_ponderado = (eh_ponderado != 0)?1:0;
09         gr->grau=(int*)calloc(nro_vert,sizeof(int));
10
11         gr->arestas=(int**)malloc(nro_vert*sizeof(int*));
12         for(i=0; i<nro_vert; i++)
13             gr->arestas[i] = (int*)malloc(Gmax * sizeof(int));
14
15         if(gr->eh_ponderado) {
16             gr->pesos=(float**)malloc(nro_vert*sizeof(float*));
17             for(i=0; i<nro_vert; i++)
18                 gr->pesos[i]=(float*)malloc(Gmax*sizeof(float));
19         }
20     }
21     return gr;
22 }
```

Destruindo um grafo

```
01 void libera_Grafo(Grafo* gr) {  
02     if(gr != NULL){  
03         int i;  
04         for(i=0; i<gr->nro_vert; i++)  
05             free(gr->arestas[i]);  
06         free(gr->arestas);  
07  
08         if(gr->eh_ponderado){  
09             for(i=0; i<gr->nro_vert; i++)  
10                 free(gr->pesos[i]);  
11             free(gr->pesos);  
12         }  
13         free(gr->grau);  
14         free(gr);  
15     }  
16 }
```

Inserindo uma aresta

```
01 int insereAresta(Grafo* gr,int orig,int dest,
                     int digrafo,float peso) {
02     if(gr == NULL)
03         return 0;
04     if(orig < 0 || orig >= gr->nro_vert)
05         return 0;
06     if(dest < 0 || dest >= gr->nro_vert)
07         return 0;
08
09     gr->arestas[orig][gr->grau[orig]] = dest;
10    if(gr->eh_ponderado)
11        gr->pesos[orig][gr->grau[orig]] = peso;
12    gr->grau[orig]++;
13
14    if(digrafo == 0)
15        insereAresta(gr,dest,orig,1,peso);
16    return 1;
17 }
```

Removendo uma aresta

```
01 int removeAresta(Grafo* gr, int orig, int dest,
                     int eh_digrafo){
02     if(gr == NULL)
03         return 0;
04     if(orig < 0 || orig >= gr->nro_vertices)
05         return 0;
06     if(dest < 0 || dest >= gr->nro_vertices)
07         return 0;
08
09     int i = 0;
10    while(i<gr->grau[orig] && gr->arestas[orig][i]!=dest)
11        i++;
12    if(i == gr->grau[orig])//elemento não encontrado
13        return 0;
14    gr->grau[orig]--;
15    gr->arestas[orig][i]=gr->arestas[orig][gr->grau[orig]];
16    if(gr->eh_ponderado)
17        gr->pesos[orig][i]=gr->pesos[orig][gr->grau[orig]];
18    if(eh_digrafo == 0)
19        removeAresta(gr,dest,orig,1);
20    return 1;
21 }
```

ALGORITMOS DE BUSCA

De modo geral, as operações de busca dependem do vértice inicial.

As operações de busca são utilizadas para resolver uma série de problemas em grafos.

Os principais tipos de busca são:

- **Busca em profundidade**
- **Busca em largura**
- **Busca pelo menor caminho**

CRIAÇÃO DO GRAFO

Arquivo Grafo.h

```
01 typedef struct grafo Grafo;
02 Grafo* cria_Grafo(int nro_vert,int Gmax,int eh_ponderado);
03 void libera_Grafo(Grafo* gr);
04 int insereAresta(Grafo* gr,int orig,int dest,
05                   int digrafo,float peso);
06 int removeAresta(Grafo* gr,int orig,int dest,int digrafo);
```

Arquivo Grafo.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "Grafo.h" //inclui os Protótipos
04 //Definição do tipo Grafo
05 struct grafo{
06     int eh_ponderado;
07     int nro_vert;
08     int Gmax;
09     int** arestas;
10     float** pesos;
11     int* grau;
12 };
```

Grafo para teste das buscas

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "Grafo.h"
04 int main(){
05     int eh_digrafo = 1;
06     Grafo* gr = cria_Grafo(5, 5, 0);
07     insereAresta(gr,0,1,eh_digrafo,0);
08     insereAresta(gr,1,3,eh_digrafo,0);
09     insereAresta(gr,1,2,eh_digrafo,0);
10     insereAresta(gr,2,4,eh_digrafo,0);
11     insereAresta(gr,3,0,eh_digrafo,0);
12     insereAresta(gr,3,4,eh_digrafo,0);
13     insereAresta(gr,4,1,eh_digrafo,0);

