Grafos – Algoritmos de busca

Prof. Renato Pimentel

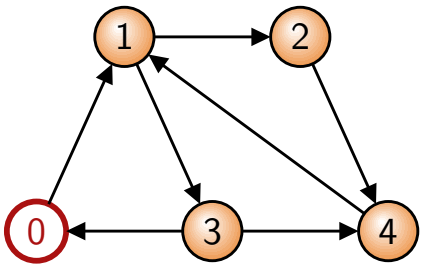
2025/2

Sumário

1 Algoritmos de busca em grafos
- Introdução
- Busca em profundidade
- Busca em largura
- Busca pelo menor caminho
- Exercícios

- O objetivo das **buscas** em grafos é explorar os mesmos, de uma maneira bem determinada.
  - I.e. correspondem a processos sistemáticos de como **caminhar** por seus vértices e arestas.
- As três principais formas de busca:
  1. Busca em profundidade
  2. Busca em largura
  3. Busca pelo menor caminho

- De modo geral, as operações de busca dependem do **vértice inicial**.
  - O ponto de partida é um aspecto bastante importante da própria busca.
  - Por exemplo, em uma busca pelo menor caminho, temos que saber qual é o ponto de partida desse caminho.

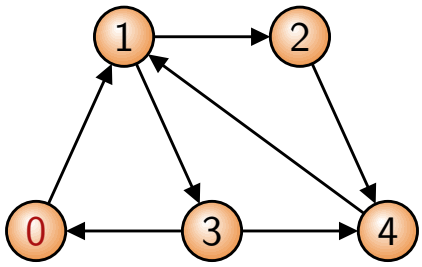


Fonte: adaptado de BACKES, A. R. Algoritmos e Estruturas de Dados em Linguagem C. https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315

- A busca pode precisar visitar todos ou apenas um subconjunto dos vértices.
- Vários problemas em grafos podem ser resolvidos efetuando uma busca.
  - Encontrar o caminho entre dois vértices conectados do grafo;
  - Determinar o menor caminho entre dois vértices do grafo;
  - Resolução de quebra-cabeças (labirintos);
  - Determinar componentes conexas (fortemente ou não), etc.

Grafo para teste I

A fim de demonstrar os algoritmos de busca, utilizamos o seguinte dígrafo:



A criação do mesmo usa a seguinte implementação, considerando-se um máximo de 5 conexões por vértice:

main.c – grafo de teste

```
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
  int eh_digrafo = 1;
```

Grafo para teste II

```
6    Grafo* gr = cria_Grafo(5, 5, 0);
7
8    insereAresta(gr, 0, 1, eh_digrafo, 0);
9    insereAresta(gr, 1, 3, eh_digrafo, 0);
10   insereAresta(gr, 1, 2, eh_digrafo, 0);
11   insereAresta(gr, 2, 4, eh_digrafo, 0);
12   insereAresta(gr, 3, 0, eh_digrafo, 0);
13   insereAresta(gr, 3, 4, eh_digrafo, 0);
14   insereAresta(gr, 4, 1, eh_digrafo, 0);
15
16   // realizar a busca aqui
17
18   libera_Grafo(gr);
19   system("pause");
20   return 0;
21 }
```


Sumário

1 Algoritmos de busca em grafos
- Introdução
- Busca em profundidade
- Busca em largura
- Busca pelo menor caminho
- Exercícios

Busca em profundidade

- O funcionamento da busca em profundidade pode ser assim descrito:

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

- Em inglês: *depth first search* (**DFS**).
- Esse tipo de busca se inicia em um vértice e se aprofunda nos vértices vizinhos deste até encontrar um dos dois casos:
  - o alvo da busca seja encontrado; ou
  - um vértice sem vizinhos ainda não visitados.

- Retrocesso ou **backtracking**:
  - O grafo é percorrido de maneira sistemática até que a busca falhe, ou se encontre um vértice sem vizinhos.
    - ★ Nesse momento, entra em funcionamento o mecanismo de *backtracking*: a busca retorna pelo mesmo caminho percorrido, visando encontrar um caminho alternativo.

- A função que inicializa a busca – `buscaProfundidade_Grafo()` – recebe três parâmetros:
  1. o grafo
  2. o vértice inicial da busca
  3. um *array* que marcará a ordem que cada vértice será visitado – correspondendo ao **resultado** da busca.
- Uma segunda função, `buscaProfundidade()`, recebe, além dos mesmos 3 parâmetros acima, um **contador de visitação**. Trabalha de forma recursiva.
  - A recursão termina quando não houver mais vizinhos a serem visitados.

main.c – busca em profundidade

```
12    .
13    .
14    insereAresta(gr, 4, 1, eh_digrafo, 0);
15
16    int vis[5];
17    buscaProfundidade_Grafo(gr, 0, vis);
18    .
```

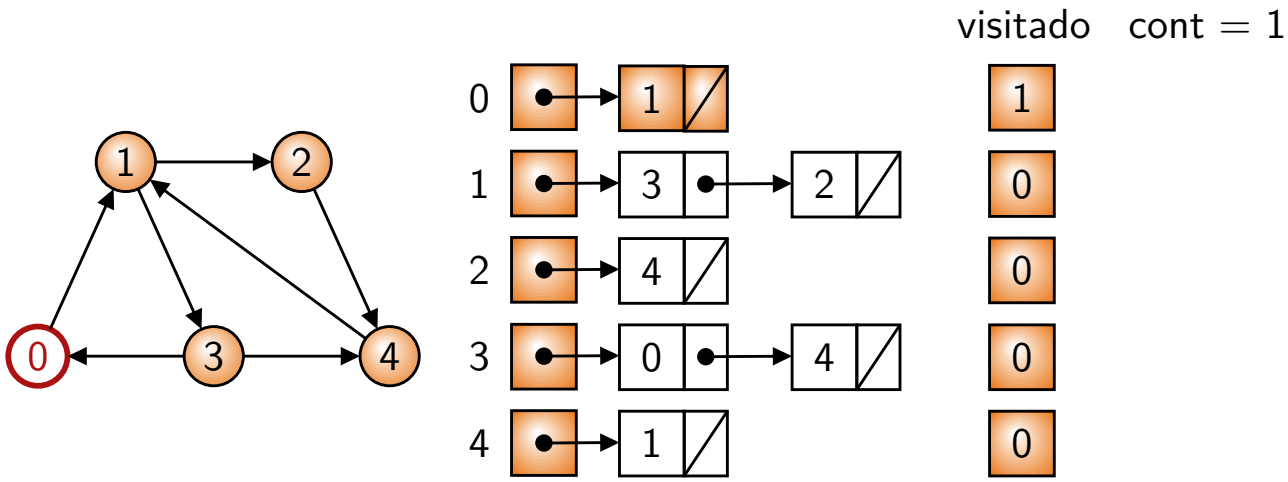

grafo.c – busca em profundidade

```
136 void buscaProfundidade(Grafo *gr, int ini, int *
      visitado, int *cont){
137   int i;
138
139   // marca o vértice como visitado; visita os vizinhos
      ainda não visitados.
140   visitado[ini] = *cont;
141   (*cont)++;
142   for(i=0; i<gr->grau[ini]; i++){
143     if(!visitado[gr->arestas[ini][i]])
144       buscaProfundidade(gr,gr->arestas[ini][i],visitado
      ,cont);
145   }
146 }
```

