Grafos

Definição

Pensando na forma de representar conjuntos de objetos e as relações entre eles, grafos são modelos matemáticos que representam essas relações.

São uma forma de solucionar problemas computáveis.

Buscam o desenvolvimento de Algoritmos mais eficientes.

Por exemplo:

1. Melhor caminho entre minha casa até outra cidade,
2. Uma Jogada eficiente no Xadrez,
3. Resolver um Quebra-Cabeças, etc.

Definição

OS Grafos são definidos por dois conjuntos G(V,A).

Onde V, Conjunto de vértices (não vazio).

A Conjunto de Arestas.

G(V,A)

V = {1,2,3,4}

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

Definição

Tipos de Grafo:

- Grafo Trivial
- Grafo Simples
- Grafo Completo
- Grafo Regular
- Subgrafo
- Grafo Bipartido
- Grafo Conexo e Desconexo
- Grafos Isomorfos
- Grafo Ponderado
- Grafo Euleriano
- Grafo Semi-Euleriano
- Grafo Hamiltoniano

Representação:

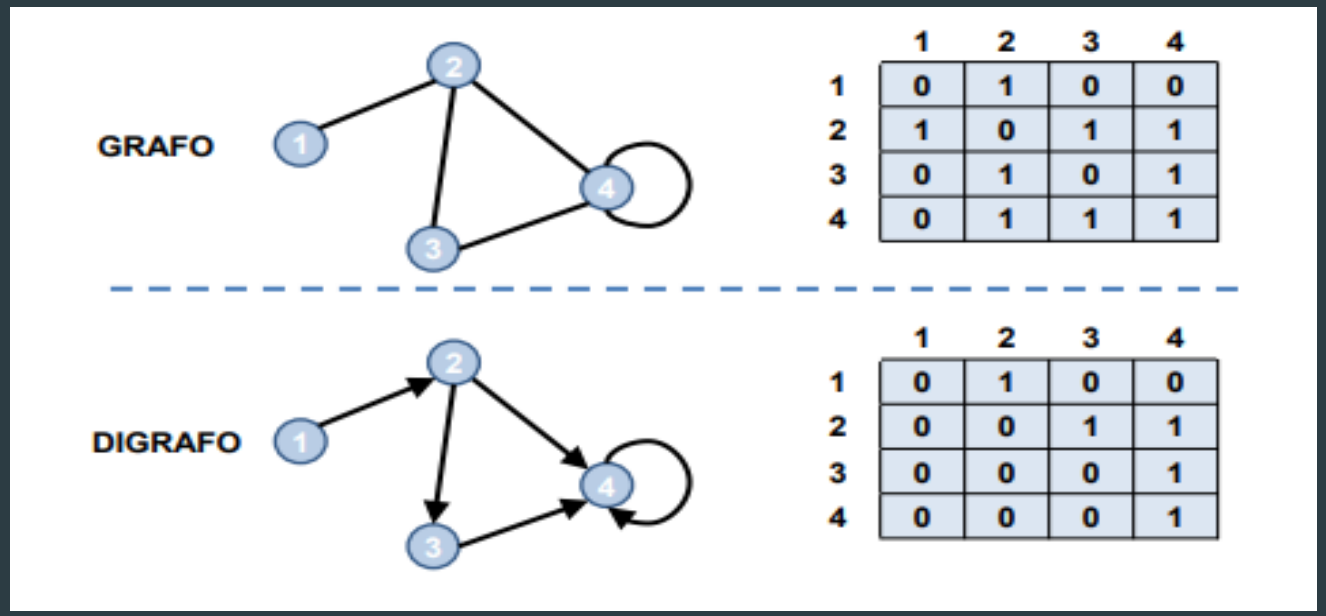
Matriz de Adjacência

Utiliza uma matriz N x N para armazenar o grafo, onde N é o número de vértices

Uma aresta é representada por uma marca na posição

(i , j) da matriz

Aresta liga o vértice i ao j



	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

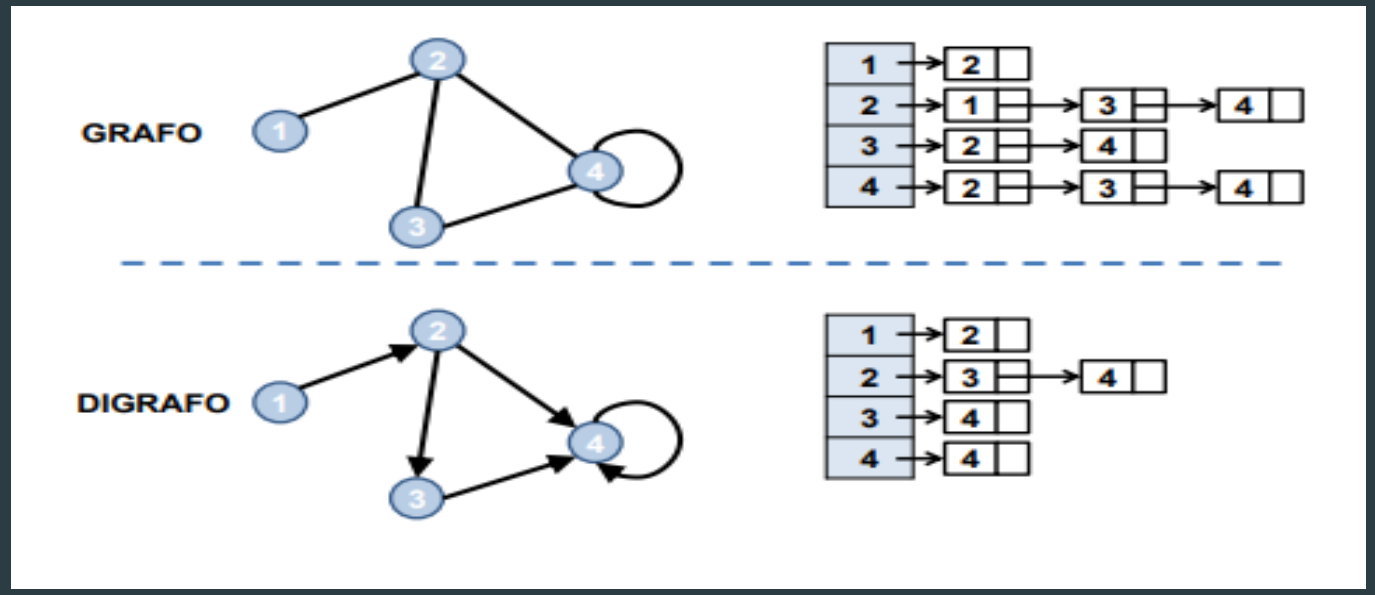
Representação:

Lista de Adjacência:

Usa lista de vértices para descrever as relações

Um grafo de N vértices utiliza um array de ponteiros com tamanho N para armazenar os vértices.

Para cada vértice é criada uma lista de arestas para armazenar o índice ao qual ele se conecta.



Grafo utilizando lista de adjacência:

As características do grafo:

```
 6  typedef struct grafo{
 7      int eh_ponderado;
 8      int nro_vertices;