14
15     //realizar a busca aqui

16
17     libera_Grafo(gr);
18     system("pause");
19
20 }
```

Digrafo é um grafo direcionado

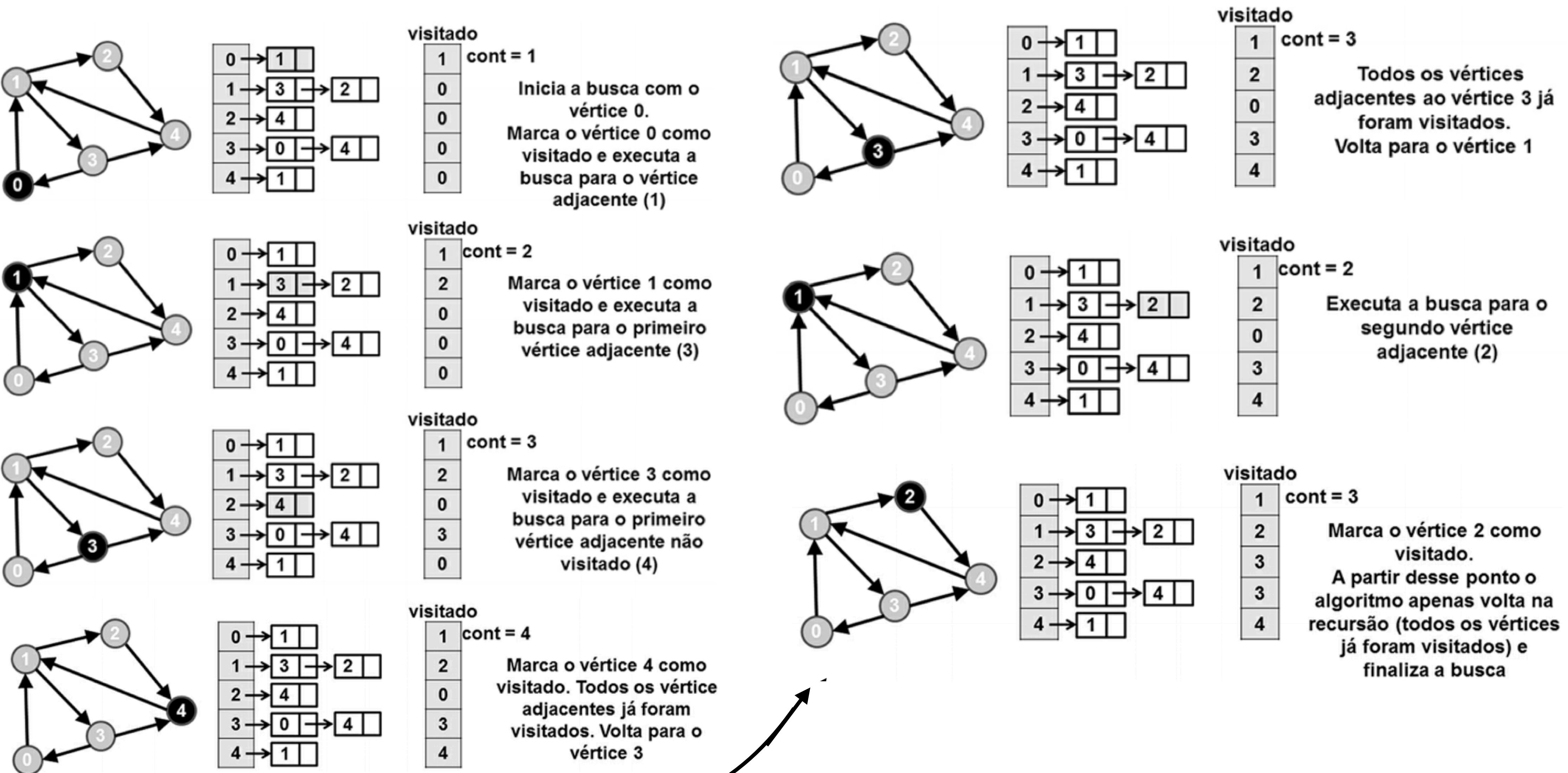
I) BUSCA EM PROFUNDIDADE

Partindo de um vértice inicial, a busca explora o máximo possível cada um dos vizinhos de um vértice antes de retroceder (backtracking).

Backtracking significa retornar pelo mesmo caminho percorrido com o objetivo de encontrar um caminho alternativo.

A busca termina quando o alvo da busca é encontrado ou um vértice sem vizinhos não-visitados é encontrado.

I) BUSCA EM PROFUNDIDADE



I) BUSCA EM PROFUNDIDADE

Busca em profundidade

```
01 void buscaProfundidade(Grafo *gr, int ini,
                           int *visitado, int cont){
02     int i;
03     visitado[ini] = cont;
04     for(i=0; i<gr->grau[ini]; i++){
05         if(!visitado[gr->arestas[ini][i]])
06             buscaProfundidade(gr, gr->arestas[ini][i],
07                                 visitado, cont+1);
08     }
09 }
10 void buscaProfundidade_Grafo(Grafo *gr, int ini,
11                               int *visitado){
12     int i, cont = 1;
13     for(i=0; i<gr->nro_vertices; i++)
14         visitado[i] = 0;
15     buscaProfundidade(gr, ini, visitado, cont);
16 }
```

A função **buscaProfundidade_Grafo** marca todos os vértices como não visitados (**linhas 11-12**) e, em seguida, chama a função **buscaProfundidade** para o vértice inicial, **ini**, com o contador de visitação, **cont**, inicializado em 1.

Já a função **buscaProfundidade** recebe como parâmetros o grafo (**gr**), o vértice inicial da busca (**ini**), o array (**visitado**) e o contador de visitação (**cont**).

A busca se inicia marcando o vértice inicial como visitado (**linha 3**).

Em seguida, para cada vizinho de **ini** (**linha 4**) a função verifica se ele já foi marcado como visitado (**linha 5**).

I) BUSCA EM PROFUNDIDADE

Busca em profundidade