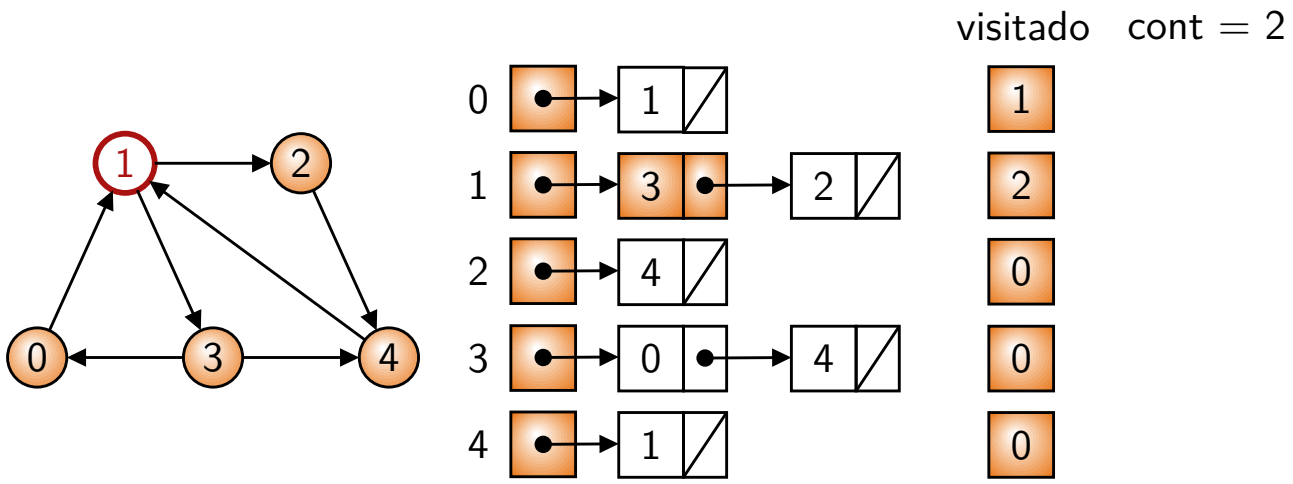
grafo.c – busca em profundidade (inicial, chamada pelo programa principal)

```c
void buscaProfundidade_Grafo(Grafo *gr, int ini, int *visitado){
  int i, cont = 1;

  // marca os vértices como não-visitados.
  for(i=0; i<gr->nro_vertices; i++)
    visitado[i] = 0;
  buscaProfundidade(gr,ini,visitado,&cont);
}
```

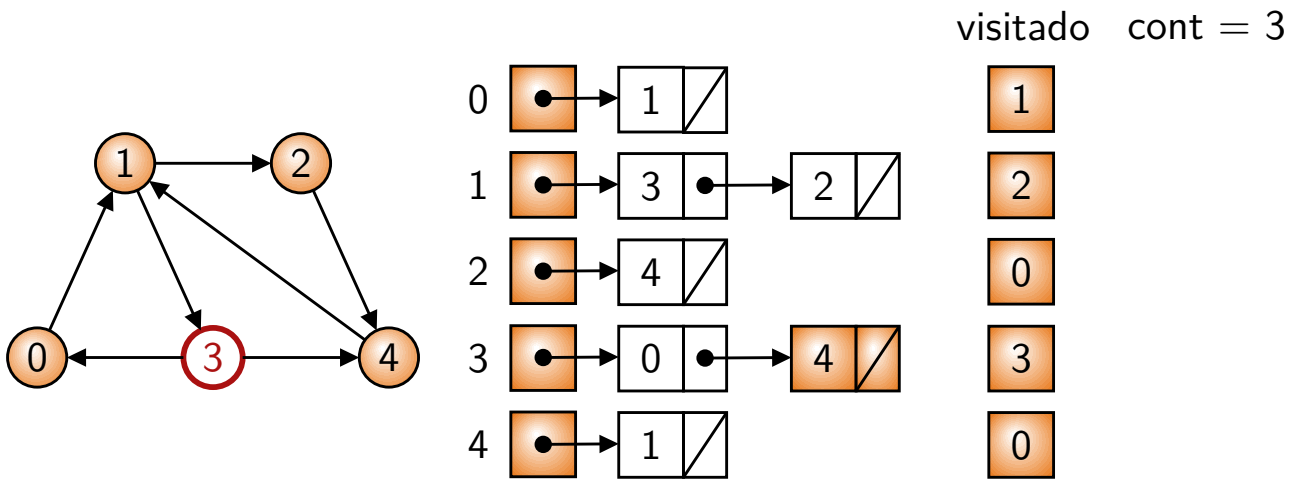
Passo a passo



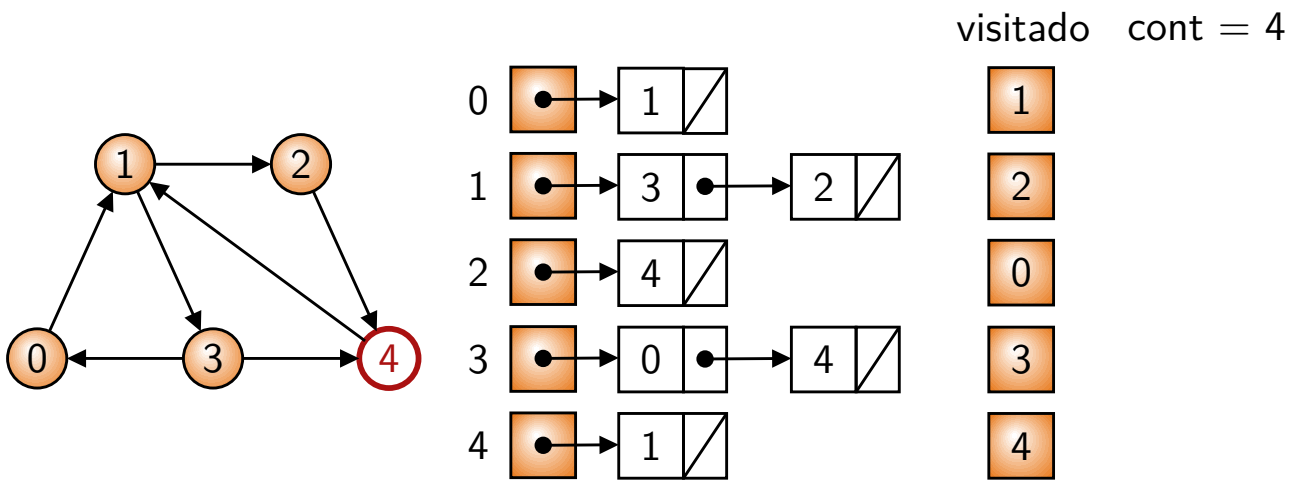
Inicia a busca com o vértice 0. Marca o vértice 0 como visitado e executa a busca para o vértice adjacente (1)



