Estruturas de Dados

Aula 13 — Grafos

Prof[a] Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia — Campus Feira de Santana

2026

Agenda

Motivação

Definição

Representação

BFS — Busca em Largura

DFS — Busca em Profundidade

BFS vs DFS

Exercícios

Motivação

Grafos estão em todo lugar

Por quê estudar Grafos?

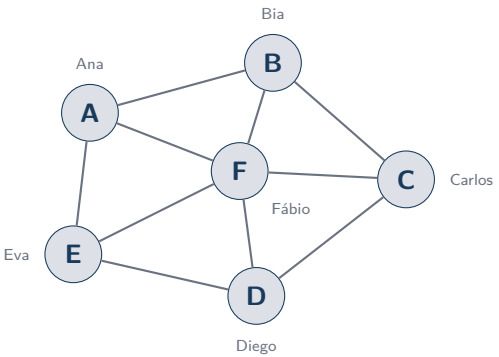
▪ Grafos modelam relações

Sempre que temos **objetos** e **conexões** entre eles, temos um grafo. É a estrutura de dados mais geral e expressiva que estudamos neste curso.

▪ Aplicações reais

- **GPS/Mapas**: cidades = vértices, estradas = arestas
- **Internet**: roteadores = vértices, cabos = arestas
- **Redes sociais**: pessoas = vértices, amizades = arestas
- **Compiladores**: dependências entre módulos
- **Redes elétricas, circuitos, pipelines**

Rede social simplificada



Definição

O que é um grafo?

Grafos — Definição Formal

Definição

Um **grafo** $G = (V, E)$ é composto por:

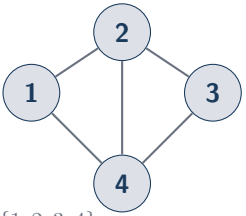
- V: conjunto de **vértices** (nós)
- $E \subseteq V \times V$: conjunto de **arestas** (edges)

Uma aresta $e = (u, v)$ conecta os vértices u e v.

Tipos principais

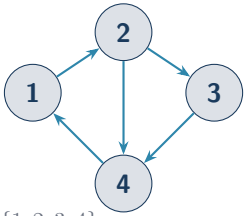
- **Não-dirigido**: arestas sem direção $(u, v) = (v, u)$
- **Dirigido** (dígrafo): arestas com direção $(u, v) \neq (v, u)$
- **Ponderado**: cada aresta tem um **peso** $w(u, v)$
- **Simples**: sem laços e sem arestas paralelas

Não-dirigido



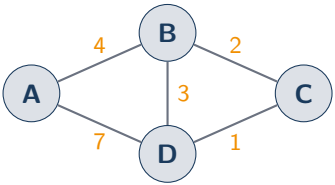
$V = \{1, 2, 3, 4\}$

Dirigido



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

Ponderado



Terminologia

Grau de um vértice

- **Grau** $d(v)$: número de arestas incidentes a v
- Em dígrafos: **grau de entrada** $d^-(v)$ e **grau de saída** $d^+(v)$
- Soma dos graus $= 2|E|$ (lema do aperto de mão)

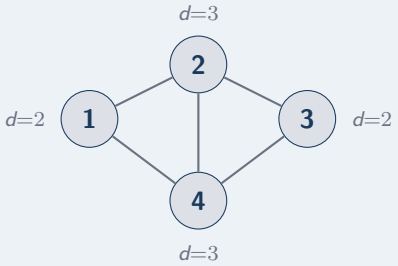
Conectividade

- **Grafo conexo**: existe caminho entre todo par de vértices
- **Componente conexa**: subgrafo conexo maximal
- **Fortemente conexo** (dígrafo): todo vértice alcança qualquer outro

Caminho e ciclo

- **Caminho**: sequência de vértices $v_1, v_2, \ldots, v_k$ onde cada par consecutivo é ligado por uma aresta
- **Ciclo**: caminho que começa e termina no mesmo vértice
- **Grafo acíclico**: sem ciclos (DAG se dirigido)

Exemplo de graus



Graus: 1→2, 2→3, 3→2, 4→3

Representação

Matriz de Adjacência e Lista de Adjacência

Representação — Matriz de Adjacência

Definição

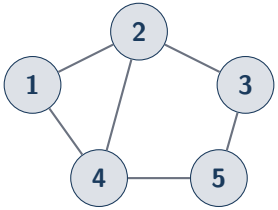
Matriz $A[n \times n]$ onde $A[i][j] = 1$ se existe aresta (i, j), 0 caso contrário. Para grafos ponderados, $A[i][j] = w(i, j)$.

Complexidades

Espaço	$O(V^2)$
Verificar aresta	$O(1)$
Listar vizinhos	$O(V)$
Inserir aresta	$O(1)$

Quando usar

Grafos **densos** ($|E| \approx |V|^2$) ou quando é preciso verificar existência de arestas frequentemente.