```
01 void buscaProfundidade(Grafo *gr, int ini,
                           int *visitado, int cont){
02     int i;
03     visitado[ini] = cont;
04     for(i=0; i<gr->grau[ini]; i++){
05         if(!visitado[gr->arestas[ini][i]])
06             buscaProfundidade(gr, gr->arestas[ini][i],
07                                 visitado, cont+1);
08     }
09 }
10 void buscaProfundidade_Grafo(Grafo *gr, int ini,
11                               int *visitado){
12     int i, cont = 1;
13     for(i=0; i<gr->nro_vertices; i++)
14         visitado[i] = 0;
15     buscaProfundidade(gr, ini, visitado, cont);
16 }
```

Caso ele não tenha sido visitado, a função buscaProfundidade é novamente chamada, tendo esse vizinho como vértice inicial da busca e o contador de visitação incrementado em uma unidade (**linha 6**).

A busca termina quando não houver mais vizinhos a serem visitados.

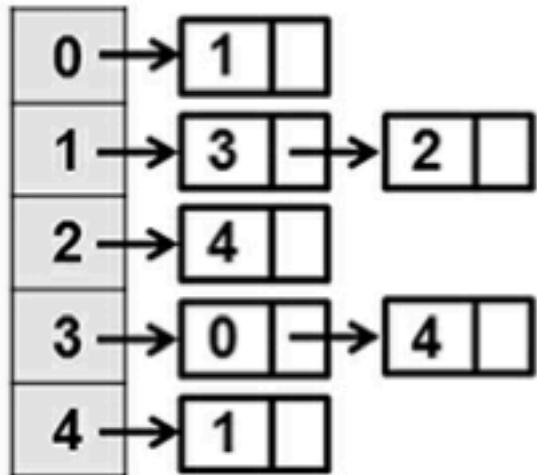
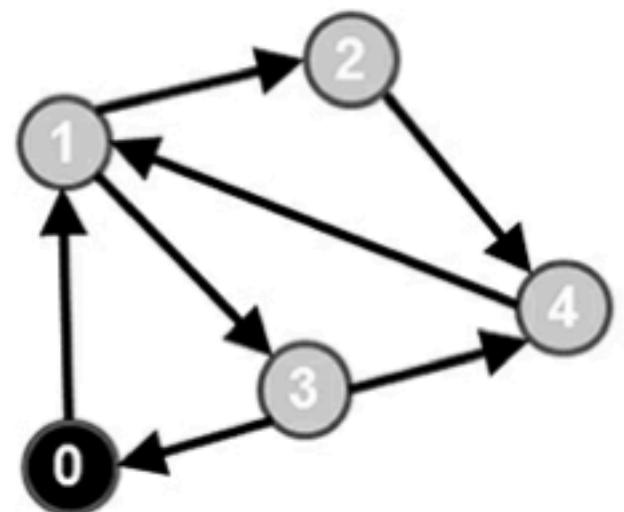
O array visitado contém agora a ordem em que cada vértice do grafo foi visitado, a partir do vértice inicial ini.

2) BUSCA EM LARGURA

Durante a busca, o grafo é percorrido de maneira sistemática: primeiro, ela marca como “visitados” todos os vizinhos de um vértice e, em seguida, começa a visitar os vizinhos de cada vértice na ordem em que eles foram marcados.

Para realizar essa tarefa, uma fila é utilizada para administrar a visitação dos vértices: o primeiro vértice marcado (ou marcado há mais tempo) é o primeiro a ser visitado.

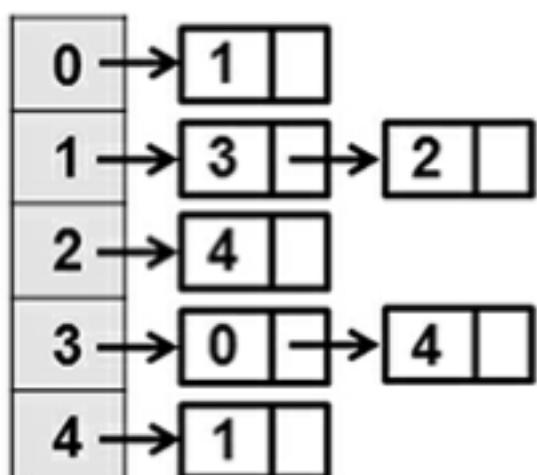
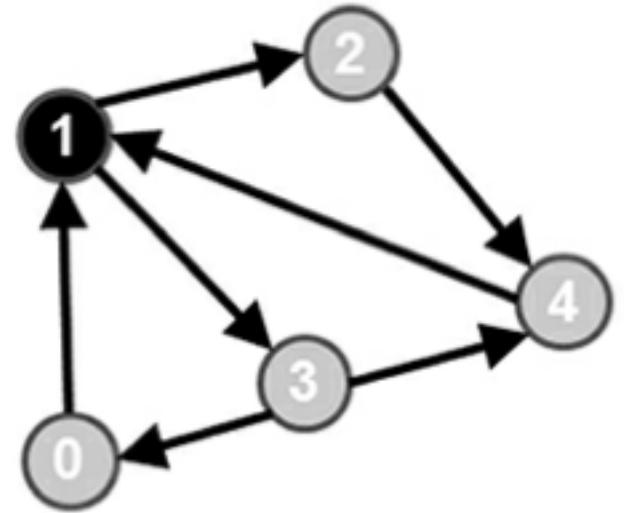
2) BUSCA EM LARGURA



visitado
0
1
0
0
0

fila 0

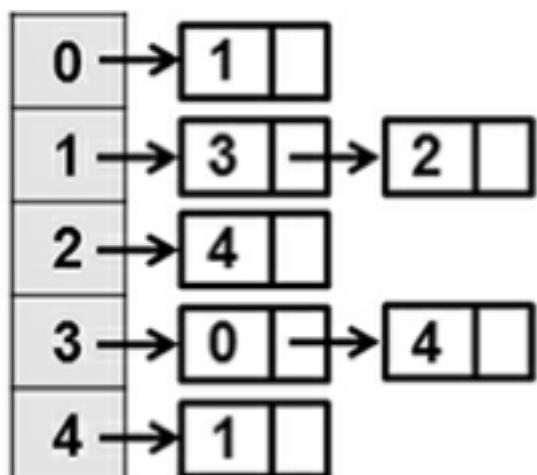
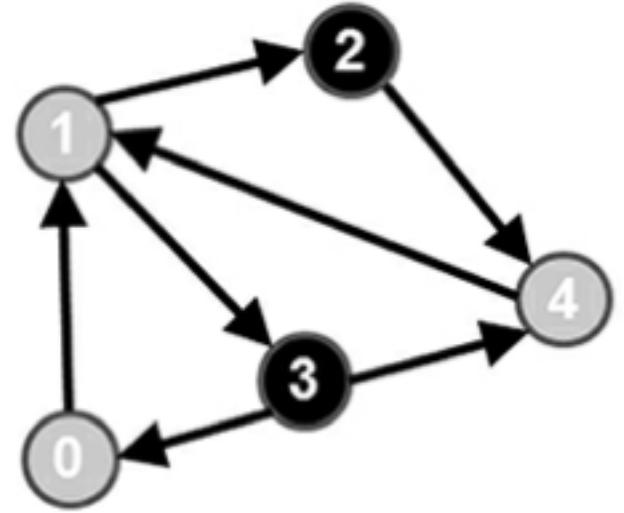
Inicializa a busca em largura.
Visita e insere na fila o vértice 0.



visitado
1
2
0
0
0

fila 1

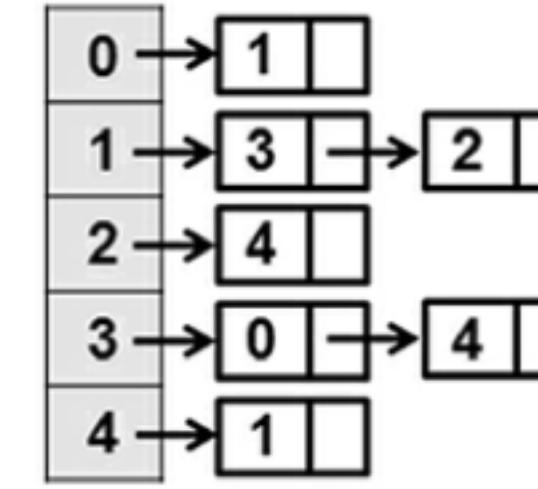
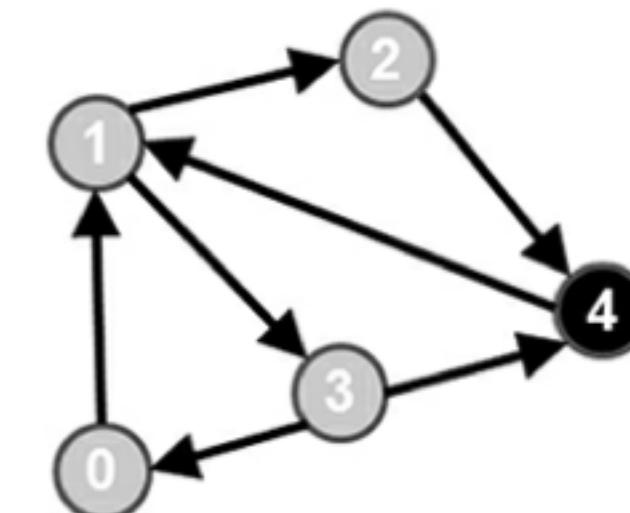
Remove vértice 0 da fila.
Insere na fila o vértice adjacente (1) que não foi visitado.



visitado
1
2
3
3
0

fila 3 2

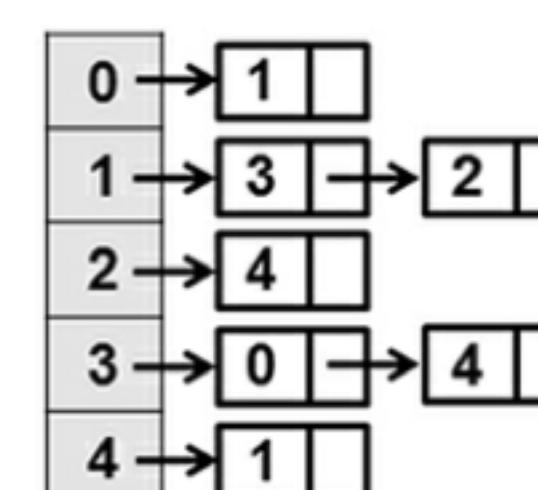
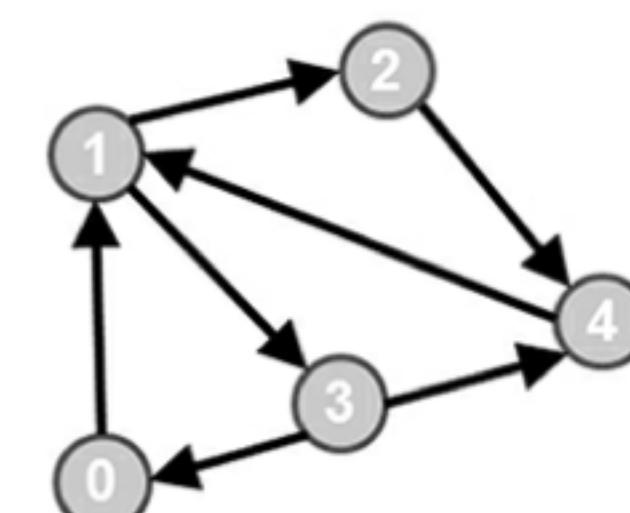
Remove vértice 1 da fila.
Insere na fila os vértices adjacentes (3 e 2) que não foram visitados.



visitado
1
2
3
3
4

fila 2 4

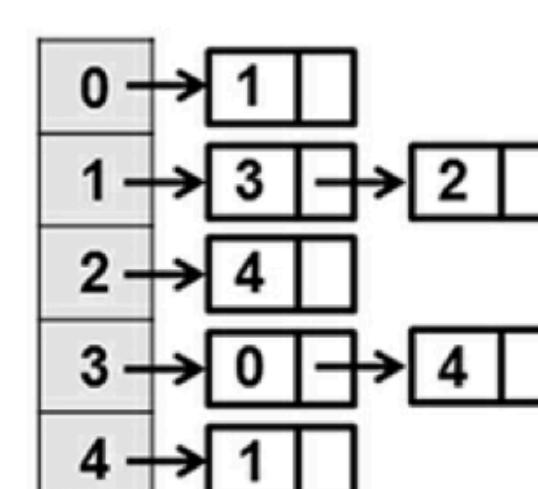
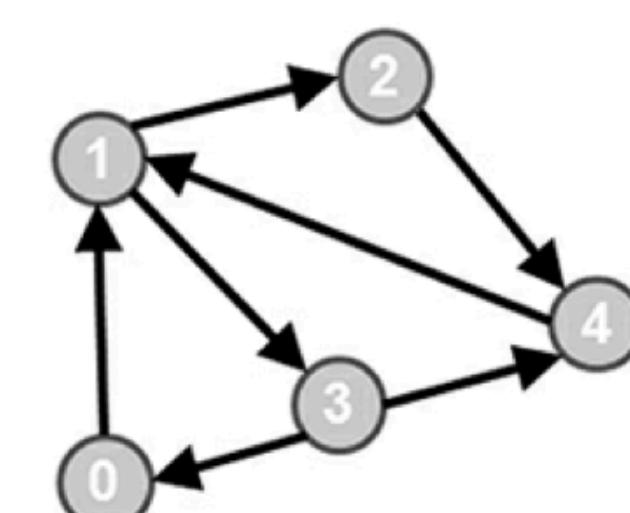
Remove vértice 3 da fila.
Insere na fila o vértice adjacente (4) que não foi visitado.



visitado
1
2
3
3
4

fila 4

Remove vértice 2 da fila.
Vértices adjacentes já foram visitados.



visitado
1
2
3
3
4

fila

Remove vértice 4 da fila.
Vértices adjacentes já foram visitados.

Fila vazia: fim da busca!

2) BUSCA EM LARGURA

Busca em largura

