



# Estrutura de Dados 2

Aula 07 parte 2 – Buscas em Grafos por Profundidade, por Largura e Menor Caminho

Antonio Angelo de Souza Tartaglia  
angelot@ifsp.edu.br



- Consiste em explorar um Grafo;

Explorar as conexões,  
os vértices, etc.

- Processo sistemático de como caminhar por seus vértices e arestas;
- Depende do vértice inicial;



- Vários problemas em Grafos podem ser resolvidos efetuando uma busca;
- A operação de busca pode precisar visitar todos os vértices. Para outros problemas, apenas um subconjunto de vértices precisa ser visitado.

Em um mapa, se procuramos o melhor caminho para ir de um lugar ao outro, por exemplo do IFSP ao Adamastor, não é necessário procurar caminhos nos vértices que representam a cidade de São Paulo.



- Principais tipos de busca em Grafos:

Existem vários outros tipos de buscas

- Busca em Profundidade;
- Busca em Largura;
- Busca pelo menor caminho.

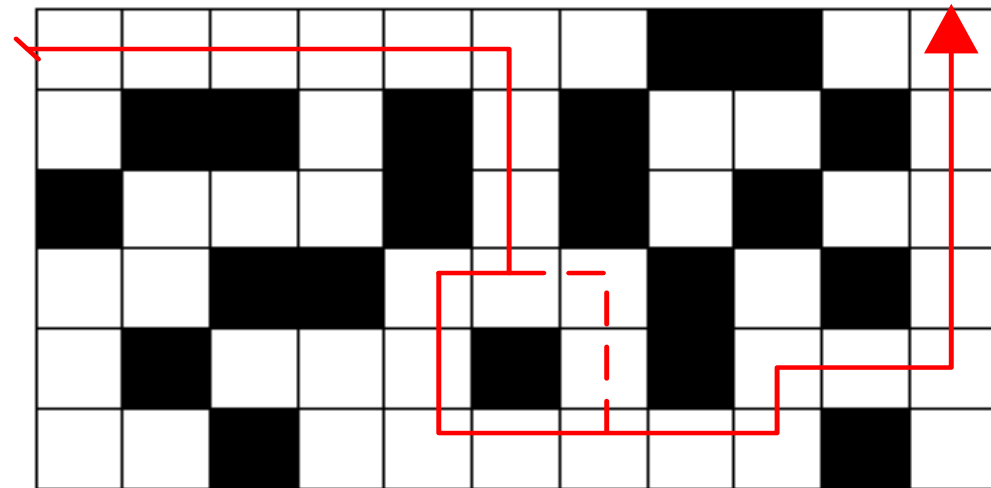
Traçado de rotas

## Buscas em Grafos

- Busca em profundidade:
- Partindo de um vértice inicial, ela explora o máximo possível cada um dos seus ramos antes de retroceder (Backtracking);
- Pode ser usado para :
  - Encontrar componentes conectados e fortemente conectados;
  - Ordenação Topológica de um Grafo;
  - Resolver quebra-cabeças (Ex. Labirinto).

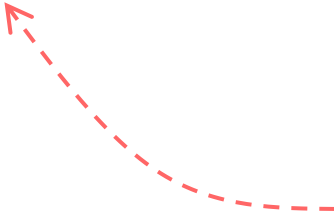
Encontrar um  
caminho dentro de  
um labirinto

Partes que não se  
ligam



## Buscas em Grafos

- Busca em Largura:
- Partindo de um vértice inicial, ela explora todos os vértices vizinhos, em seguida, para cada vértice vizinho, ela repete o processo, visitando os vértices ainda inexplorados;
- Pode ser usada para:
  - Achar componentes conectados;
  - Achar todos os vértices conectados a apenas 1 componente;
  - Achar o menor caminho entre dois vértices;
  - Testar bipartição em Grafos.



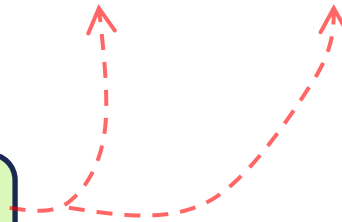
Dois conjuntos de vértices que não se conectam entre si, apenas um com o outro





- Busca pelo menor caminho:
- Partindo de um vértice inicial, calcula a menor distância desse vértice a todos os demais, desde que exista um caminho entre eles;
  - **Algoritmo de Dijkstra**
  - Resolve este problema para os Grafos **dirigido** ou **não dirigido**, com arestas de peso não negativo.

Dígrafos ou Grafos



## Buscas em Grafos

- Busca em Profundidade:
- Partindo de um vértice inicial, ela explora o máximo possível cada um dos seus ramos antes de retroceder (*Backtracking*);
- Pode ser usada para :
  - Encontrar componentes conectados e fortemente conectados;
  - Ordenação Topológica de um Grafo
  - Resolver Quebra-Cabeças (Ex. Labirinto)

Vetor com o mesmo tamanho que o número de vértices, onde será marcada a ordem de visita dos vértices.  
Este Vetor será a resposta para a Busca.

```
//No arquivo grafo.h  
void buscaProfundidade_Grafo(Grafo *gr, int ini, int *visitado);
```





## Buscas em Grafos



```
//No programa principal
int i, eh_digrafo = 1;
Grafo *gr = cria_Grafo(5, 5, 0);
insere_Aresta(gr, 0, 1, eh_digrafo, 0);
insere_Aresta(gr, 1, 3, eh_digrafo, 0);
insere_Aresta(gr, 1, 2, eh_digrafo, 0);
insere_Aresta(gr, 2, 4, eh_digrafo, 0);
insere_Aresta(gr, 3, 0, eh_digrafo, 0);
insere_Aresta(gr, 3, 4, eh_digrafo, 0);
insere_Aresta(gr, 4, 1, eh_digrafo, 0);
int vis[5];
```

Vetor de  
resposta com o  
mesmo número  
de vértices.