A

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

Representação — Lista de Adjacência

Definição

Vetor de $|V|$ listas encadeadas: `adj[v]` contém todos os vizinhos do vértice v.

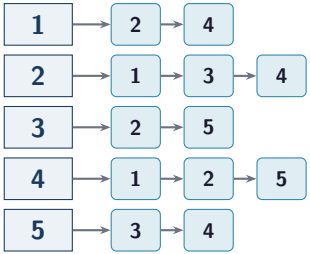
Complexidades

Espaço	$O(V+E)$
Verificar aresta	$O(\text{grau}(v))$
Listar vizinhos	$O(\text{grau}(v))$
Inserir aresta	$O(1)$

Quando usar

Grafos **esparsos** ($|E| \ll |V|^2$), que é o caso mais comum em aplicações reais.

Lista de adjacência do mesmo grafo



Representação em C — Lista de Adjacência

grafo.h — Estrutura

```c
#define MAXV 100
typedef struct No {
    int dest; //vértice destino
    int peso; //peso da aresta
    struct No *prox;
} No;
typedef struct {
    No  *adj[MAXV];//listas adj.
    int  nv; //qtde vértices
    int  na; //qtde arestas
    bool dirigido;
} Grafo;
Grafo* criarGrafo(int nv,bool dir){
  Grafo *g = malloc(sizeof(Grafo));
  g->nv = nv; g->na = 0;
  g->dirigido = dir;
  for (int i=0; i<nv; i++)
    g->adj[i] = NULL;
  return g;
}
```

grafo.c — Inserção de aresta

```c
void addAresta(Grafo *g, int u,
               int v, int w) {
    /* u -> v */
    No *novo = malloc(sizeof(No));
    novo->dest = v;
    novo->peso = w;
    novo->prox = g->adj[u];
    g->adj[u] = novo;
    g->na++;
    if (!g->dirigido) {
        /* v -> u (nao dirigido) */
        No *volta = malloc(sizeof(
            No));
        volta->dest = u;
        volta->peso = w;
        volta->prox = g->adj[v];
        g->adj[v] = volta;
    }
}
```

BFS

Busca em Largura (Breadth-First Search)

BFS — Algoritmo

Ideia

Visita todos os vértices **camada por camada**, começando na fonte s. Usa uma **fila** (FIFO) para controlar a ordem de visita.

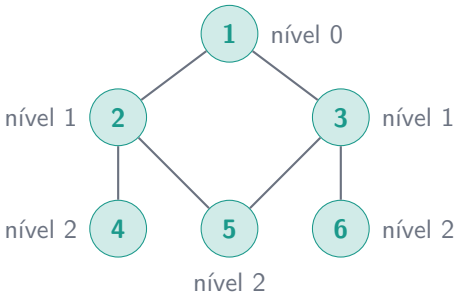
Passos

1. Marca s como visitado e enfileira s
2. Enquanto a fila não estiver vazia:
   1. Desenfileira v
   2. Para cada vizinho u de v: se u não visitado: **marcar, enfileirar**

Complexidade

$O(V + E)$ — cada vértice e cada aresta são processados uma vez.

BFS a partir do vértice 1



Ordem de visita: **1, 2, 3, 4, 5, 6**

Fila: | **1** | **2** | **3** → visita 1

BFS — Implementação em C

bfs.c — Busca em Largura

```c
void bfs(Grafo *g, int inicio) {
  bool visitado[MAXV] = {false};
  int  fila[MAXV];
  int  frente=0, fim=0, dist[MAXV];
  visitado[inicio] = true;
  dist[inicio] = 0;
  fila[fim++] = inicio;
  while (frente < fim){
    int v = fila[frente++];
    printf("Visita: %d (dist=%d)\n", v,
        dist[v]);
    No *p = g->adj[v];
    while (p != NULL){
      if (!visitado[p->dest]){
        visitado[p->dest] = true;
        dist[p->dest] = dist[v]+1;
        fila[fim++] = p->dest;
      }
      p = p->prox;
}}}
```

Propriedades da BFS

Garante a **menor distância** (em arestas) de s a todo vértice alcançável

Funciona para grafos **dirigidos** e **não-dirigidos**

Identifica **componentes conexas**

Pode reconstruir o **caminho mínimo** usando o vetor `pai[]`

Trace para o grafo anterior

Iteração	Ação
1	visita 1, enfileira 2, 3
2	visita 2, enfileira 4, 5
3	visita 3, enfileira 5 (já na fila?)
4	visita 4
5	visita 5
6	visita 6

DFS

Busca em Profundidade (Depth-First Search)

DFS — Algoritmo e Trace

Ideia

Vai o mais **fundo** possível em cada caminho antes de retroceder (*backtrack*). Pode ser implementada de forma **recursiva** ou com uma **pilha** explícita.

