Algoritmos e Estruturas de Dados II

Capítulo da Aula 14: Introdução a Grafos

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Fevereiro de 2026

1. O Despertar da Topologia: O Rompimento da Hierarquia

Até este momento, fomos tecnicamente escravos do paradigma hierárquico estrito das Árvores Binárias (mãe-filho, raiz-folha). O mundo real de Alta Complexidade e transações massivas cruzadas, entretanto, não se organiza em monarquias rígidas descendentes de parentesco estático. Ele se estrutura e respira por meio de uma malha anárquica e vital de interações fluidas caóticas não-direcionais absolutas. Para emular e acorrentar essa complexidade arquitetural sistêmica em hardware, a Computação invoca a Teoria dos **Grafos**.

Um **Grafo** é o mapeamento universal matricial $G = (V, E)$, cuja engrenagem isolada repousa integralmente sobre duas classes de abstrações atômicas:

- V: O Conjunto de **Vértices (Nodes)** $\rightarrow$ As instâncias atômicas base (indivíduos, roteadores IP pesados Core, processos suspensos do S.O.).
- E: O Conjunto de **Arestas (Edges/Arcos)** $\rightarrow$ Os barramentos rústicos de relacionamento que fundem conexões dinâmicas atritando o vértice livre u ao livre vértice v.

Esta topologia matricial cega livre fundamenta esmagadoramente as lógicas cruciais do topo da pirâmide computacional contemporânea:

- **Data Centers e Protocolos BGP:** Vértices transmutam-se cruamente em Roteadores Core, e as Arestas estouram com o peso logístico elétrico de Fibras Ópticas transoceânicas ativas.
- **Sistemas de Posicionamento Espacial Distribuído Geográfico (Routing):** O escopo onde varreduras de caminhos curtos e gargalos colossais estouram num trânsito caótico analisados por motores assutadores implacáveis na busca elástica contínua da raiz de Dijkstra.
- **Mecanismos de IA e Large Language Models (LLMs):** Malhas base semânticas extraindo grafos imaculados colossais de conhecimento.

Terminologia Essencial

- **Grau (Degree):** Número de arestas conectadas a um vértice.
- **Grafo Direcionado (Digrafo):** Arestas têm sentido ($A \rightarrow B$). Ex: Twitter (Seguir).
- **Grafo Não-Direcionado:** O sentido é bidirecional ($A - B$). Ex: Facebook (Amizade).
- **Ponderado (Weighted):** Arestas têm um "peso" ou custo. Ex: Distância em km.
- **Caminho (Path):** Sequência de vértices conectados por arestas.
- **Ciclo:** Caminho que começa e termina no mesmo vértice.