5 vértices com  
5 arestas

```
buscaProfundidade_Grafo(gr, 0, vis);
```

Partindo do  
vértice inicial



Marca vértices  
como não visitados

```
//No arquivo Grafo.c
//Função principal de interface com o usuário
void buscaProfundidade_Grafo(Grafo *gr, int ini, int *visitado){
    int i, cont = 1;
    for(i = 0; i < gr->nro_vertices; i++){
        > visitado[i] = 0;
    }
    buscaProfundidade(gr, ini, visitado, cont);
}
```

A interface garante que o  
vetor visitado seja  
inicializado, para que não  
fique a cargo do usuário  
sua inicialização

A recursão explora  
todos os ramos de um  
vizinho antes de voltar  
para o vértice original

# Estrutura de Dados 2

## Buscas em Grafos



```
//No arquivo Grafo.c  
//Função auxiliar para realizar o cálculo
```

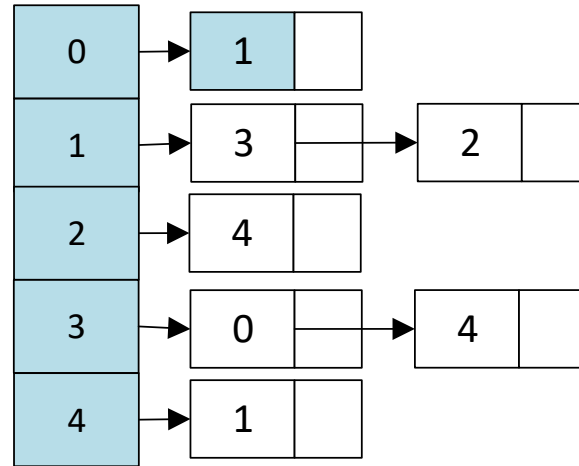
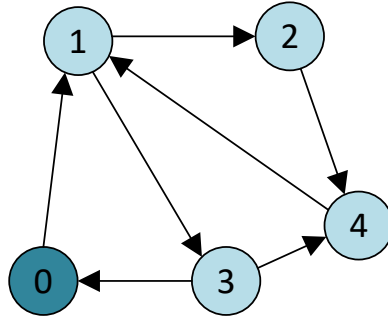
```
void buscaProfundidade(Grafo *gr, int ini, int *visitado, int cont){  
    int i;  
    visitado[ini] = cont;  
    for(i = 0; i < gr->grau[ini]; i++){  
        if(!visitado[gr->arestas[ini][i]]){  
            buscaProfundidade(gr, gr->arestas[ini][i], visitado, cont + 1);  
        }  
    }  
}
```

Marca o vértice inicial como visitado.  
0 = não visitado

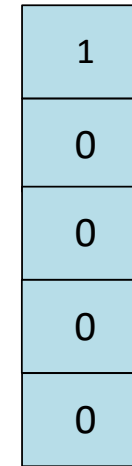
if: Verifica se o vizinho na posição [ini][i], foi visitado ou não. Se não foi, executa recursivamente a busca em profundidade partindo dele; é incrementando o contador.

for: percorre todas as arestas que partem deste vértice, ou seja, todos os vizinhos.  
Marca o vértice como visitado e visita os vizinhos ainda não visitados.

## Buscas em Grafos



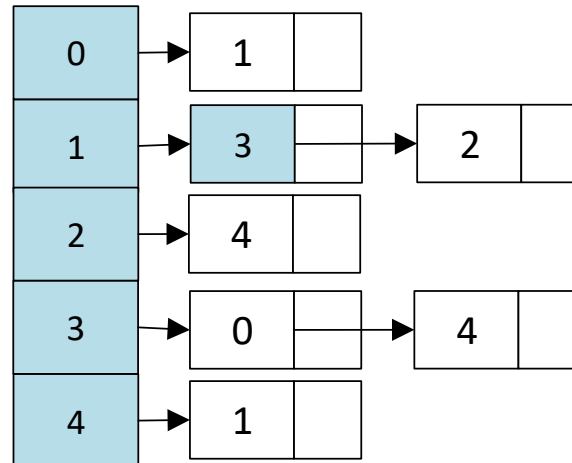
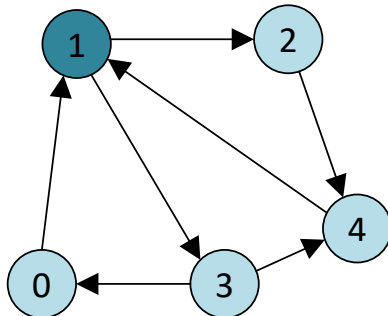
visitado



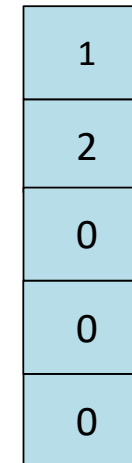
cont = 1

Inicia a busca com o vértice 0.