```
01 void buscaLargura_Grafo(Grafo *gr,int ini,int *visitado){  
02     int i, vert, NV, cont = 1;  
03     int *fila, IF = 0, FF = 0;  
04     for(i=0; i<gr->nro_vertices; i++)  
05         visitado[i] = 0;  
06  
07     NV = gr->nro_vertices;  
08     fila = (int*) malloc(NV * sizeof(int));  
09     FF++;  
10     fila[FF] = ini;  
11     visitado[ini] = cont;  
12     while(IF != FF){  
13         IF = (IF + 1) % NV;  
14         vert = fila[IF];  
15         cont++;  
16         for(i=0; i<gr->grau[vert]; i++){  
17             if(!visitado[gr->arestas[vert][i]]){  
18                 FF = (FF + 1) % NV;  
19                 fila[FF] = gr->arestas[vert][i];  
20                 visitado[gr->arestas[vert][i]] = cont;  
21             }  
22         }  
23     }  
24     free(fila);  
25 }
```

A função recebe três parâmetros: o grafo (**gr**), o vértice inicial da busca (**ini**) e um array (**visitado**), cujo tamanho é o número de vértices do grafo.

O array é usado para marcar a ordem em que cada vértice será visitado e esse será o resultado da nossa função.

Inicialmente, marcamos todos os vértices como não visitados (linhas 4-5).

Em seguida, criamos um array auxiliar fila (linhas 7-8). Este array é utilizado como sendo nossa fila estática, sendo **IF** e **FF** o início e o final da fila, respectivamente.

Depois, inserimos o vértice inicial no final da fila e o marcamos como visitado (linhas 9-11).

Tem então início a busca.

2) BUSCA EM LARGURA

Busca em largura