 9      int grau_max;
10      int** arestas;
11      float** pesos;
12      int* grau;
13  }Grafo;
```

Grafo utilizando lista de adjacência:

A criação do grafo:

```
15  Grafo* cria_Grafo(int nro_vertices, int grau_max, int eh_ponderado){
16      Grafo *gr;
17      gr = (Grafo*) malloc(sizeof(struct grafo));
18      if(gr != NULL){
19          int i;
20          gr->nro_vertices = nro_vertices;
21          gr->grau_max = grau_max;
22          gr->eh_ponderado = (eh_ponderado != 0)?1:0;
23          gr->grau = (int*) calloc(nro_vertices,sizeof(int));
24
25          gr->arestas = (int**) malloc(nro_vertices * sizeof(int*));
26          for(i=0; i<nro_vertices; i++)
27              gr->arestas[i] = (int*) malloc(grau_max * sizeof(int));
28
29          if(gr->eh_ponderado){
30              gr->pesos = (float**) malloc(nro_vertices * sizeof(float*));
31              for(i=0; i<nro_vertices; i++)
32                  gr->pesos[i] = (float*) malloc(grau_max * sizeof(float));
33          }
34
35      }
36      return gr;
37  }
```

Grafo utilizando lista de adjacência:

Inserindo arestas no grafo:

```
55
56  int insereAresta(Grafo* gr, int orig, int dest, int eh_digrafo, float peso){
57      if(gr == NULL)
58          return 0;
59      if(orig < 0 || orig >= gr->nro_vertices)
60          return 0;
61      if(dest < 0 || dest >= gr->nro_vertices)
62          return 0;
63
64      gr->arestas[orig][gr->grau[orig]] = dest;
65      if(gr->eh_ponderado)
66          gr->pesos[orig][gr->grau[orig]] = peso;
67      gr->grau[orig]++;
68
69      if(eh_digrafo == 0)
70          insereAresta(gr,dest,orig,1,peso);
71      return 1;
72  }
73
```