Marca o vértice 0 como visitado e executa a busca para o vértice adjacente (1)



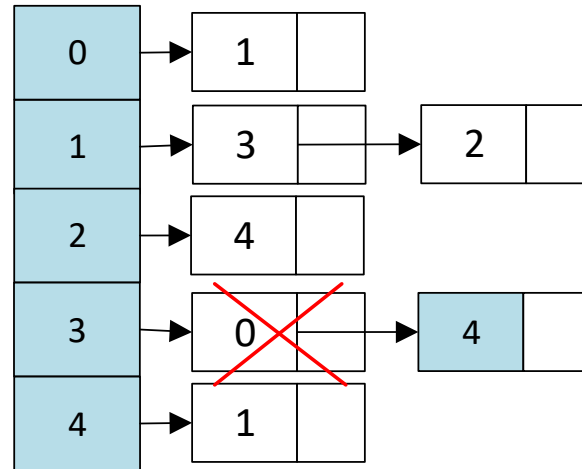
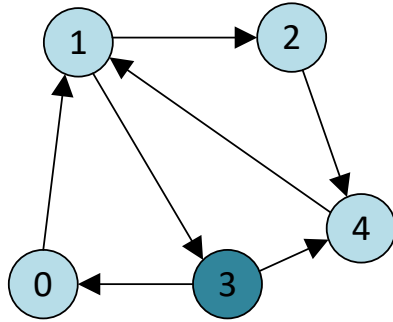
visitado



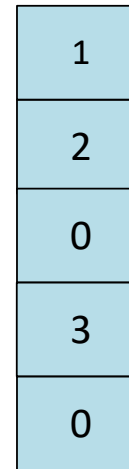
cont = 2

Marca o vértice 1 como visitado e executa a busca para o primeiro vértice adjacente (3).

## Buscas em Grafos

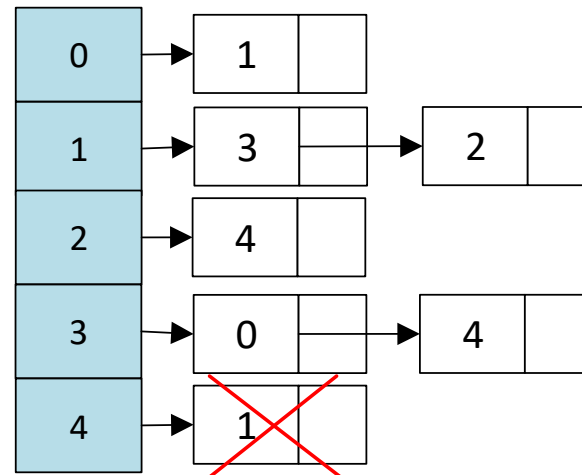
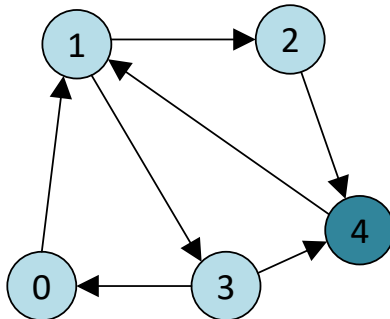


visitado

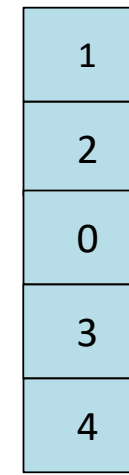


cont = 3

Marca o vértice 3 como visitado e executa a busca para o primeiro vértice adjacente não visitado (4)



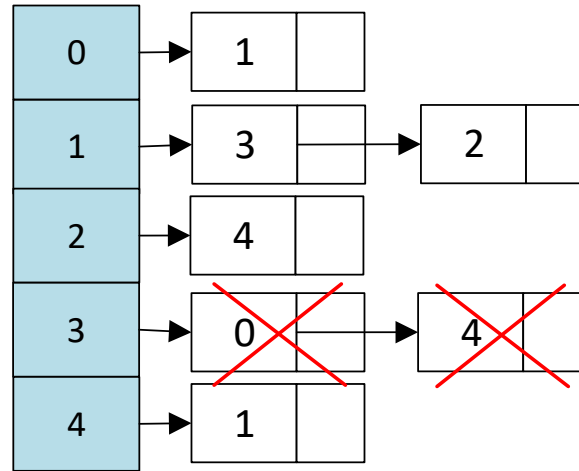
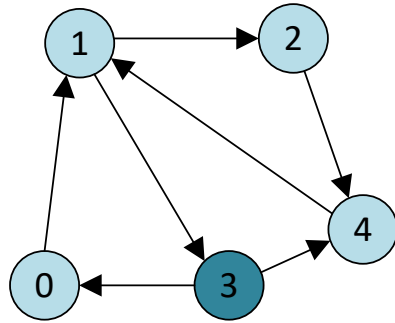
visitado



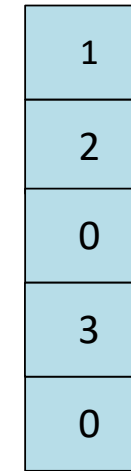
cont = 4

Marca o vértice 4 como visitado. Todos os vértices adjacentes já foram visitados. Volta para o vértice 3.