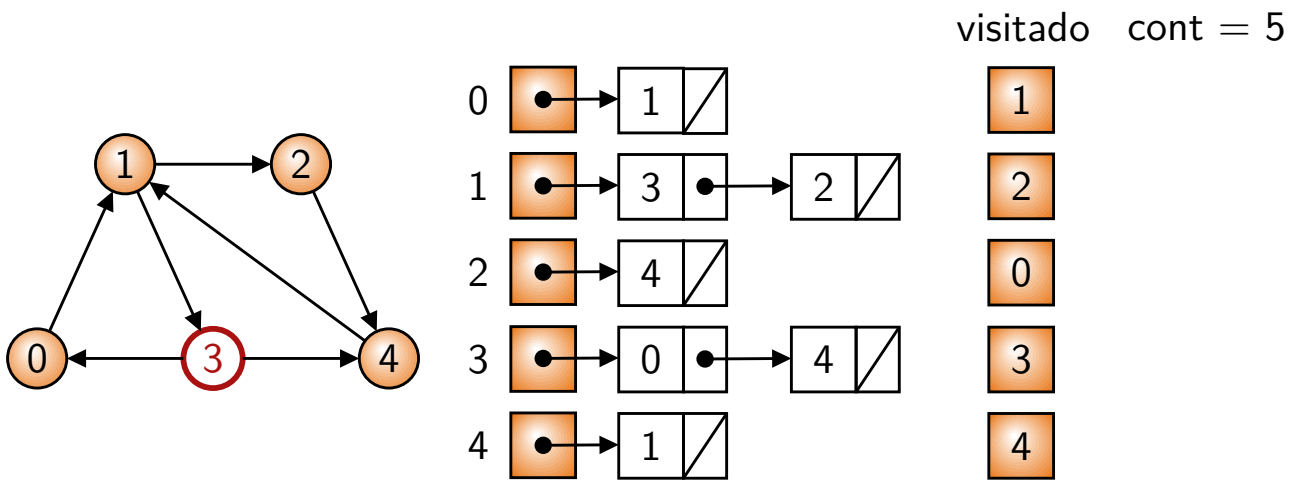
Marca o vértice 1 como visitado e
executa a busca para o primeiro vértice adjacente (3)



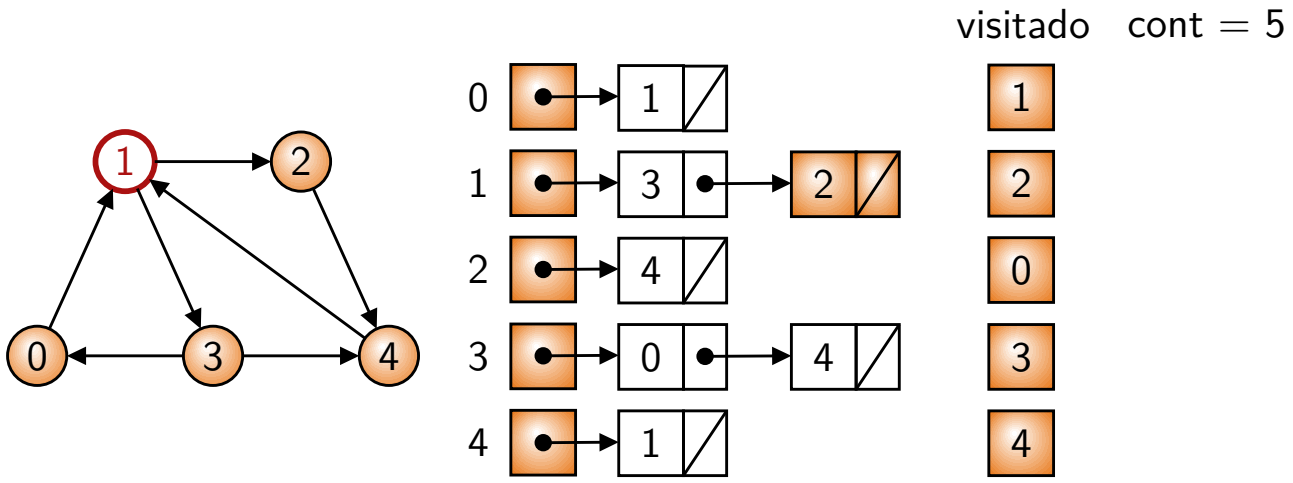
Marca o vértice 3 como visitado e
executa a busca para o primeiro vértice adjacente não visitado (4)



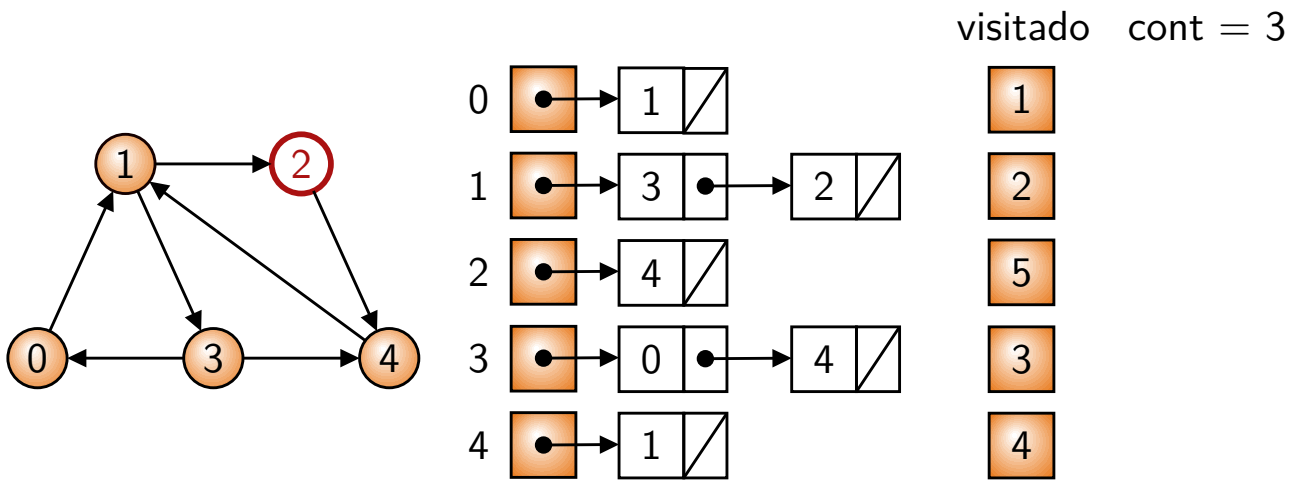
Marca o vértice 4 como visitado.
Todos os vértices adjacentes já foram visitados. Volta para o vértice 3



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



Executa a busca para o segundo vértice adjacente (2).



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

Complexidade

- Considerando um grafo $G(V, A)$, onde $|V|$ é o número de vértices e $|A|$ é o número de arestas, a complexidade no pior caso é:
  - Custo de ir para cada vértice: $O(|V|)$;
  - Custo de transitar em cada aresta: $O(|A|)$;
  - Complexidade da busca no pior caso: $O(|V| + |A|)$.