Grafo utilizando lista de adjacência:

Inicializando e colocando as arestas:

```
5   int main(){
6
7       Grafo* gr = cria_Grafo(2, 2, 0);
8
9       insereAresta(gr, 0, 1, 1, 0);
10
11
12      return 0;
13  }
14
```

Busca em Profundidade

- Aplicações:
- Procurar a Saída de um labirinto;
- Resolver Quebra Cabeças;
- Útil para encontrar componentes conectados e fortemente conectados;
- Ordenação Topológica do Grafo;

- Funcionamento:
- A partir de uma busca inicial, percorre todos os possíveis vizinhos de um vértice antes de efetuar o backtracking. Ou seja ele percorre até ocorrer uma falha, não encontrando vértices vizinhos.

Busca em Profundidade

- Criação do Grafo

*main.c

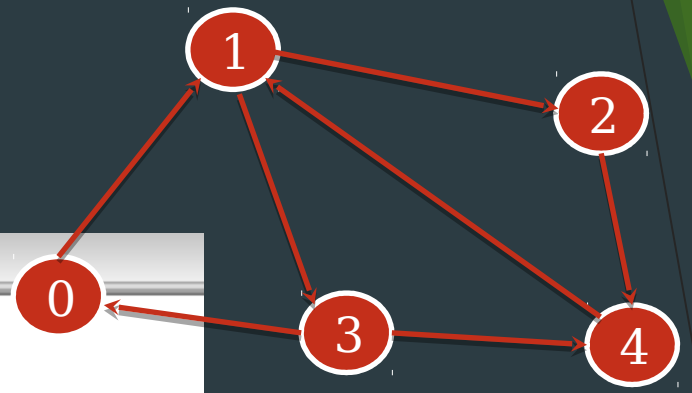
```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"

int main(){
    int eh_digrafo = 1;
    Grafo* gr = cria_Grafo(5,5,0);
    insereAresta(gr, 0, 1, eh_digrafo, 0);
    insereAresta(gr, 1, 3, eh_digrafo, 0);
    insereAresta(gr, 1, 2, eh_digrafo, 0);
    insereAresta(gr, 2, 4, eh_digrafo, 0);
    insereAresta(gr, 3, 0, eh_digrafo, 0);
    insereAresta(gr, 3, 4, eh_digrafo, 0);
    insereAresta(gr, 4, 1, eh_digrafo, 0);
    int vis[5];

    buscaProfundidade_Grafo(gr, 0, vis);

    libera_Grafo(gr);

    system("pause");
```



Busca em Profundidade

*main.c

```
28
29  // função que realiza o calculo
30  void buscaProfundidade(Grafo *gr, int ini, int *visitado, int cont){
31
32      int i;
33      visitado[ini] = cont;
34      for(i=0; i<gr->grau[ini]; i++){
35          if(!visitado[gr->arestas[ini][i])
36              buscaProfundidade(gr,gr->arestas[ini][i], visitado, cont+1);
37      }
38  }
39  // Função de Interface
40  void buscaProfundidade_Grafo(Grafo *gr, int ini, int *visitado){
41
42      int i, cont=1;
43      for(i=0; i<gr->nro_vertices; i++)
44          visitado[i] = 0;
45      buscaProfundidade(gr,ini,visitado,cont);
46
47  }
48
```

Busca em largura

- Funcionamento:

Partindo de um vértice inicial, a busca explora todos os vizinhos de um vértice. Em seguida, para cada vértice vizinho, ela repete esse processo, visitando os vértices ainda inexplorados. Esse processo continua até que o alvo da busca seja encontrado ou não existam mais vértices a serem visitados.