## Buscas em Grafos

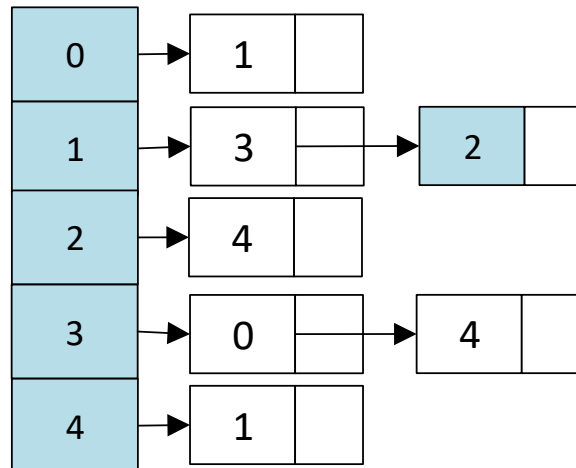
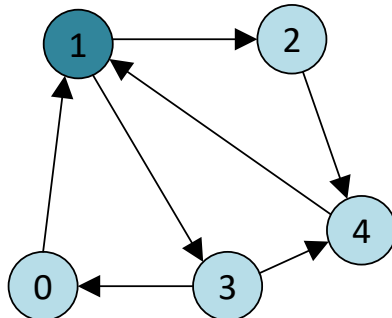


visitado

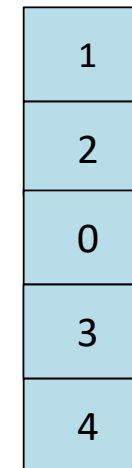


cont = 3

Todos os vértices adjacentes  
ao vértice 3 já foram visitados  
volta para o vértice 1



visitado

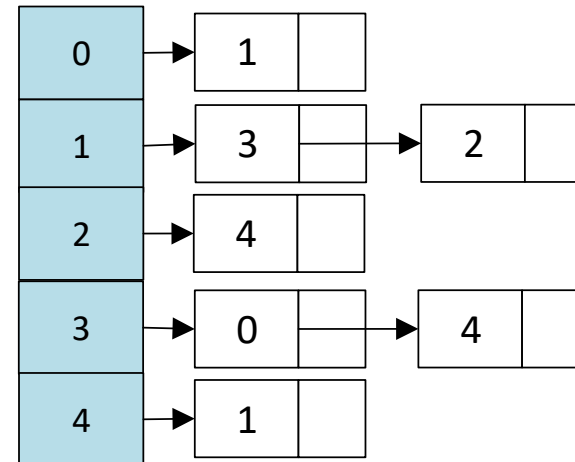
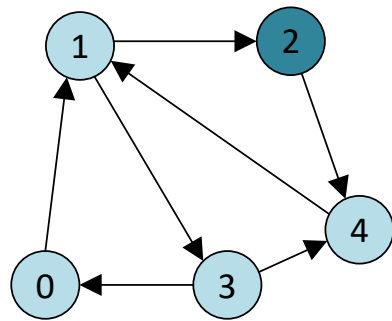


cont = 2

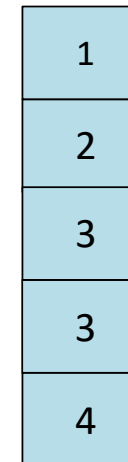
Executa a Busca para o  
segundo vértice adjacente (2)

# Estrutura de Dados 2

## Buscas em Grafos



visitado



cont = 3

Marca o vértice 2 como visitado a partir desse ponto a algoritmo apenas volta na recursão, já que todos os vértices foram visitados, e finaliza a busca.

A recursão então volta e verifica que todos os vértices foram visitados

Vetor final que será devolvido ao usuário, ficará com a ordem em que cada vértice foi atingido, ou seja, a sua profundidade.



## Buscas em Grafos

- Busca em Largura:
- Partindo de um vértice inicial, ela explora todos os vértices vizinhos. Em seguida, para cada vértice vizinho, ela repete este processo, visitando os vértices ainda inexplorados
- Pode ser usada para:
  - Achar componentes conectados;
  - Achar todos os vértices conectados a apenas um componente;
  - Testar bipartição em Grafos.

```
//No arquivo grafo.h
```

```
void buscaLargura_Grafo(Grafo *gr, int ini, int *visitado);
```

Devolve a ordem  
de visitação







## Buscas em Grafos

```
//No programa principal
int i, eh_digrafo = 1;
Grafo *gr = cria_Grafo(5, 5, 0);
insere_Aresta(gr, 0, 1, eh_digrafo, 0);
insere_Aresta(gr, 1, 3, eh_digrafo, 0);
insere_Aresta(gr, 1, 2, eh_digrafo, 0);
insere_Aresta(gr, 2, 4, eh_digrafo, 0);
insere_Aresta(gr, 3, 0, eh_digrafo, 0);
insere_Aresta(gr, 3, 4, eh_digrafo, 0);
insere_Aresta(gr, 4, 1, eh_digrafo, 0);
int vis[5];

buscaProfundidade_Grafo(gr, 0, vis);
printf("Resultado da busca em profundidade: ");
for(i = 0; i < 5; i++){
    printf("%d ", vis[i]);
}
printf("\n\n");

buscaLargura_Grafo(gr, 0, vis);
printf("Resultado da busca em largura: ");
for(i = 0; i < 5; i++){
    printf("%d ", vis[i]);
}
```

Vértice inicial



## Buscas em Grafos

```
//No arquivo Grafo.c
void buscaLargura_Grafo(Grafo *gr, int ini, int *visitado){
    int i, vert, NV, cont = 1, *fila, IF = 0, FF = 0;
    for(i = 0; i < gr->nro_vertices; i++){
        visitado[i] = 0;
    }
    NV = gr->nro_vertices;
    fila = (int*) malloc(NV * sizeof(int));
    FF++;
    fila[FF] = ini;
    visitado[ini] = cont;
    while(IF != FF){
        IF = (IF + 1) % NV;
        vert = fila[IF];
        cont++;
        for(i = 0; i < gr->grau[vert]; i++){
            if(!visitado[gr->arestas[vert][i]]){
                FF = (FF + 1) % NV;
                fila[FF] = gr->arestas[vert][i];
                visitado[gr->arestas[vert][i]] = cont;
            }
        }
    }
    free(fila);
}
```

Cria fila visita  
e insere  
"ini" na fila

Enquanto a fila  
não estiver  
vazia...