Aplicações

- Encontrar componentes conexas e fortemente conexas;
- Ordenação topológica de um grafo;
- Procurar a saída de um labirinto;
- Verificar se um grafo é completamente conexo:
  - por exemplo, se uma rede de computadores está funcionando corretamente ou não;
- implementar a **ferramenta de preenchimento de cor** em editores de imagens.

Sumário

1 Algoritmos de busca em grafos
- Introdução
- Busca em profundidade
- Busca em largura
- Busca pelo menor caminho
- Exercícios

Busca em largura

- O funcionamento da busca em largura pode ser assim descrito:

> Partindo de um vértice inicial, a busca explora todos os vizinhos de um vértice. Em seguida, para cada vizinho, repete-se esse processo, visitando os vértices ainda inexplorados.

- Em inglês: *breadth first search* (**BFS**).
- Esse tipo de busca se inicia em um vértice e então visita todos os seus vizinhos antes de se aprofundar na busca. Esse processo continua até que:
  - o alvo da busca seja encontrado; ou
  - não existam mais vértices a serem visitados.

- O algoritmo usa o conceito de **fila**:
  - O grafo é percorrido de maneira sistemática, primeiro marcando 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 a mais tempo) é o primeiro a ser visitado.

- A função que realiza a busca recebe três parâmetros:
  1. o grafo
  2. o vértice inicial da busca
  3. um *array* que marcará a ordem que cada vértice será visitado – correspondendo ao **resultado** da busca.

main.c – busca em largura

```
12  .
13  .
14  insereAresta(gr, 4, 1, eh_digrafo, 0);
15
16  int vis[5];
17  buscaLargura_Grafo(gr, 0, vis);
18  .
19  .
```

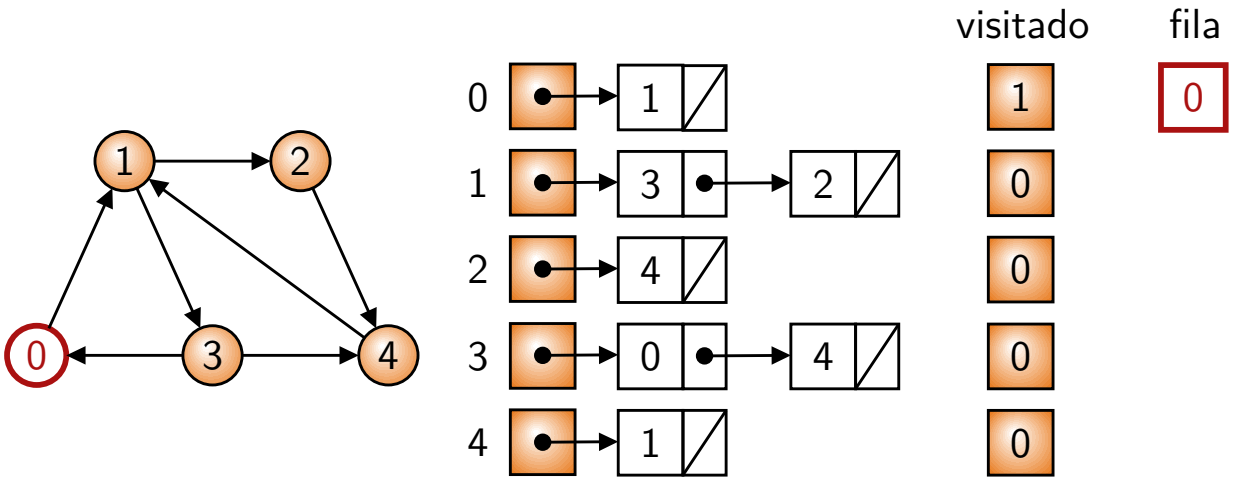
grafo.c – busca em largura

```c
void buscaLargura_Grafo(Grafo *gr, int ini, int *visitado){
  int i, vert, NV, cont = 1;
  int *fila, IF = 0, FF = 0; // IF/FF: início/final da fila

  // marca os vértices como não-visitados
  for(i=0; i<gr->nro_vertices; i++)
    visitado[i] = 0;

  // Cria a fila; visita e insere "ini" na mesma.
  NV = gr->nro_vertices;
  fila = (int*) malloc(NV * sizeof(int));
  FF++;
  fila[FF] = ini;
  visitado[ini] = cont;
  while(IF != FF){
    // toma o primeiro da fila
```

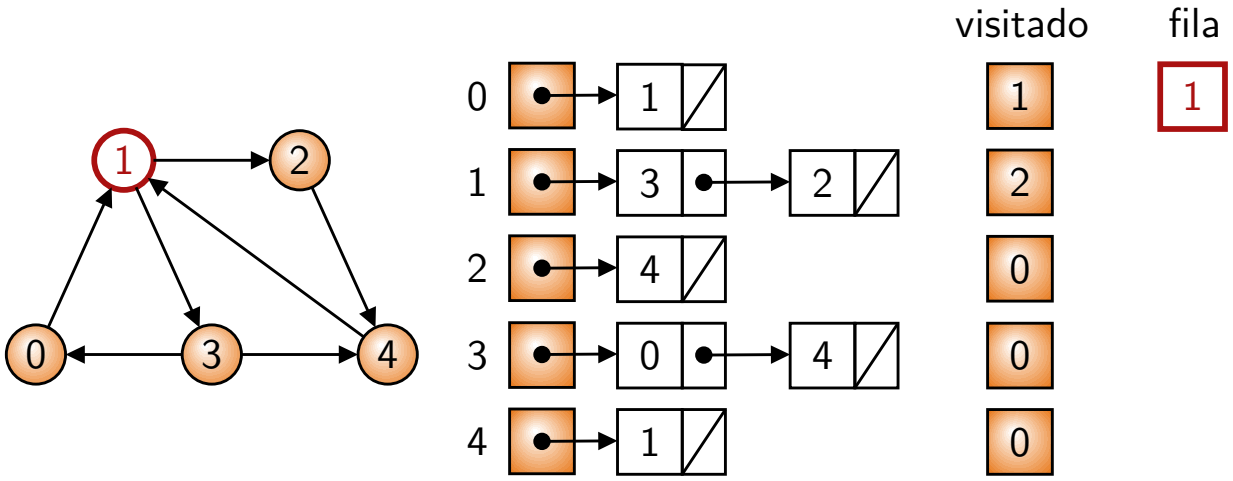


```c
    IF = (IF + 1) % NV;
    vert = fila[IF];

    //cont++;

    // visita os vizinhos ainda não visitados, colocando-os na fila
    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);
}
```