```
01 void buscaLargura_Grafo(Grafo *gr,int ini,int *visitado){  
02     int i, vert, NV, cont = 1;  
03     int *fila, IF = 0, FF = 0;  
04     for(i=0; i<gr->nro_vertices; i++)  
05         visitado[i] = 0;  
06  
07     NV = gr->nro_vertices;  
08     fila = (int*) malloc(NV * sizeof(int));  
09     FF++;  
10     fila[FF] = ini;  
11     visitado[ini] = cont;  
12     while(IF != FF){  
13         IF = (IF + 1) % NV;  
14         vert = fila[IF];  
15         cont++;  
16         for(i=0; i<gr->grau[vert]; i++){  
17             if(!visitado[gr->arestas[vert][i]]){  
18                 FF = (FF + 1) % NV;  
19                 fila[FF] = gr->arestas[vert][i];  
20                 visitado[gr->arestas[vert][i]] = cont;  
21             }  
22         }  
23     }  
24     free(fila);  
25 }
```

Enquanto houver vértices na fila (linha 12), o seguinte conjunto de passos será realizado:

- Remove-se um vértice da fila, vert (linhas 13-14).
- Incrementa-se o contador de visitação, cont (linha 15).
- Para cada vizinho de vert (linha 16):
 - Verifica-se se ele já foi marcado como visitado (linha 17).
 - Caso ele não tenha sido visitado, ele é inserido no final da fila e marcado como visitado (linhas 18-20).

2) BUSCA EM LARGURA

Busca em largura