Visita os vizinhos  
ainda não  
visitados e os  
coloca na fila

IF e FF: inicio e  
final da fila

Marca vértices como  
não visitados

Marca vértice inicial como visitado

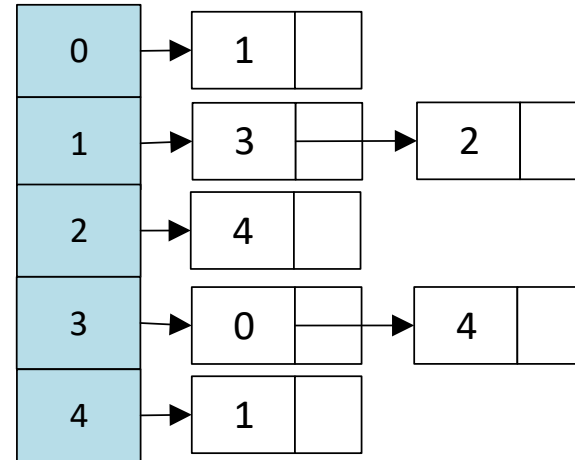
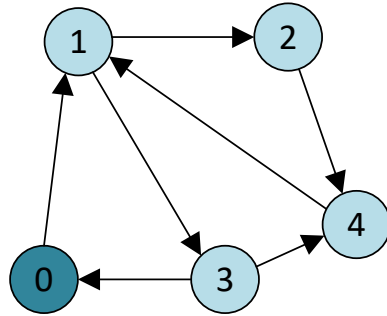
Recupera o primeiro da fila

A fila guarda a ordem  
em que os vértices  
foram visitados

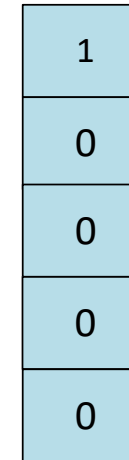
if: verifica se foi visitado,  
se não foi é colocado na fila

Acessa o primeiro da fila, marca todos  
que estão conectados a ele como  
visitados e repete o processo

## Buscas em Grafos

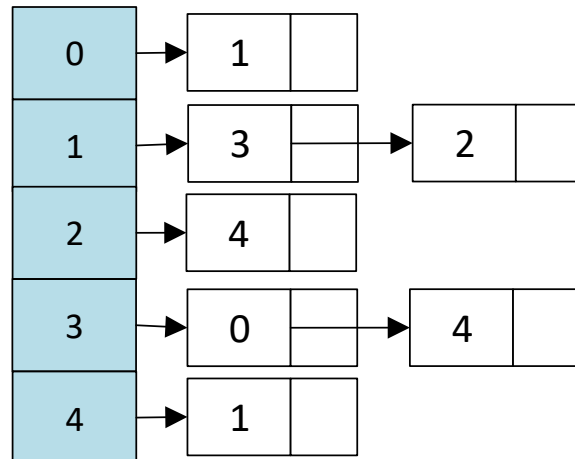
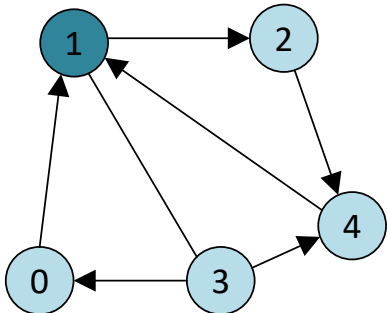


visitado

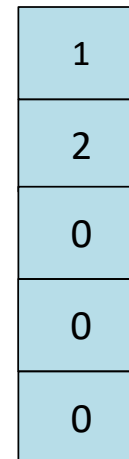


Fila 0

Inicializa a busca em largura,  
visita e insere na fila o vértice 0



visitado



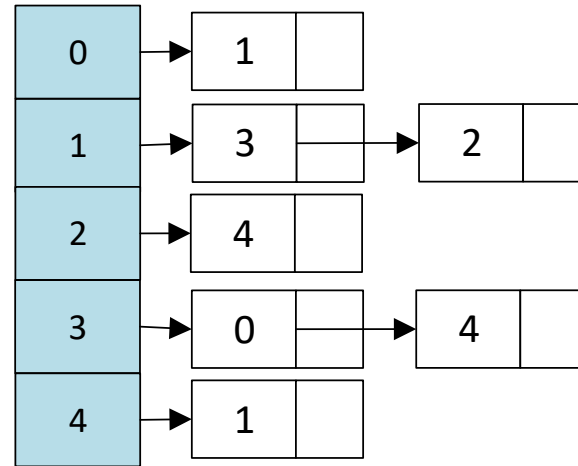
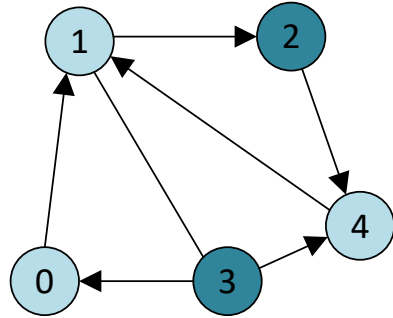
Fila 1