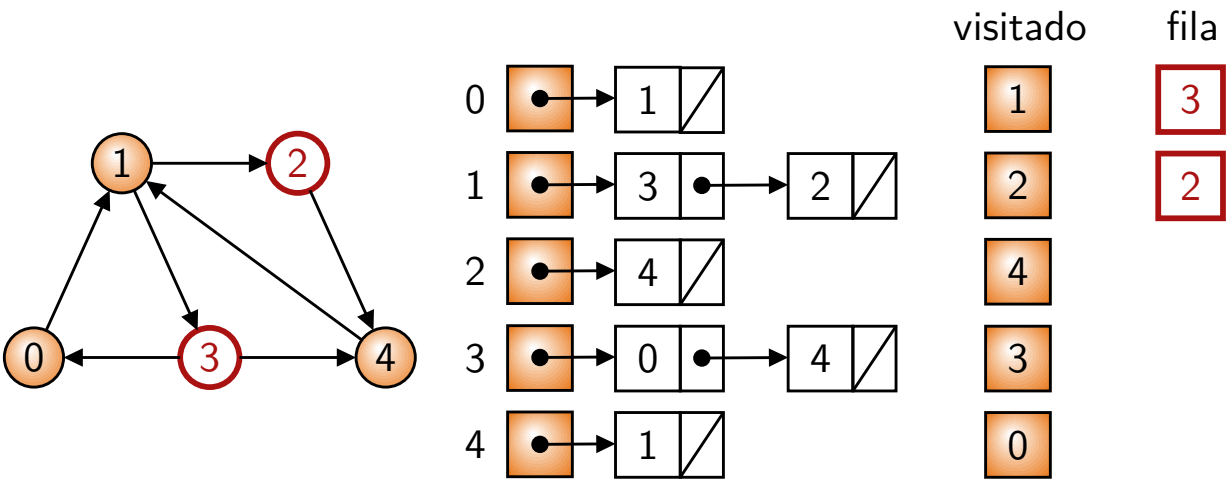
Passo a passo



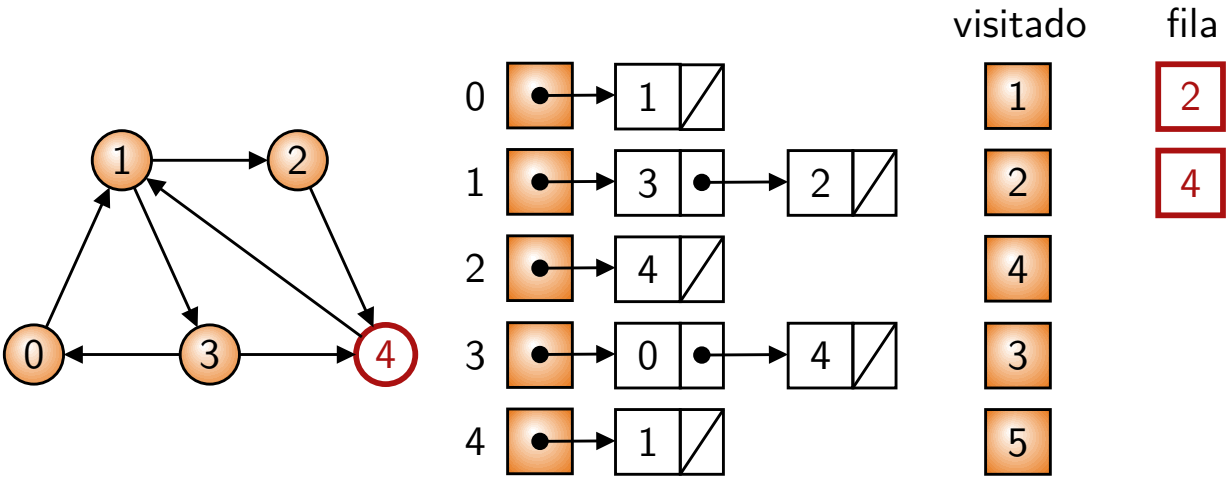
Inicializa a busca em largura.
Visita e insere na fila o vértice 0.



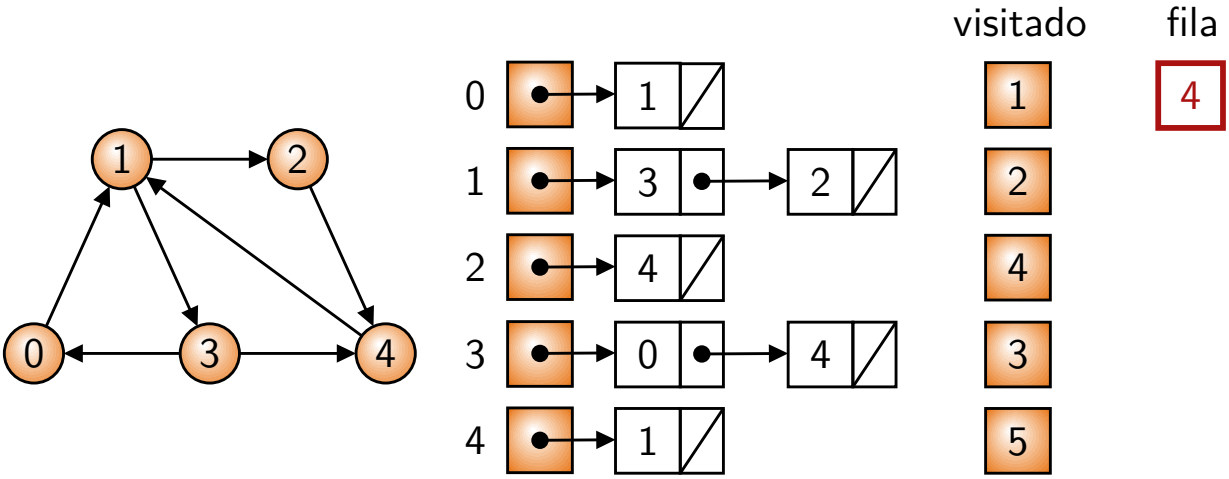
Remove vértice 0 da fila.
Insere na fila o vértice adjacente (1) que não foi visitado.



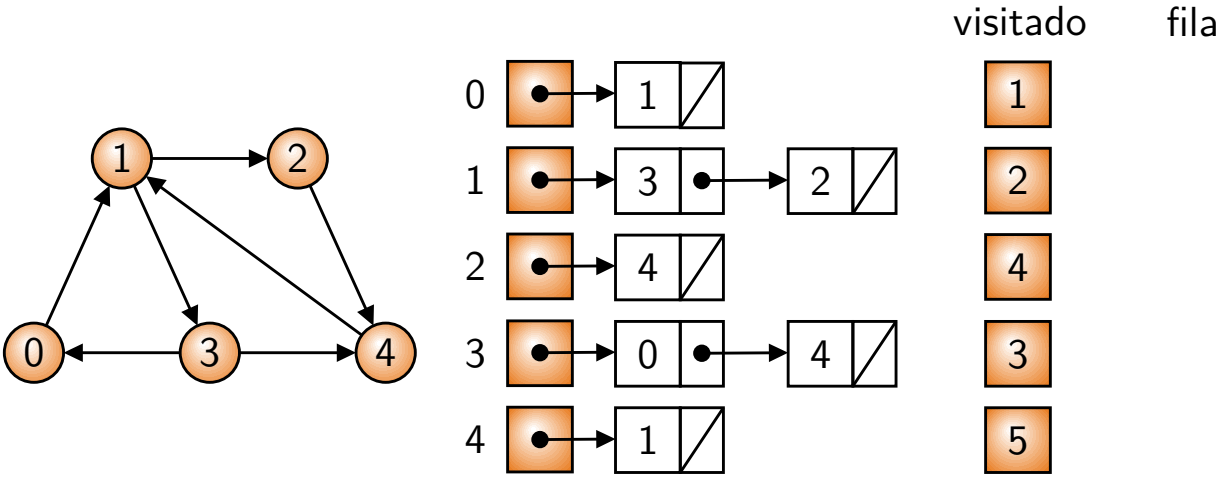
Remove vértice 1 da fila.
Insere na fila os vértices adjacentes (3 e 2) que não foram visitados.