Busca em largura

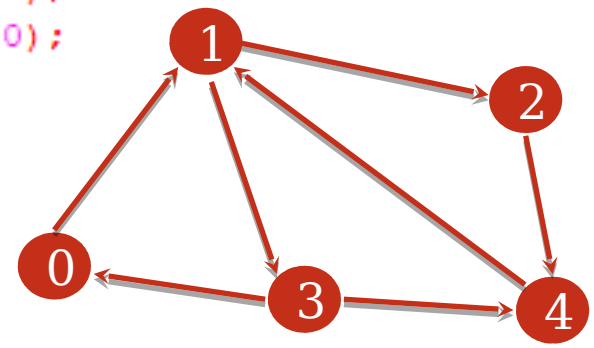
- Aplicações:

- achar todos os vértices conectados a apenas um componente;
- achar o menor caminho entre dois vértices;
- testar se um grafo é bipartido;
- encontrar número mínimo de intermediários entre 2 pessoas

Busca em largura

- Função principal

```
5   int main(){
6
7       int eh_digrafo = 1;
8       Grafo* gr = cria_Grafo(5, 5, 0);
9
10      insereAresta(gr, 0, 1, eh_digrafo, 0);
11      insereAresta(gr, 1, 2, eh_digrafo, 0);
12      insereAresta(gr, 1, 3, eh_digrafo, 0);
13      insereAresta(gr, 2, 4, eh_digrafo, 0);
14      insereAresta(gr, 3, 0, eh_digrafo, 0);
15      insereAresta(gr, 3, 4, eh_digrafo, 0);
16      insereAresta(gr, 4, 1, eh_digrafo, 0);
17
18      int vis[5];
19      buscaLargura_Grafo(gr,0,vis);
20
21
22      return 0;
23  }
```



Busca em largura

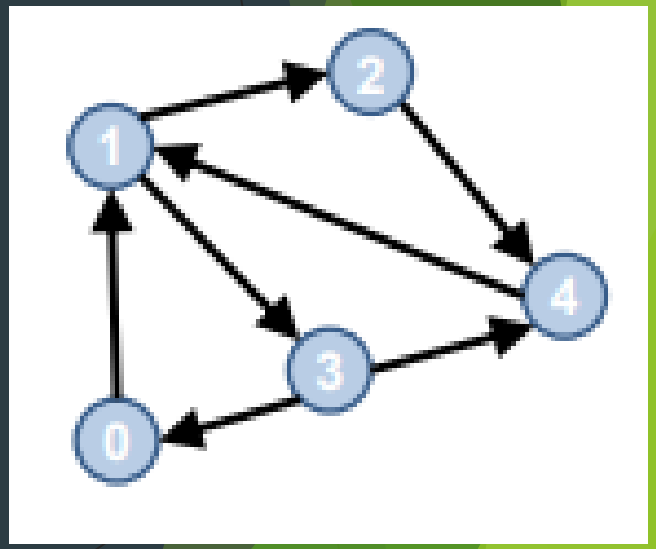
- Função

```
193  void buscaLargura_Grafo(Grafo *gr, int ini, int *visitado){
194      int i, vert, NV, cont = 1;
195      int *fila, IF = 0, FF = 0;
196      for(i=0; i<gr->nro_vertices; i++)
197          visitado[i] = 0;
198
199      NV = gr->nro_vertices;
200      fila = (int*) malloc(NV * sizeof(int));
201      FF++;
202      fila[FF] = ini;
203      visitado[ini] = cont;
204      while(IF != FF){
205          IF = (IF + 1) % NV;
206          vert = fila[IF];
207          cont++;
208          for(i=0; i<gr->grau[vert]; i++){
209              if(!visitado[gr->arestas[vert][i]]){
210                  FF = (FF + 1) % NV;
211                  fila[FF] = gr->arestas[vert][i];
212                  visitado[gr->arestas[vert][i]] = cont;
213              }
214          }
215      }
216      free(fila);
217  }
```

Busca pelo menor caminho

- O menor caminho entre dois vértices é a aresta que os conecta.
- No entanto, é muito comum não existir uma aresta conectando dois vértices