```
01 void buscaLargura_Grafo(Grafo *gr,int ini,int *visitado){  
02     int i, vert, NV, cont = 1;  
03     int *fila, IF = 0, FF = 0;  
04     for(i=0; i<gr->nro_vertices; i++)  
05         visitado[i] = 0;  
06  
07     NV = gr->nro_vertices;  
08     fila = (int*) malloc(NV * sizeof(int));  
09     FF++;  
10     fila[FF] = ini;  
11     visitado[ini] = cont;  
12     while(IF != FF){  
13         IF = (IF + 1) % NV;  
14         vert = fila[IF];  
15         cont++;  
16         for(i=0; i<gr->grau[vert]; i++){  
17             if(!visitado[gr->arestas[vert][i]]){  
18                 FF = (FF + 1) % NV;  
19                 fila[FF] = gr->arestas[vert][i];  
20                 visitado[gr->arestas[vert][i]] = cont;  
21             }  
22         }  
23     }  
24     free(fila);  
25 }
```

Uma vez que a fila fique vazia (linha 12), a busca termina e a fila pode ser destruída (linha 24).

O array visitado contém agora a ordem em que cada vértice do grafo foi visitado, a partir do vértice inicial ini.

3) MENOR CAMINHO ENTRE DOIS VÉRTICES

O menor caminho entre dois vértices é a aresta que os conecta.

No entanto, é muito comum em um grafo não existir uma aresta conectando dois vértices v_1 e v_5 , isto é, eles não são adjacentes.

Apesar disso, dois vértices que não são adjacentes podem ser conectados por uma sequência de arestas, como (v_1, v_2) , (v_2, v_3) , (v_3, v_4) , (v_4, v_5) .

Caso essa seja a menor sequência de aresta que liga os dois vértices, dizemos que é o menor caminho ou caminho mais curto ou caminho geodésico entre eles.

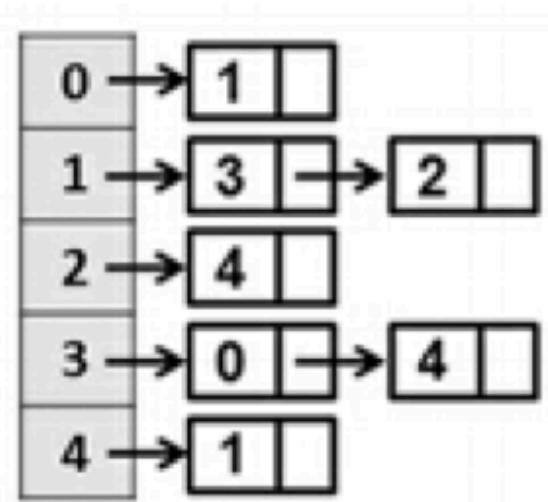
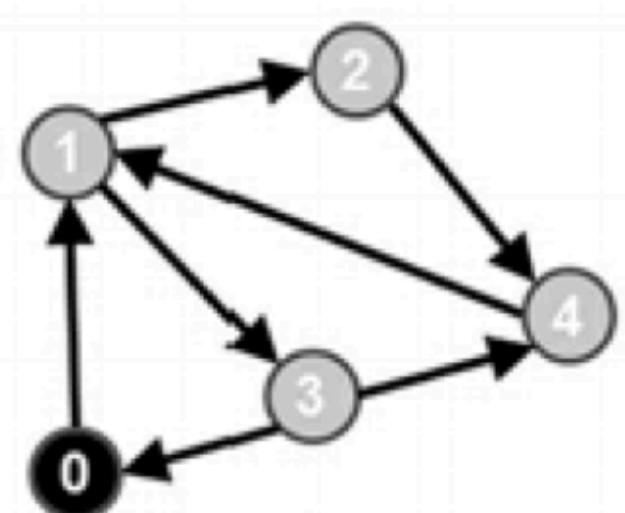
ALGORITMO DIJKSTRA

O algoritmo de Dijkstra é talvez o mais conhecido algoritmo para resolver esse problema.

Ele resolve o problema do menor caminho para grafos e digrafos, ponderados ou não.

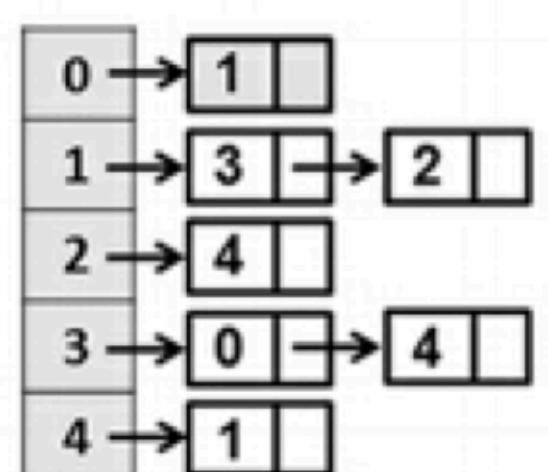
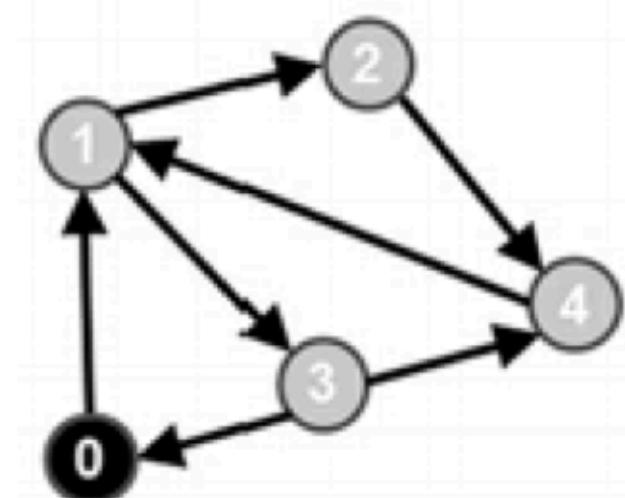
Sua limitação é que, no caso de um grafo ponderado, as arestas não podem ter pesos negativos.

3) MENOR CAMINHO ENTRE DOIS VÉRTICES



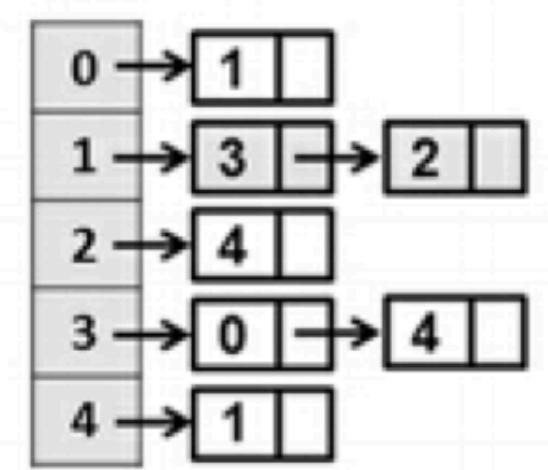
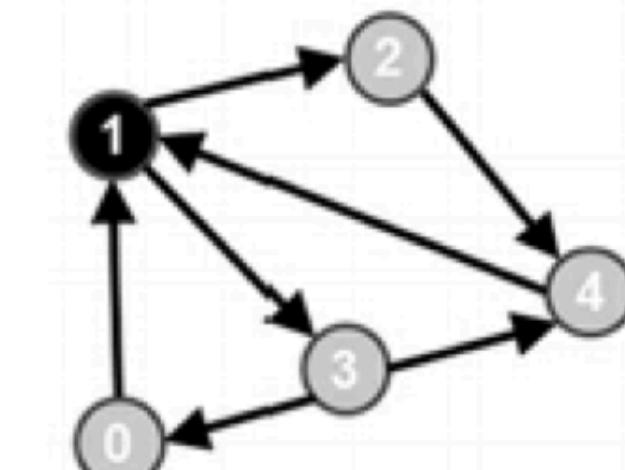
dist	ant	visitado
0	-1	0
1	-1	0
2	-1	0
3	-1	0
4	-1	0

Inicia o cálculo com o vértice 0. Atribui distância ZERO a ele (início). O restante dos vértices recebem distância -1



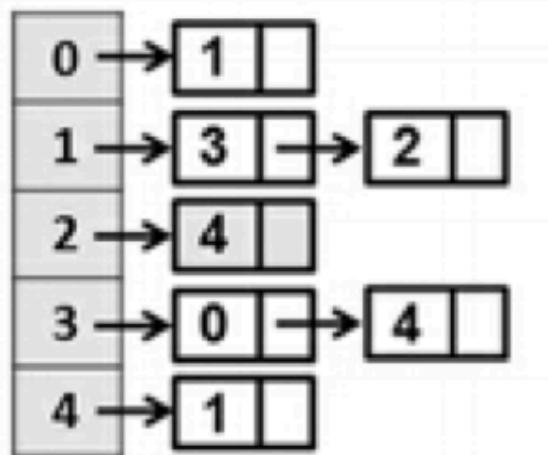
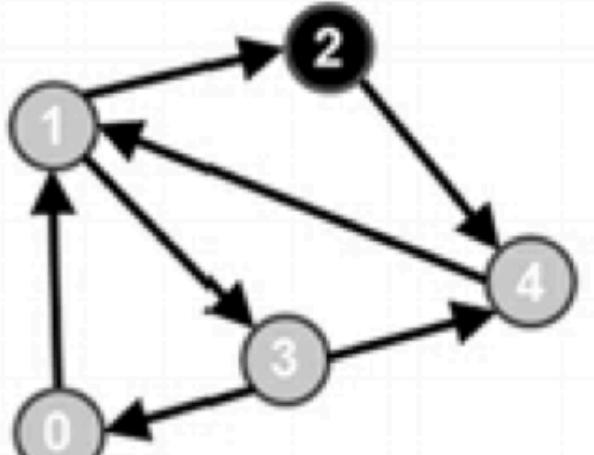
dist	ant	visitado
0	-1	1
1	0	0
2	-1	0
3	-1	0
4	-1	0

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 0. Verifica e atualiza (se necessário) dist e ant do vértice adjacente (1)



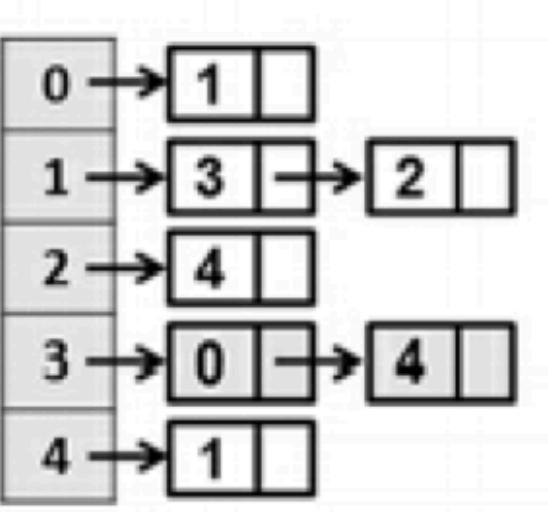
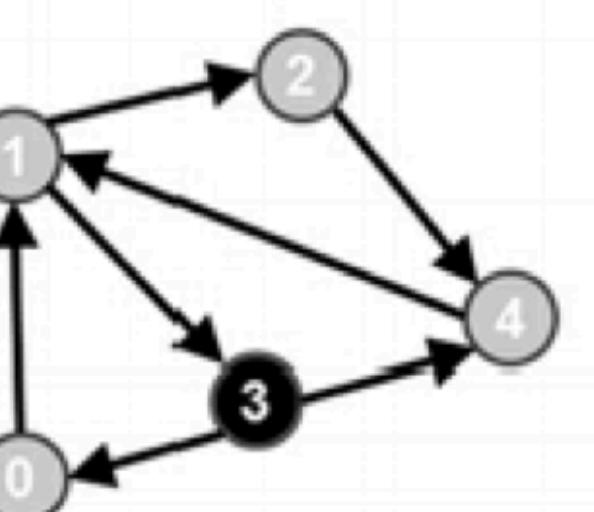
dist	ant	visitado
0	-1	1
1	0	1
2	1	0
3	1	0
4	-1	0

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 1. Verifica e atualiza (se necessário) dist e ant dos vértices adjacentes (2 e 3)



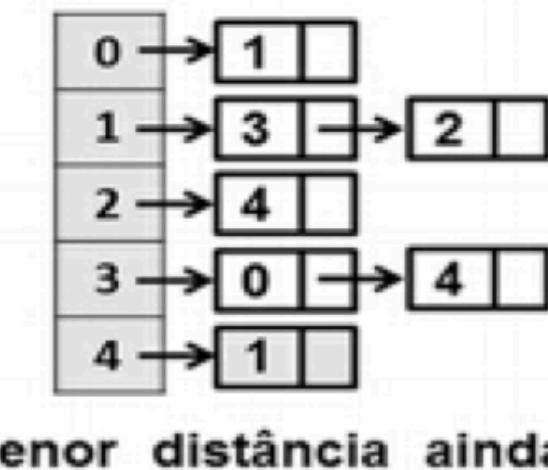
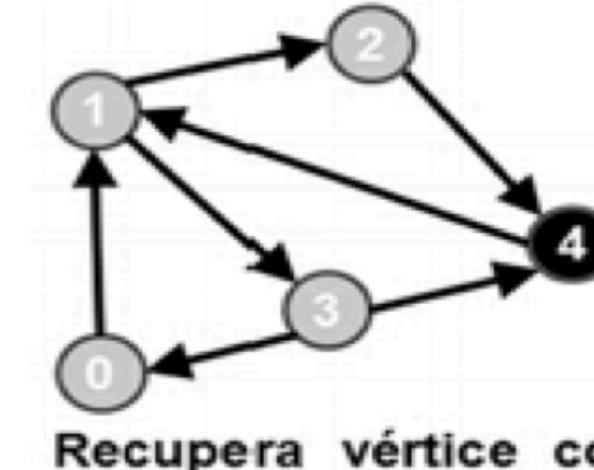
dist	ant	visitado
0	-1	1
1	0	1
2	1	1
3	1	0
4	2	0

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 2. Verifica e atualiza (se necessário) dist e ant do vértice adjacente (4)



dist	ant	visitado
0	-1	1
1	0	1
2	1	1
3	1	1
4	2	0

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 3. Verifica e atualiza (se necessário) dist e ant dos vértices adjacentes (0 e 4)



dist	ant	visitado
0	-1	1
1	0	1
2	1	1
3	2	1
4	3	0

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 4. Verifica e atualiza (se necessário) dist e ant do vértice adjacente (1)

Todos os vértices já foram visitados. Cálculo do menor caminho chegou ao fim.

Menor caminho entre dois vértices