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



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



Remove vértice 4 da fila. Vértices adjacentes já foram visitados.
Fila vazia: fim da busca!

Complexidade

- Considerando um grafo $G(V, A)$, onde $|V|$ é o número de vértices e $|A|$ é o número de arestas, a complexidade no pior caso é:
  - Custo de inserção e remoção em fila: constante;
  - Custo de enfileirar e remover todos os vértices uma vez: $O(|V|)$;
  - Custo de utilizar todas as arestas: $O(|A|)$;
  - Complexidade da busca no pior caso: $O(|V| + |A|)$.

Aplicações

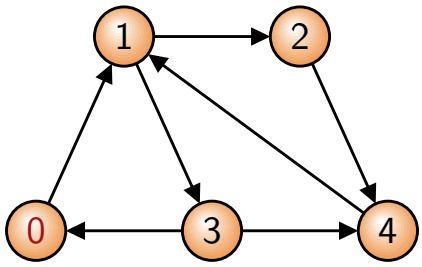
- Encontrar todos os vértices conectados a apenas uma componente conexa;
- Encontrar o **menor caminho** entre dois vértices;
- Testar se um grafo é bipartido;
- Roteamento: encontrar um número mínimo de *hops* em uma rede;
  - Os **hops** são os vértices intermediários no caminho correspondente à conexão.
- Encontrar o número mínimo de intermediários entre 2 pessoas.

Sumário

1. Algoritmos de busca em grafos
   - Introdução
   - Busca em profundidade
   - Busca em largura
   - Busca pelo menor caminho
   - Exercícios

Busca pelo menor caminho

- Relembrando:
  - Dado um grafo $G(V, A)$, um vértice $u \in V$ está **conectado** a outro vértice, $v \in V$ se existe um caminho de u para v.
- Quando u e v são adjacentes (vizinhos), o menor caminho de u para v é a aresta que os conecta.
- No entanto, é muito comum não existir uma aresta conectando u e v diretamente.
  - Por exemplo, os vértices 0 e 4 não são adjacentes.



- Quando u e v não são adjacentes, mas conectados, é possível representar o(s) **caminho(s)** de u para v em termos das arestas que compõem o(s) mesmo(s);
  - No dígrafo sendo usado, um possível caminho de 1 para 4 é dado por $(0,1)$, $(1,2)$, $(2,4)$.
    Outra possibilidade: $(0,1)$, $(1,3)$, $(3,4)$.

Menor caminho

O **menor caminho** (ou **caminho geodésico**) entre dois vértices é o que representa o **menor comprimento**[a] dentre todos os caminhos possíveis entre ambos.

[a] Neste caso, definido em termos da quantidade de arestas do caminho.

  - Ambos os caminhos acima têm comprimento 3.
  - em **grafos ponderados**, o comprimento do caminho corresponde à **soma dos pesos** das arestas que o define.

Algoritmo de Dijkstra

- Uma das maneiras de achar o menor caminho é utilizando o **algoritmo de Dijkstra**:
  - Talvez o mais conhecido algoritmo;
  - Trabalha com grafos e dígrafos, ponderados ou não.
  - No caso de um grafo ponderado, as arestas **não** podem ter pesos negativos;
- Como funciona o algoritmo de Dijkstra:

Partindo de um vértice inicial, o algoritmo de Dijkstra calcula a menor distância deste vértice a todos os demais a ele conectados.

- A função que inicializa a busca – `menorCaminho_Grafo()` – recebe quatro parâmetros:
  1. o grafo
  2. o vértice inicial
  3. um *array*, `ant`, para armazenar o antecessor dos vértices no caminho
  4. outro *array*, `dist`, para a distância a partir do vértice inicial.
- Outra função, `procuraMenorDistancia()`, recebe três parâmetros:
  - o *array* `dist`
  - um segundo *array*, para representar os vértices visitados; e
  - o número de vértices do grafo.

main.c – busca pelo menor caminho

```
12    .
13    .
14    insereAresta(gr, 4, 1, eh_digrafo, 0);
15
16    int ant[5];
17    float dist[5];
18    menorCaminho_Grafo(gr,0, ant, dist);
19    .
```


grafo.c – menor caminho

```
156 int procuraMenorDistancia(float *dist, int *visitado,
      int NV){
157   int i, menor = -1, primeiro = 1;
158   for (i=0; i < NV; i++){
159     // procura vértice com menor distância e que não
      tenha sido visitado
160     if (dist[i] >= 0 && visitado[i] == 0){
161       if(primeiro){
162         menor = i;
163         primeiro = 0;
164       } else {
165         if (dist[menor] > dist[i])
166           menor = i;
167       }
168     }
169   }
170   return menor;
171 }
```

grafo.c – menor caminho

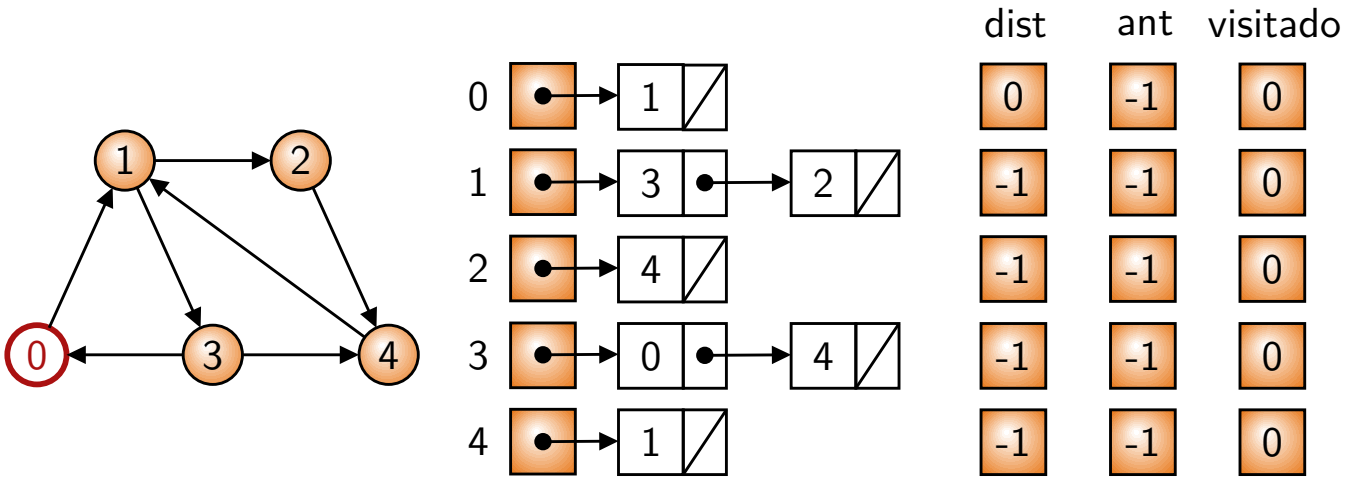
```c
173 void menorCaminho_Grafo(Grafo *gr, int ini, int *ant,
      float *dist){
174   int i, cont, NV, ind, *visitado, vert;
175   cont = NV = gr->nro_vertices;
176
177   // cria vetor auxiliar. Inicializa distâncias e
      anteriores
178   visitado = (int*) malloc(NV * sizeof(int));
179   for(i=0; i < NV; i++){
180     ant[i] = -1;
181     dist[i] = -1;
182     visitado[i] = 0;
183   }
184   dist[ini] = 0;
185
186   // procura vértice com menor distância e o marca como
      visitado
187   while(cont > 0){
```