  Por exemplo: Os vértices 0 e 4 não são adjacentes



Busca por menor caminho

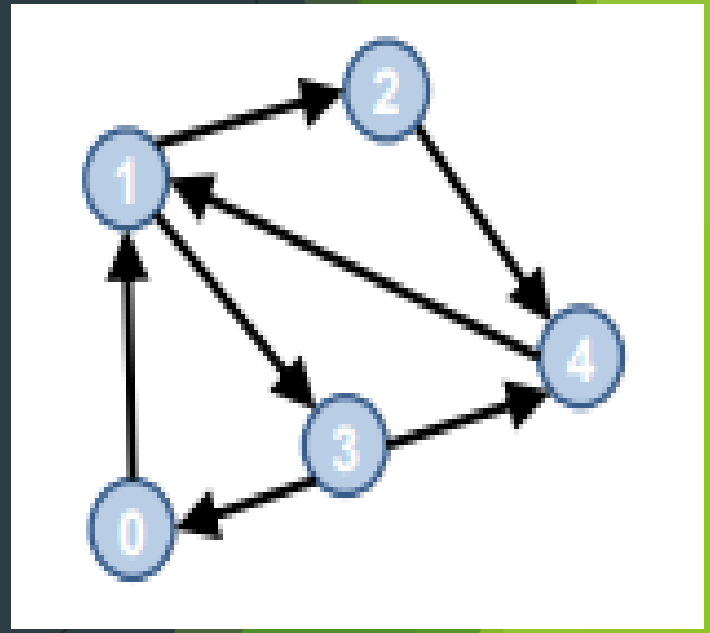
- **Caminho:**

  Quando dois vértices que não são adjacentes, eles podem ser conectados por uma sequência de arestas (0,1),(1,2),(2,4)

- **Menor caminho:**

  É a menor sequência de arestas que liga os dois vértices.

- O comprimento pode ser o número de arestas que conectam os dois vértices ou a soma dos pesos das arestas que compõem esse caminho (grafo ponderado

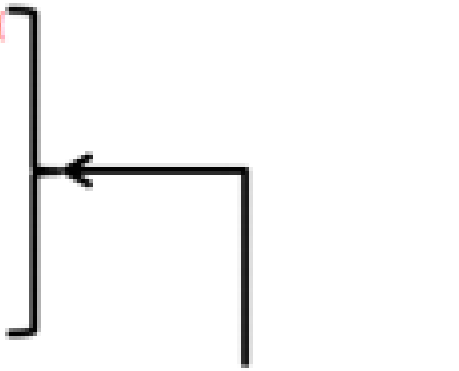


Busca pelo menor caminho

- **Funcionamento**

Partindo de um vértice inicial, o algoritmo de Dijkstra calcula a menor distância deste vértice a todos os demais (desde que exista um caminho entre eles):

```
1  int procuraMenorDistancia(float *dist, int *visitado,
2                            int NV){
3      int i, menor = -1, primeiro = 1;
4      for(i=0; i < NV; i++){
5          if(dist[i] >= 0 && visitado[i] == 0){
6              if(primeiro){
7                  menor = i;
8                  primeiro = 0;
9              }else{
10                 if(dist[menor] > dist[i])
11                     menor = i;
12             }
13         }
14     }
15     return menor;
16 }
```



Procura vértice com menor distância e que não tenha sido visitado

Busca pelo menor caminho

Cria vetor auxiliar.
Inicializa distâncias
e anteriores

Procura vértice com
menor distância e
marca como visitado

```
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){
        u = procuraMenorDistancia(dist, visitado, NV);
        if(u == -1)
            break;

        visitado[u] = 1;
        cont--;
        //CONTINUA...
    }
    free(visitado);
}
```