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

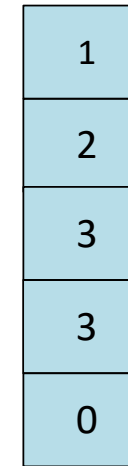
# Estrutura de Dados 2



## Buscas em Grafos



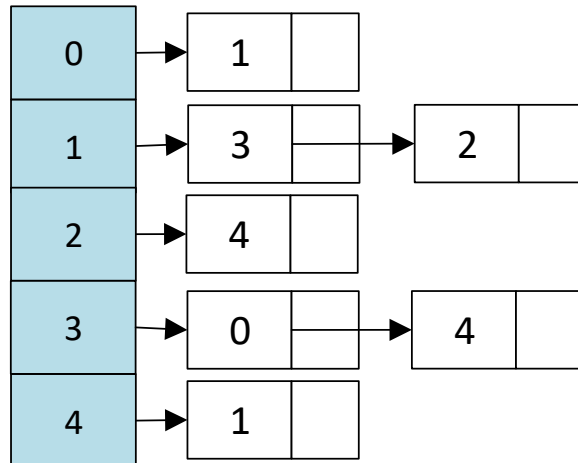
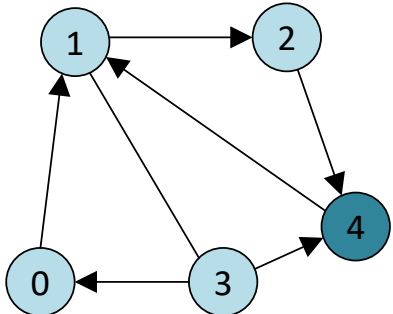
visitado



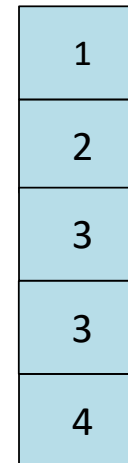
Fila 

3	2
---	---

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



visitado



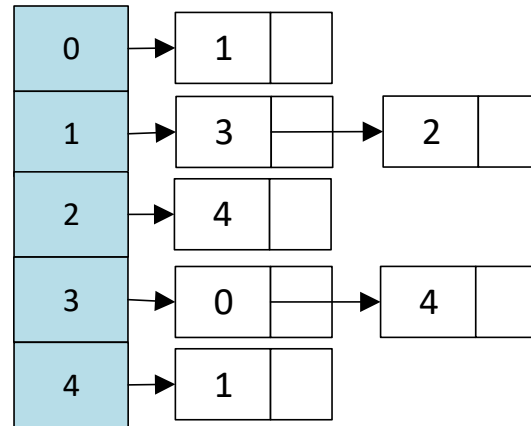
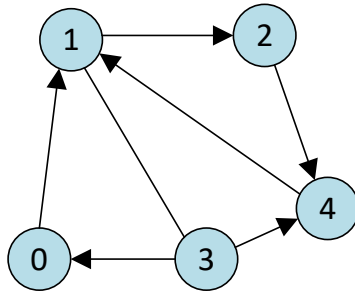
Fila 

2	4
---	---

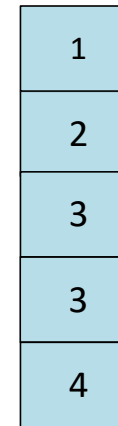
Remove o vértice 3 da fila.  
Insere na fila o vértice adjacente (4), que não foi visitado.

## Buscas em Grafos

- Todos já foram visitados, o processo agora é esvaziar a fila e verificar se faltou visitar algum vértice:



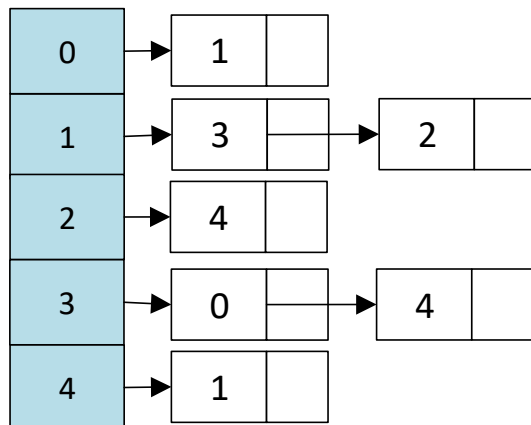
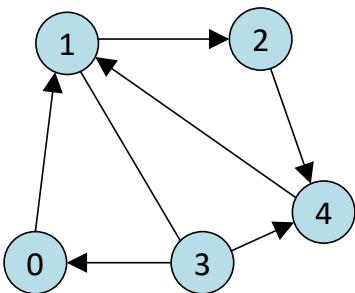
visitado



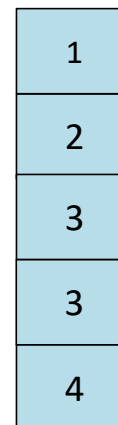
Fila 4

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

Ordem de visitação da  
Busca em Largura,  
partindo do vértice 0.



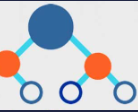
visitado



Fila

Remove vértice 4 da fila.  
Vértices adjacentes já foram visitados

Fila vazia: fim da Busca.