```
01 int procuraMenorDistancia(float *dist, int *visitado,
                                int NV) {
02     int i, menor = -1, primeiro = 1;
03     for(i=0; i < NV; i++) {
04         if(dist[i] >= 0 && visitado[i] == 0) {
05             if(primeiro) {
06                 menor = i;
07                 primeiro = 0;
08             }else{
09                 if(dist[menor] > dist[i])
10                     menor = i;
11             }
12         }
13     }
14     return menor;
15 }
```

A função `procuraMenorDistancia` recebe três parâmetros: o array de distâncias (**dist**), o array de vértices visitados (**visitado**) e o número de vértices do grafo (**NV**).

Basicamente, a função testa todos os vértices (linha 3) à procura daquele que tiver a menor distância não negativa e que ainda não tenha sido visitado (linhas 4-12).

O índice do vértice que satisfaçõe essas condições é o retorno da função.

```

16 void menorCaminho_Grafo(Grafo *gr, int ini,
17     int *ant, float *dist){
18     int i, cont, NV, ind, *visitado, vert;
19     cont = NV = gr->nro_vertices;
20     visitado = (int*) malloc(NV * sizeof(int));
21     for(i=0; i < NV; i++){
22         ant[i] = -1;
23         dist[i] = -1;
24         visitado[i] = 0;
25     }
26     dist[ini] = 0;
27     while(cont > 0){
28         vert = procuraMenorDistancia(dist, visitado, NV);
29         if(vert == -1)
30             break;
31         visitado[vert] = 1;
32         cont--;
33         for(i=0; i<gr->grau[vert]; i++){
34             ind = gr->arestas[vert][i];
35             if(dist[ind] < 0){
36                 dist[ind] = dist[vert] + 1;
37                 ant[ind] = vert;
38             }else{
39                 if(dist[ind] > dist[vert] + 1){
40                     dist[ind] = dist[vert] + 1;
41                     ant[ind] = vert;
42                 }
43             }
44         }
45     }
46     free(visitado);
47 }
```

Já a função menorCaminho_Grafo recebe como parâmetros o grafo (**gr**), o vértice inicial (**ini**) e dois arrays: **ant**, para armazenar o antecessor do vértice dentro do caminho, e **dist**, que armazenará a distância do vértice inicial até ele.

Esses arrays têm tamanho igual ao número de vértices do grafo.

Inicialmente, criamos uma variável para guardar o número de vértices que faltam ser visitados, **cont**, e um array auxiliar **visitado** para gerenciar os vértices que ainda precisam ser visitados (linhas 18-19).

Em seguida, marcamos todos os vértices como não possuindo um antecessor (-1), distância inválida (-1) e não visitados (0) (linhas 20-24).

Por fim, marcamos que a distância até o vértice inicial é 0 (linha 25).

Tem então início a busca.