Busca pelo menor caminho

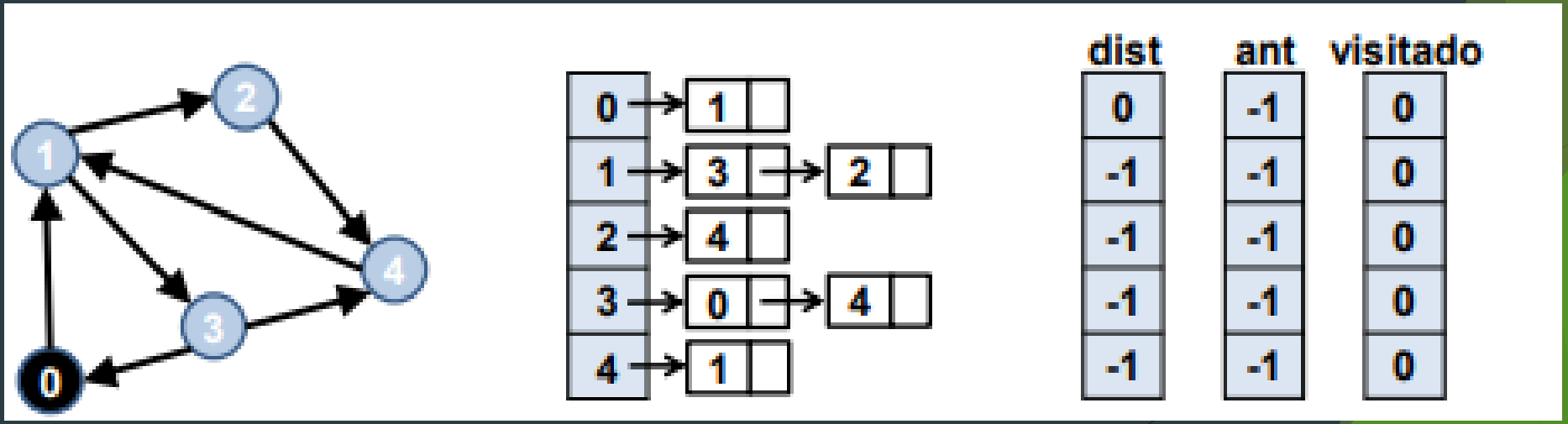
Atualizar distâncias dos vizinhos

```
for(i=0; i<gr->grau[u]; i++){ Para cada vértice vizinho
    ind = gr->arestas[u][i];
    if(dist[ind] < 0){
        dist[ind] = dist[u] + 1;
        //ou peso da aresta
        //dist[ind] = dist[u] + gr->pesos[u][i];
        ant[ind] = u;
    }else{
        if(dist[ind] > dist[u] + 1){
        //if(dist[ind] > dist[u] + 1){//ou peso da are
            dist[ind] = dist[u] + 1;
            //ou peso da aresta
            //dist[ind] = dist[u] + gr->pesos[u][i];
            ant[ind] = u;
        }
    }
}
```

Busca pelo menor caminho

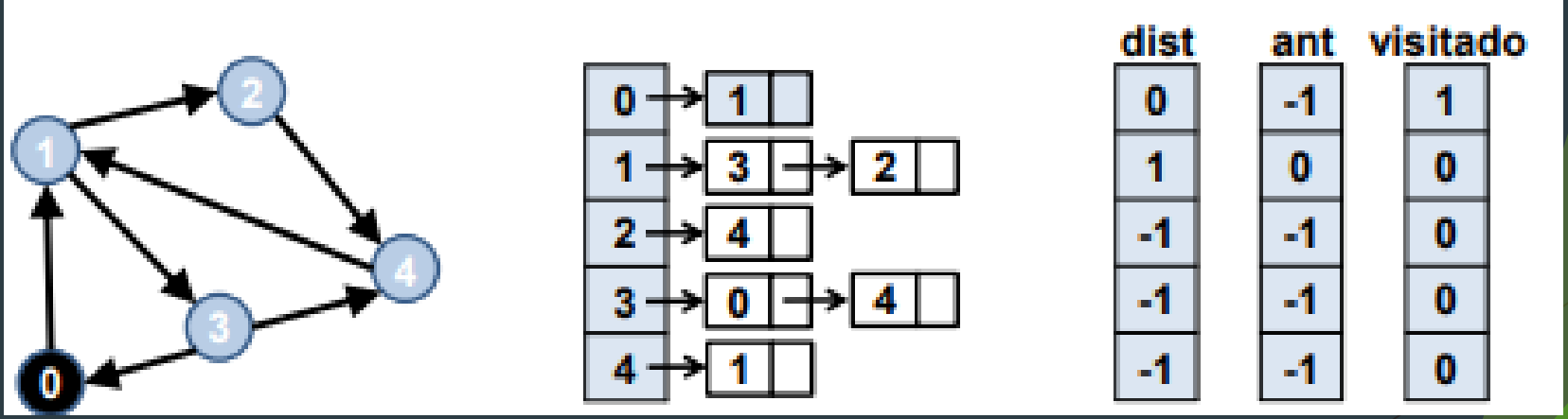
- PASSO A PASSO:

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



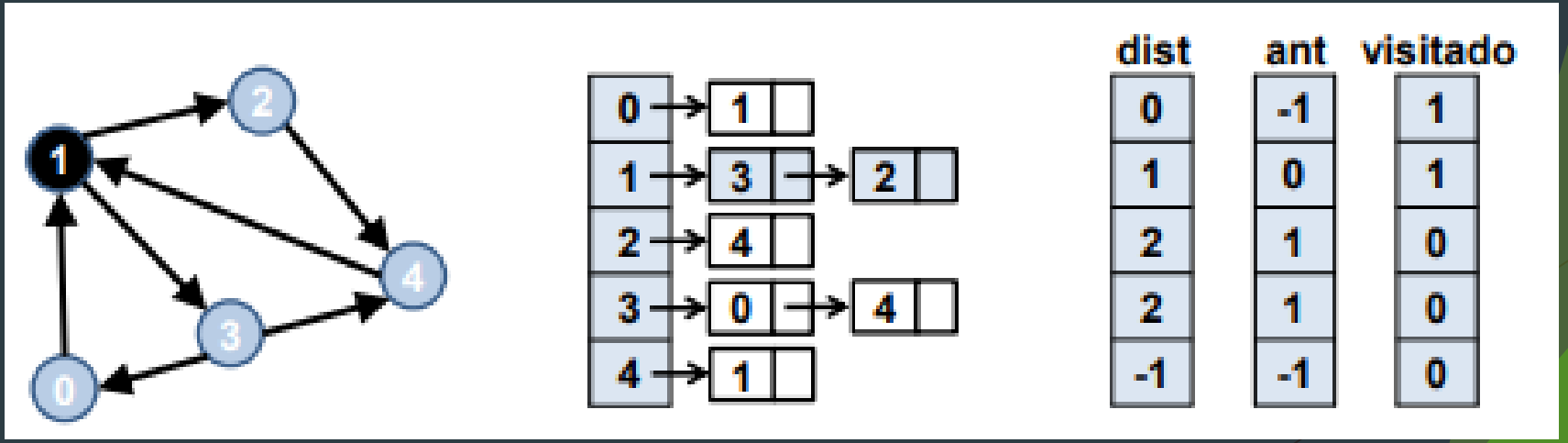
Busca pelo menor caminho

- 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)



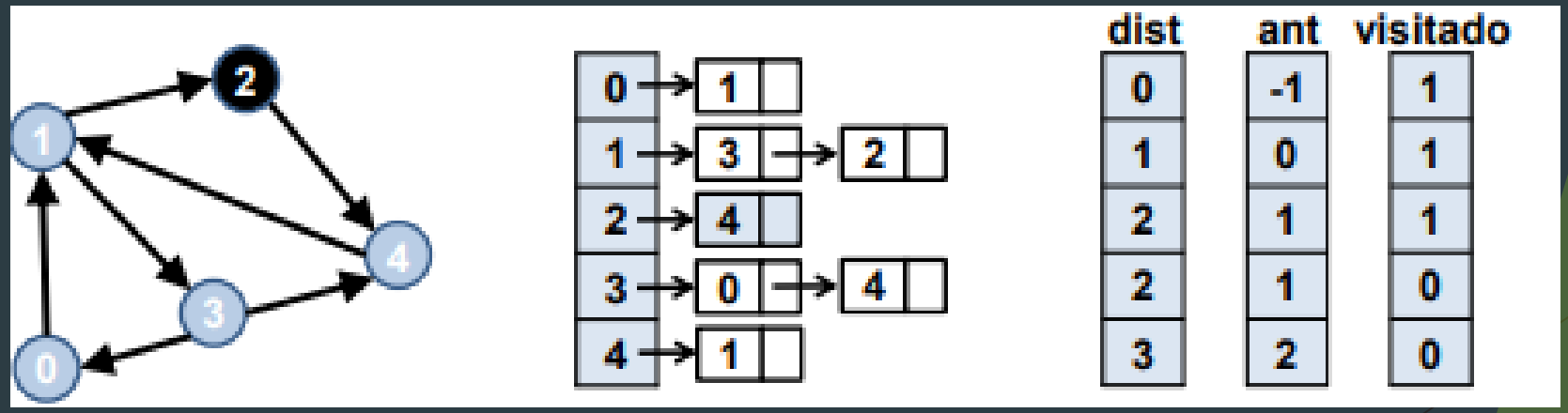
Busca pelo menor caminho

- 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)