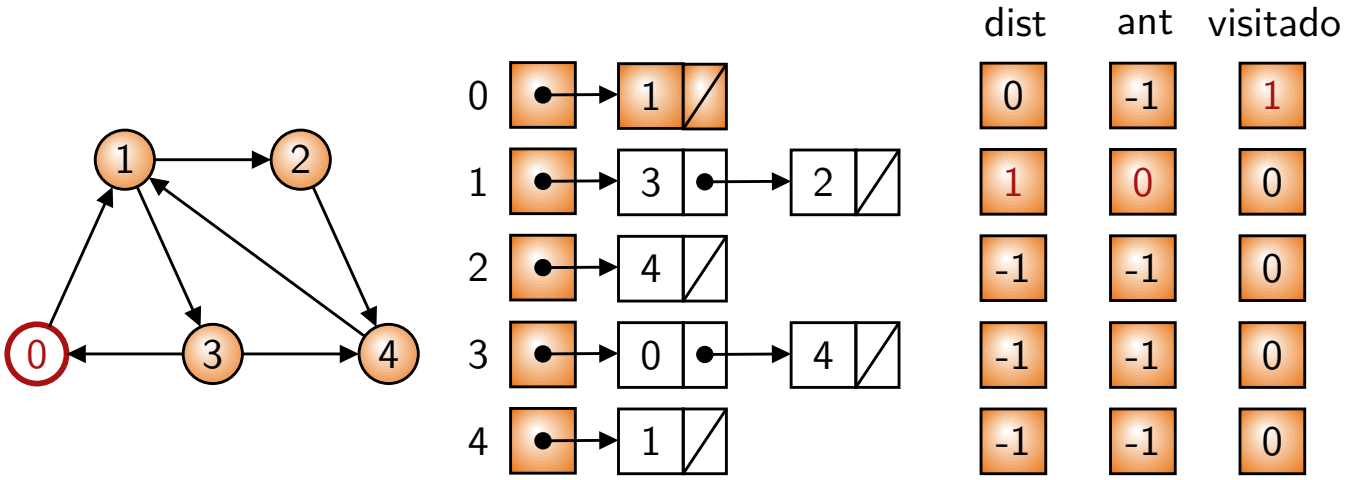
```c
188     vert = procuraMenorDistancia(dist, visitado, NV);
189     //printf("u = %d\n",u);
190     if(vert == -1)
191       break;
192
193     visitado[vert] = 1;
194     cont--;
195
196     for(i=0; i<gr->grau[vert]; i++){ // para cada vé
      rtice vizinho
197       ind = gr->arestas[vert][i];
198       // atualiza distâncias dos vizinhos
199       if(dist[ind] < 0){
200          dist[ind] = dist[vert] + 1;//ou peso da aresta
201          ant[ind] = vert;
202       }else{
203         if(dist[ind] > dist[vert] + 1){
204           dist[ind] = dist[vert] + 1;//ou peso da
      aresta
205           ant[ind] = vert;
```

```
            }
        }
    }
  }

  free(visitado);
}
```