```

16 void menorCaminho_Grafo(Grafo *gr, int ini,
17     int *ant, float *dist){
18     int i, cont, NV, ind, *visitado, vert;
19     cont = NV = gr->nro_vertices;
20     visitado = (int*) malloc(NV * sizeof(int));
21     for(i=0; i < NV; i++){
22         ant[i] = -1;
23         dist[i] = -1;
24         visitado[i] = 0;
25     }
26     dist[ini] = 0;
27     while(cont > 0){
28         vert = procuraMenorDistancia(dist, visitado, NV);
29         if(vert == -1)
30             break;
31         visitado[vert] = 1;
32         cont--;
33         for(i=0; i<gr->grau[vert]; i++){
34             ind = gr->arestas[vert][i];
35             if(dist[ind] < 0){
36                 dist[ind] = dist[vert] + 1;
37                 ant[ind] = vert;
38             }else{
39                 if(dist[ind] > dist[vert] + 1){
40                     dist[ind] = dist[vert] + 1;
41                     ant[ind] = vert;
42                 }
43             }
44         }
45     }
46     free(visitado);
47 }
```

Enquanto houver vértices a visitar (linha 26), o seguinte conjunto de passos será realizado:

- Use a função procuraMenorDistancia para achar o vértice com a menor distância e que ainda não foi visitado, vert (linha 27). Caso o índice do vértice seja inválido (-1), a busca termina (linhas 28-29).
- Marque esse vértice como visitado e diminua o número de vértices a serem visitados (linhas 31-32).
- Para cada vizinho de vert (linha 33): Se a distância até ele for negativa (linha 35), a distância passa a ser a distância de vert mais uma unidade (ou seja, mais uma aresta) e vert se torna o antecessor desse vértice no caminho (linhas 36-37).

```

16 void menorCaminho_Grafo(Grafo *gr, int ini,
17     int *ant, float *dist){
18     int i, cont, NV, ind, *visitado, vert;
19     cont = NV = gr->nro_vertices;
20     visitado = (int*) malloc(NV * sizeof(int));
21     for(i=0; i < NV; i++){
22         ant[i] = -1;
23         dist[i] = -1;
24         visitado[i] = 0;
25     }
26     dist[ini] = 0;
27     while(cont > 0){
28         vert = procuraMenorDistancia(dist, visitado, NV);
29         if(vert == -1)
30             break;
31         visitado[vert] = 1;
32         cont--;
33         for(i=0; i<gr->grau[vert]; i++){
34             ind = gr->arestas[vert][i];
35             if(dist[ind] < 0){
36                 dist[ind] = dist[vert] + 1;
37                 ant[ind] = vert;
38             }else{
39                 if(dist[ind] > dist[vert] + 1){
40                     dist[ind] = dist[vert] + 1;
41                     ant[ind] = vert;
42                 }
43             }
44         }
45     }
46     free(visitado);
47 }
```

Se a distância até ele for positiva (linha 38), verifique se a distância de **vert** mais uma unidade é menor do que a distância atual (linha 39).

Em caso afirmativo, essa passa a ser a nova distância até o vértice e **vert** se torna o antecessor desse vértice no caminho (linhas 40-41).

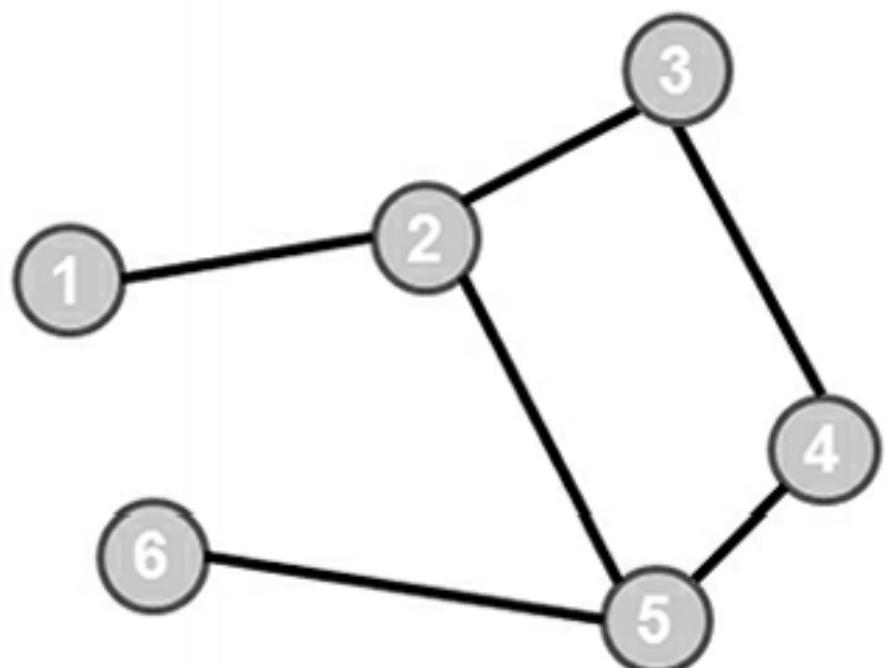
Uma vez que não existam mais vértices a ser visitados (linha 26), a busca termina e o array auxiliar pode ser destruído (linha 46).

Os arrays **ant** e **dist** contêm agora os antecessores e a distância de cada vértice do grafo no caminho traçado a partir do vértice inicial **ini**.

Esse processo é mais bem ilustrado pela Figura 10.34, que considera o vértice 0 como o vértice inicial.

TAREFAS

- 1) Escreva uma função para obter todos os nós adjacentes (vizinhos) de um nó do grafo. Considere que o grafo é representado por uma matriz de adjacências.
- 2) Escreva um algoritmo para verificar se um grafo é acíclico. Para isso, use o algoritmo de busca em profundidade.
- 3) Dado o grafo, mostre o resultado da busca em largura e em profundidade. Considere o vértice 1 o início da busca. Confira se está correto usando os códigos em linguagem c.





OBRIGADO!

RAFAELVC2@GMAIL.COM