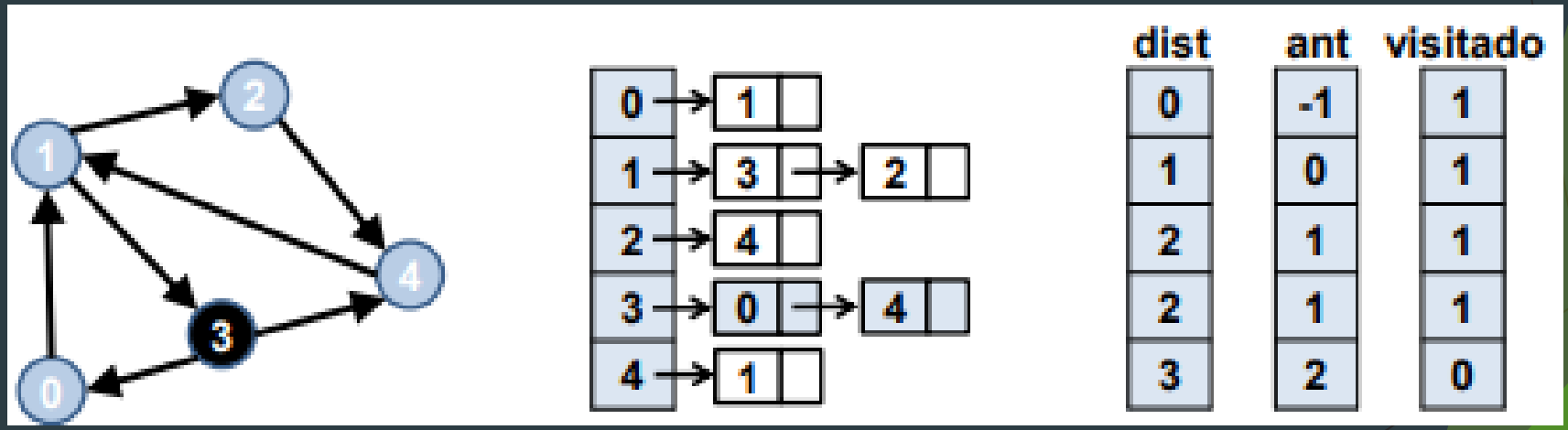
Busca pelo menor caminho

- 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)



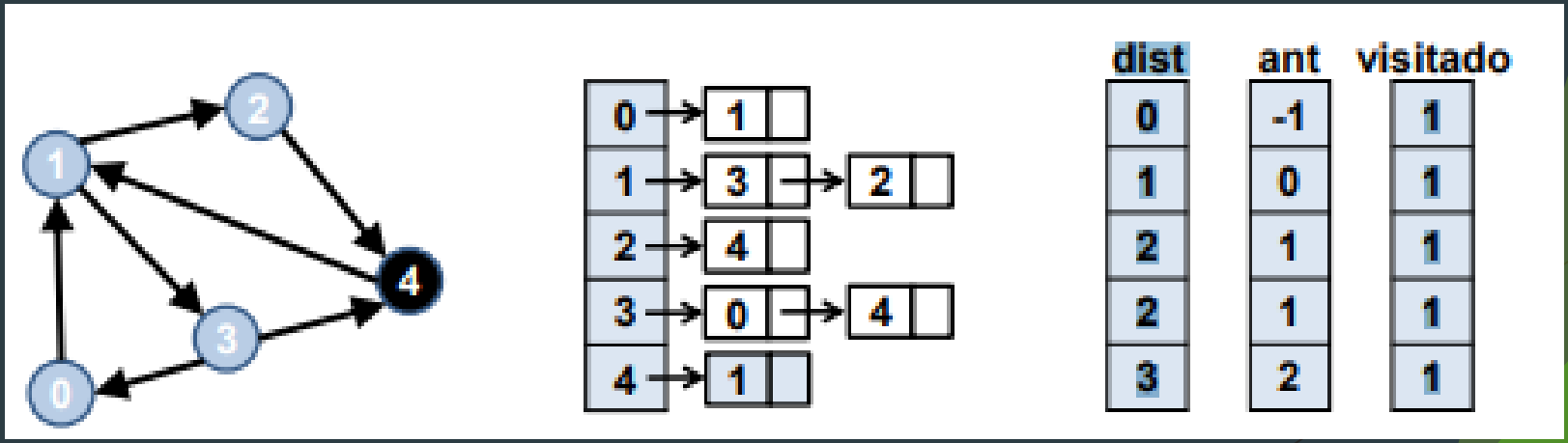
Busca pelo menor caminho

- 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)



Busca pelo menor caminho

- 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) 0 1 2 2 3 dist 1 1 1 1 1 Todos os vértices já foram visitados. Cálculo do menor caminho chegou ao fim.



Grafos

Alunos:

Betania Assunção - 11411BSI243

Joyce Emanuele - 11721BCC010

Vanclerison Souza - 11521BSI250