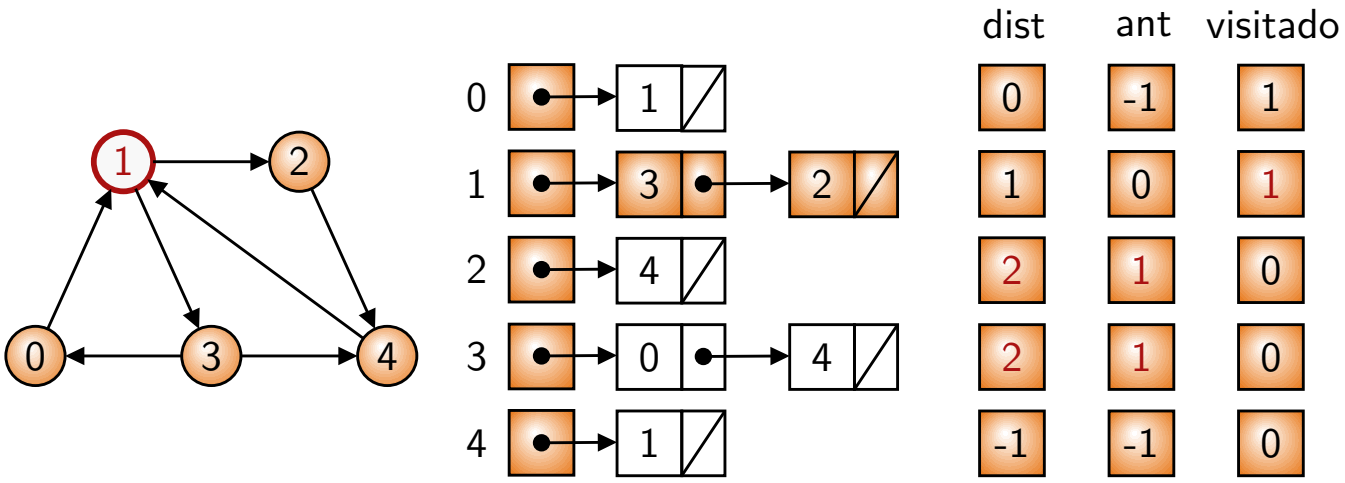
Passo a passo



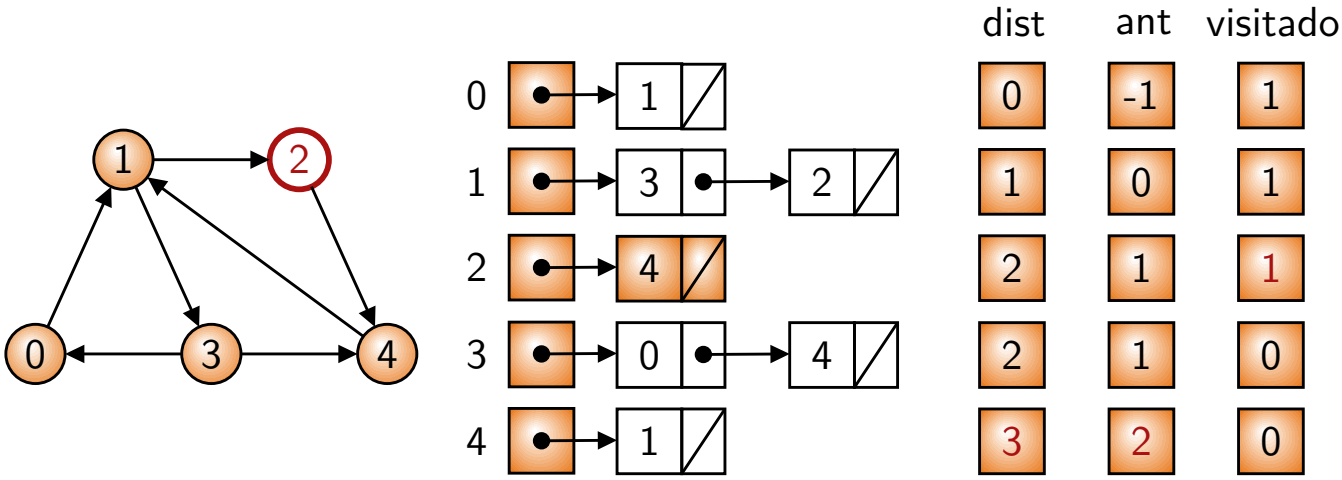
Inicia o cálculo com o vértice 0.
Atribui distância ZERO a ele (início).
O restante dos vértices recebem distância -1



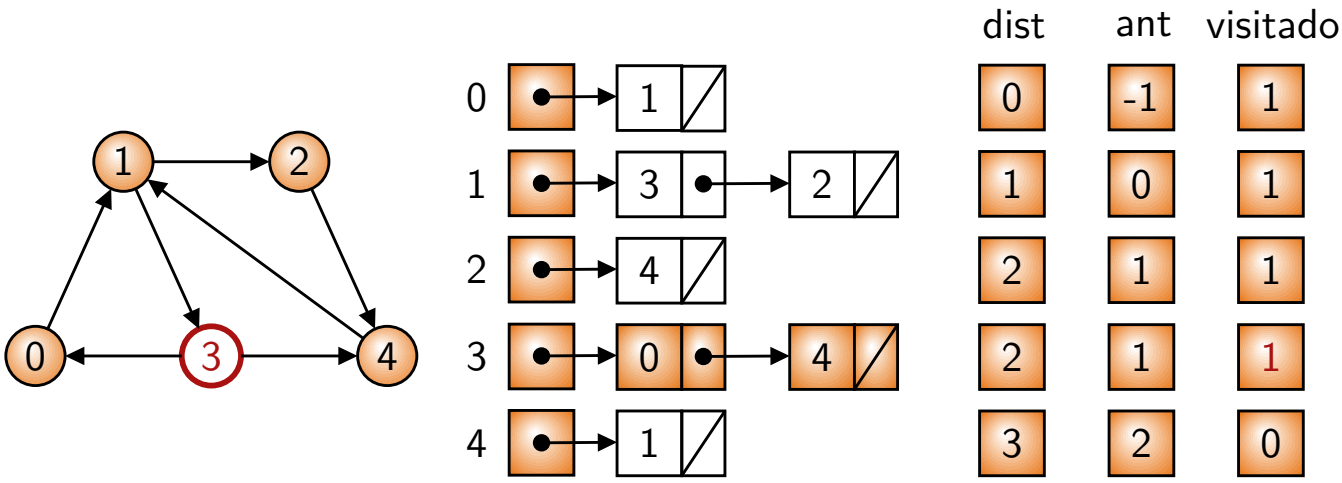
Recupera vértice com menor distância ainda não visitado
e o marca como visitado: vértice 0.
Verifica e atualiza (se necessário) dist e ant do vértice adjacente (1)



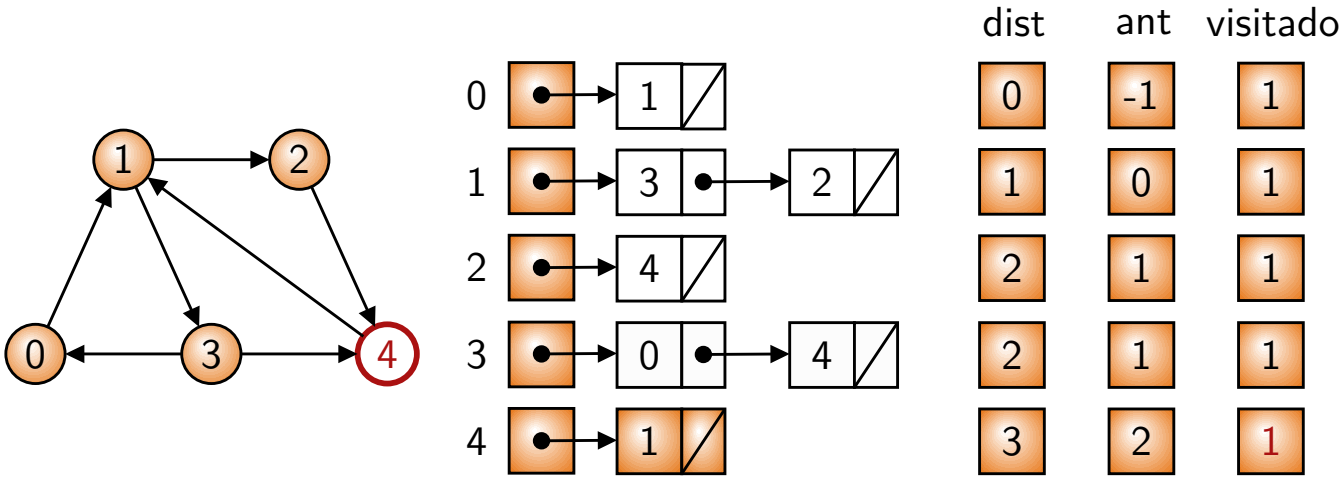
Recupera vértice com menor distância ainda não visitado
e o marca como visitado: vértice 1.
Verifica e atualiza (se necessário) dist e ant dos vértices adjacentes (2 e 3)



Recupera vértice 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)



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)



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 visitados: fim**

Aplicações

- Encontrar o grau de separação entre duas pessoas em uma rede social;
- Encontrar um trajeto num mapa rodoviário;
- Programar robôs para explorar áreas;
- Algoritmos de roteamento.

Sumário

1. Algoritmos de busca em grafos
   - Introdução
   - Busca em profundidade
   - Busca em largura
   - Busca pelo menor caminho
   - Exercícios

Exercícios I

1. Escreva um algoritmo para verificar se um grafo é acíclico, usando busca em profundidade.
2. Escreva um algoritmo para determinar as componentes conexas de um grafo, usando busca em profundidade.
3. Escreva uma versão não recursiva do algoritmo de busca em profundidade. Dica: o algoritmo de busca em largura usa uma fila. No caso da DFS, qual ED auxiliar seria mais apropriada para evitar a recursão?
4. Adapte o algoritmo de Dijkstra visto para trabalhar com grafos ponderados, e em seguida, explique: por que não é possível trabalhar com pesos menores que zero?
5. Adapte os algoritmos das buscas em profundidade e largura para se trabalhar com matriz de adjacências.

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