- Busca pelo menor caminho:
- Partindo de um vértice inicial, calcula a menor distância desse vértice a todos os demais, desde que exista um caminho entre eles;
  - **Algoritmo de Dijkstra**
  - Resolve este problema para os Grafos **dirigido** ou **não dirigido**, com arestas de peso não negativo.

```
//No arquivo grafo.h  
void menorCaminho_Grafo(Grafo *gr, int ini, int *ant, float *dist);
```

Dois vetores auxiliares com o mesmo tamanho que a quantidade de vértices no Grafo.

dist – distancia do vértice inicial até o vértice atual (sendo visitado).  
ant – acumula os vértices anteriores ao atual (ordem necessária de visitação para chegar ao destino)



Mesmo  
número de  
vértices do  
Grafo.

```
//no arquivo main()
```

```
{ int ant[5];
```

```
float dist[5];
```

```
menorCaminho_Grafo(gr, 0, ant, dist);
```

```
printf("Resultado da busca menor Caminho (distancia): ");
```

```
for(i = 0; i < 5; i++){
```

```
    printf("%.2f ", dist[i]);
```

```
}
```

```
printf("Resultado da busca menor Caminho (vetor de anteriores): ");
```

```
for(i = 0; i < 5; i++){
```

```
    printf("%d ", ant[i]);
```

```
}
```

Vértice inicial

## Buscas em Grafos

Verifica se é o primeiro vértice encontrado nessa busca.

```
//No arquivo Grafo.c
//Função auxiliar: calcula a menor distancia
int procuraMenorDistancia(float *dist, int *visitado, int NV){
    int i, menor = -1, primeiro = 1;
    for(i = 0; i < NV; i++){
        if(dist[i] >= 0 && visitado[i] == 0){
            if(primeiro){
                menor = i;
                primeiro = 0;
            }else{
                if(dist[menor] > dist[i]){
                    menor = i;
                }
            }
        }
    }
    return menor;
}
```

if: se vértice não foi visitado, então distância é -1. Inicialmente todos os vértices não foram visitados portanto têm distância -1, exceto o vértice inicial, que tem distancia 0.

Procura vértice com menor distância e que não tenha sido visitado.

Verifica se é o primeiro vértice encontrado nessa busca.

Marca como primeiro.

Se não for o primeiro vértice encontrado então é feita a comparação se a distancia do "menor" é maior que a distância do atual [i]. Se for, efetua a troca

Índice do vértice que satisfaz a condição do primeiro "if": possui a menor distância e não foi visitado ainda

A informação que importa: Dentro do Grafo qual é o vértice que tem a menor distância até o momento e que não foi visitado ainda. Esta informação será usada na função menorCaminho\_Grafo()



# Estrutura de Dados 2

## Buscas em Grafos



cont garante que todos os vértices foram visitados, pois ele recebe a quantidade total de vértices.

Cria vetor auxiliar e inicializa distância e anteriores

Enquanto existirem vértices para serem visitados...

```
//No arquivo Grafo.c
//Função principal
void menorCaminho_Grafo(Grafo *gr, int ini, int *ant, float *dist){
    int i, cont, NV, ind, *visitado, u;
    > cont = NV = gr->nro_vertices;
    {
        visitado = (int*) malloc(NV * sizeof(int));
        for(i = 0; i < NV; i++){
            ant[i] = -1;
            dist[i] = -1;
            visitado[i] = 0;
        }
        dist[ini] = 0;
        while(cont > 0){
            .....//restante em outro slide
        }
        free(visitado);
    }
}
```

Vértice inicial recebe distância

# Estrutura de Dados 2



Procura vértice com menor distância e marca como visitado

Conseguir recuperar o vértice "u", marca como visitado e decrementa cont.

for: para cada vértice, visita todos os seus vizinhos

Se for  $< 0$ , significa que ninguém chegou nele ainda, está com distância inválida.

dist[ind] assume novo valor, nesse caso é considerado que a distância é o número de vértices que se tem que passar para chegar em alguém

```
while(cont > 0){
    u = procuraMenorDistancia(dist, visitado, NV);
    if(u == -1){
        break;
    }
    visitado[u] = 1;
    cont--;
    for(i = 0; i < gr->grau[u]; i++){
        ind = gr->arestas[u][i];
        if(dist[ind] < 0){
            dist[ind] = dist[u] + 1;
            //dist[ind] = dist[u]+gr->pesos[u][i];
            //ou peso da aresta
            ant[ind] = u;
        }else{
            if(dist[ind] > dist[u] + 1){
                //if(dist[ind] > dist[u]+1){
                //ou peso da aresta
                dist[ind] = dist[u] + 1;
                //dist[ind] = dist[u]+gr->pesos[u][i];
                //ou peso da aresta
                ant[ind] = u;
            }
        }
    }
}
```

Procura vértice que não foi visitado ainda e tem a menor distância

Teste: não conseguiu encontrar? Pode ser que não exista caminho de todos para todos, é possível que existam vértices inalcançáveis partindo de "ini".

Vértice vizinho de "u".

"u" passa a ser vértice anterior. Para chegar em "ind" tem que vir por "u".

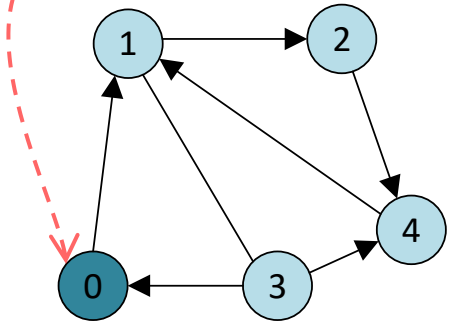
Se já tiver valor de distância válido, verifica se a distância é maior do que  $u+1$ . Se for maior significa que existe um caminho passando por "u" com custo menor

Vou para "ind" com distância de "u+1", e anterior de "ind" passa a ser "u". Modifico o antecessor e qual é a distância do vértice "ind", se existir um caminho menor até lá.