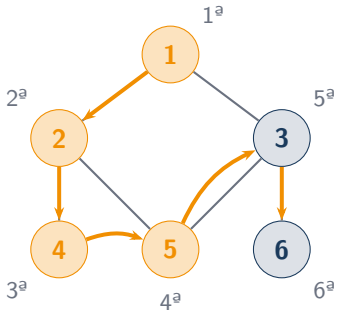
Algoritmo recursivo

1. Marca v como visitado
2. Para cada vizinho u de v:
   se u não visitado:
   DFS(u)

Complexidade

$O(V + E)$ — igual à BFS.

DFS a partir do vértice 1



Ordem: **1, 2, 4, 5, 3, 6**
Arestas em laranja = **árvore DFS**

DFS — Implementação em C

dfs.c — Busca em Profundidade

```c
bool visitado[MAXV];
void dfs(Grafo *g, int v) {
  visitado[v] = true;
  printf("Visita: %d\n", v);
  No *p = g->adj[v];
  while (p != NULL) {
    if (!visitado[p->dest]) dfs(g, p
        ->dest);
    p = p->prox; }
}
/* Chama DFS para todo vertice
   (cobre grafos desconexos) */
void dfsTotal(Grafo *g) {
  for (int i = 0; i < g->nv; i++)
    visitado[i] = false;
  for (int i = 0; i < g->nv; i++)
    if (!visitado[i]) dfs(g, i);
}
```

Aplicações da DFS

- **Detecção de ciclos**
- **Componentes fortemente conexas** (Kosaraju, Tarjan)
- **Ordenação topológica** de DAGs
- **Pontos de articulação** e pontes
- Resolver labirintos

Ordenação topológica

Em um **DAG** (grafo dirigido acíclico), a DFS produz uma ordenação topológica: os vértices são listados na ordem em que sua DFS **termina** (ordem inversa de saída). Usada em: compilação, agendamento de tarefas, dependências de pacotes.

BFS vs DFS

Comparação e aplicações

BFS vs. DFS — Comparação

Critério	BFS	DFS
Estrutura auxiliar	Fila (FIFO)	Pilha / recursão
Estratégia	Camada por camada	O mais fundo possível
Complexidade	$O(V+E)$	$O(V+E)$
Memória (pior caso)	$O(V)$ (fila grande)	$O(V)$ (pilha de recursão)
Caminho mais curto	Sim (arestas)	Não garante
Detecção de ciclos	Sim	Sim (natural)
Ordenação topológica	Não	Sim
Grafos desconexos	Precisa loop externo	Precisa loop externo

▪ Use BFS quando...

...precisar do **menor caminho** (sem pesos) ou explorar em **níveis**. Ex.: encontrar contatos de grau k em rede social.

▪ Use DFS quando...

...precisar de **ordenação topológica**, detectar **ciclos**, ou explorar **todo o grafo** rapidamente com menos memória.

Exercícios

Pratique!

Exercícios Propostos

Exercício 1 — Representação

Dado o grafo não-dirigido com vértices $\{1, 2, 3, 4, 5\}$ e arestas $\{(1,2), (1,3), (2,4), (3,4), (3,5), (4,5)\}$:

1. Construa a **matriz de adjacência**
2. Construa a **lista de adjacência**
3. Qual o grau de cada vértice?

Exercício 2 — Trace

Para o grafo acima, execute e registre passo a passo: **BFS** a partir do vértice 1 e **DFS** a partir do vértice 1
Em qual ordem os vértices são visitados em cada caso?

Exercício 3 — Implementação

Implemente em C um programa completo que:

1. Lê V e E do usuário
2. Lê as E arestas e constrói a lista de adjacência
3. Oferece menu: BFS, DFS, grau de um vértice, verificar se o grafo é conexo
4. Libera a memória ao sair

Exercício 4 — Desafio

Modifique a BFS para imprimir, além da ordem de visita, a **distância** (em número de arestas) de cada vértice à fonte. Teste com um grafo desconexo — o que acontece?

Fim da Aula 13

Dúvidas e próximos passos

Próxima Aula e Referências

Próxima aula — Caminhos Mínimos

- **Dijkstra**: menor caminho com pesos positivos
- **Bellman-Ford**: pesos negativos
- **Floyd-Warshall**: todos os pares
- Aplicação em GPS, roteamento de redes

Referências

- CORMEN et al. *Introduction to Algorithms*, 4.ª ed. Cap. 20–22.
- TENENBAUM et al. *Estruturas de Dados em C*, Cap. 8.

Resumo da Aula 13

- Grafo $G = (V, E)$: vértices + arestas
- Tipos: não-dirigido, dígrafo, ponderado
- Representação: matriz $O(V^2)$ ou lista $O(V + E)$
- BFS: menor caminho (sem peso), usa fila
- DFS: topológica, ciclos, componentes
- Ambos: $O(V + E)$

Dúvidas?

carolsoko@ifba.edu.br

IFBA – Campus Feira de Santana