# Estrutura de Dados 2



Recupera 1º vértice:  
dist  $\geq 0$   
Que não tinha sido visitado

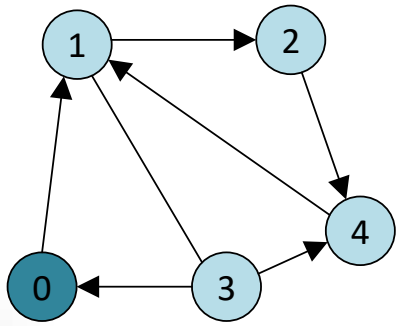


0	→	1	
1	→	3	→ 2
2	→	4	
3	→	0	→ 4
4	→	1	

dist	ant	visitado
0	-1	0
-1	-1	0
-1	-1	0
-1	-1	0
-1	-1	0

Inicia o cálculo com o vértice 0.  
Atribui distância zero a ele  
(início). Restante dos vértices  
recebem distância -1

Atualiza a distância de todos os vizinhos  
dele. A distância do vértice 1 passa à:



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

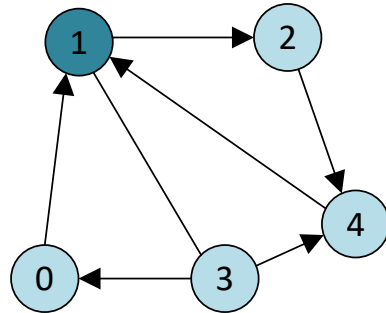
dist	ant	visitado
0	-1	1
1	0	0
-1	-1	0
-1	-1	0
-1	-1	0

Antecessor do vértice 1: para  
chegar no 1 vou pelo vértice 0.

Zero recuperado,  
marca como visitado.

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

# Estrutura de Dados 2



Visita os vizinhos e atualiza distância

Recupera o vértice 1: tem distância e não foi visitado ainda

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

Antecessores de 2 e 3

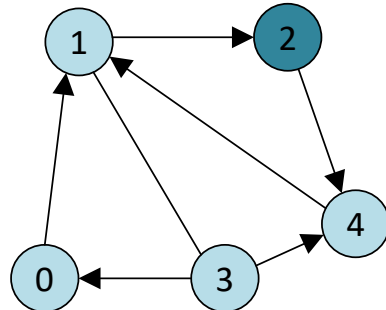
Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 1.  
Verifica e atualiza (se necessário) "dist" e "ant" dos vértices adjacentes (2 e 3).

Recupera vértice 2, visita vizinho (4) e atualiza a distância

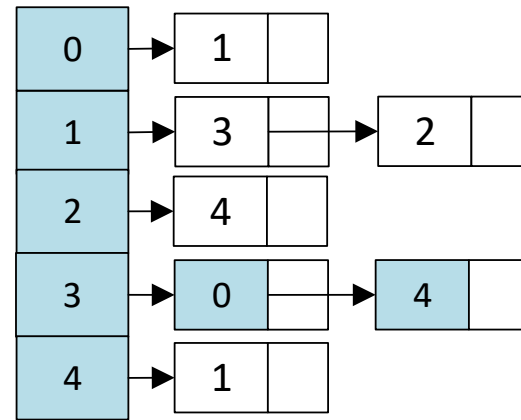
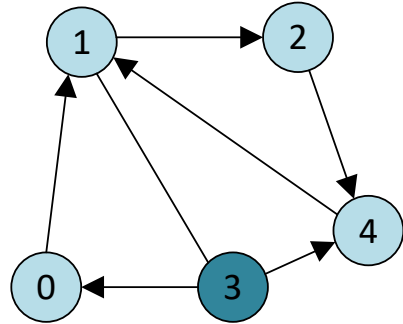
		dist	ant	visitado
0	1	0	-1	1
1	3	1	0	1
2	4	2	1	1
3	0	2	1	0
4	1	3	2	0

Antecessor de 4

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

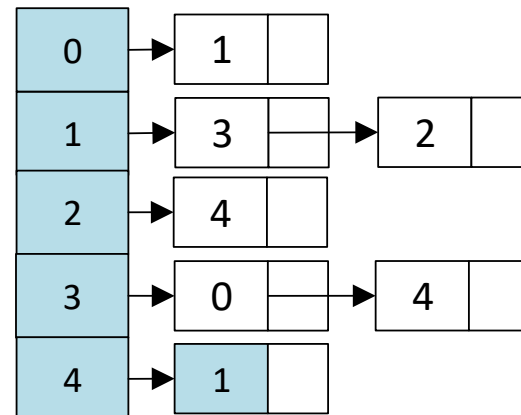
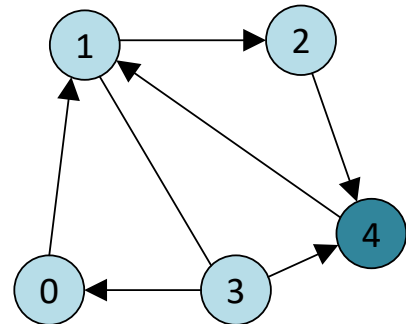


# Estrutura de Dados 2



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

Finda busca pelo menor caminho, temos como resultado dois vetores: **dist** com as distâncias dos vértices em relação ao vértice inicial e **ant** com os vértices anteriores de cada vértice visitado, ou seja, o caminho até ele.

Recupera vértice com menor distância ainda não visitado e 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.

# Estrutura de Dados 2

## Atividade

- Monte o programa Grafo com todas as suas etapas, e entregue no Moodle .