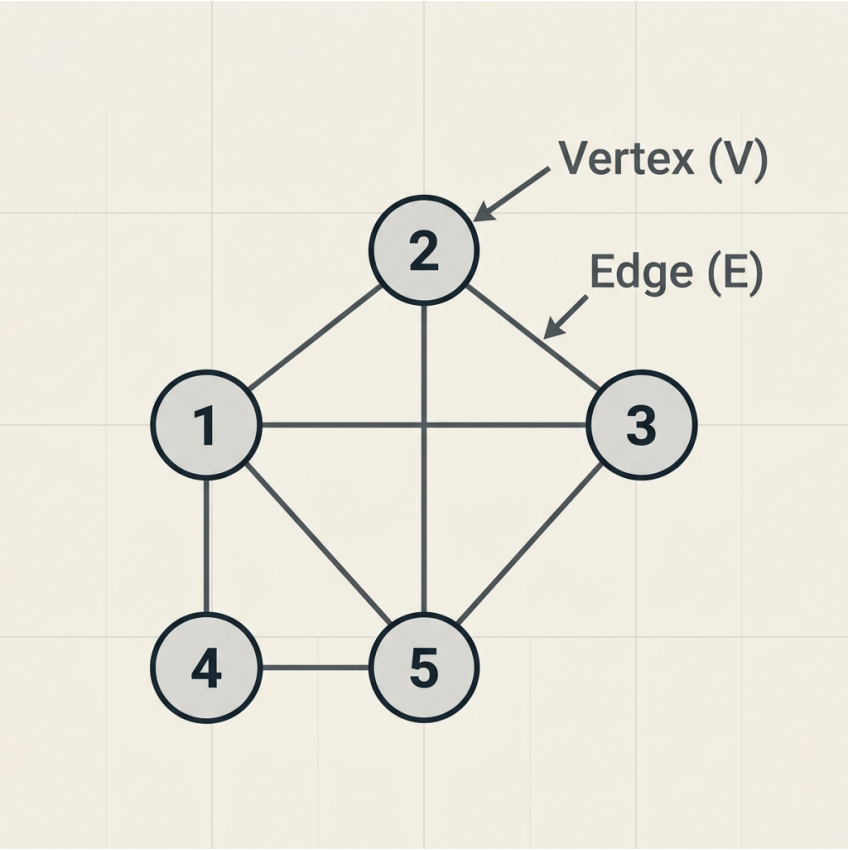

Figure 1: Grafo G(V,E): vértices (nós) e arestas (conexões) entre pares.

2. O Gargalo do Hardware: As Batalhas da Representação

Enraizar um Grafo livre dentro da restrição de uma máquina física de Von Neumann linear não é um passo trivial inócuo. Essa decisão de base reescreve atrocidades vitais nos níveis de *Big O* absolutos, definindo o sucesso ou a asfixia em gargalos da RAM cruzados por pesagens atômicas das duas grandes classes: O Universo dos **Grafos Esparsos** ($E \ll V^2$) frente ao caos do peso denso **Aglomerado** ($E \approx V^2$).

2.1. Matriz de Adjacência: A Locality Estática Blindada ($O(V^2)$)

Forjamos uma grade cruzada de memória integral alocada estaticamente limpa alfabética restrita através de uma **Matriz Dimensional Bruta bidirecional quadrada** de blocos limitados engessados M com largura massiva simétrica $V \times V$:

- $M[i][j] = \text{Peso} \rightarrow$ Se a artéria de comunicação $i \rightarrow j$ repousa ativada.
- $M[i][j] = 0$ (ou ∞) $\rightarrow$ Caso a ligação jaza na fenda obstrutiva desconexa fria.

Essa estratégia ostenta a absurda **Soberania Cache-Friendly** da CPU, garantindo o custo blindado intransponível $O(1)$ para aferir em um puro ciclo de clock cravado imediato restrito se a aresta se impõe ou inexiste. No entanto, sua arrogância falha e cobra um preço colapso-letal de *Overhead de Memória*: Oitenta por cento do tempo global do ecossistema a RAM agonizará despejando infinitos mares alocados inertes `null` de buracos engessados estéreis se for aplicada à massa vital natural de Grafos Livres Esparsos globais de `1 milhão` de nós cravando $O(V^2)$ fatalísticos estourados! Restringindo-se então rigidamente no limiar unicamente plausível: Duelos Densos Críticos exatos limítrofes.

2.2. Lista de Adjacência: A Malha Elástica Absoluta ($O(V + E)$)

A ascensão ao **Padrão Máster Universal da Indústria** se desfaz da grosseria dimensional cega para adotar alocação elástica vetorial exata alinhada e cirúrgica limpa. Constrói-se um Vetor Linear restrito e enxuto restrito no limiar universal absoluto globalizado do tamanho exato da contagem dos Vértices base cravado: contendo em cada posição passiva vital um engate orgânico livre purista vetorizado empurrando Listas soltas Dinâmicas dinamicamente povoadas atreladas englobando e capturando singularmente apenas estritamente e cruamente as ocorrências vizinhas isoladas fáticas validadas blindadas vitais reais (E), economizando atrocidades físicas!

| Vetores Focais | Implicações Reais | | :— | :— | | Suprema Eficiência de RAM pura | Degradação milimétrica a custo de *Cache Miss* cruzado | | Imbatível em travessias ($O(Grau(u))$) | Perde o fator instável estático $O(1)$ para validar se aresta solta singular reside limpa ou obedece lacuna solta ($O(Grau(u))$ tempo arrastado percorrido linear local) |

Uso de Comando Oficial: Embutido passivamente absoluto na engenharia civil em base plena para mais de 99% dos casos vitais pesados de estruturas limitadas do ecossistema computacional contíguo do mundo, abrigando tranquilamente matrizes de mais de ≈ 100.000 nós.

3. Implementação em Java

Em C++, usamos `vector<int> adj[]`. Em Java, temos um pequeno problema: arrays de genéricos (`ArrayList<Integer>[]`) causam warnings. Soluções comuns: 1. `ArrayList<ArrayList<Int` (Mais OO, levemente mais lento e verboso). 2. `List<Integer>[]` com casting (Mais compacto).

Modelo Padrão (Adjacency List)

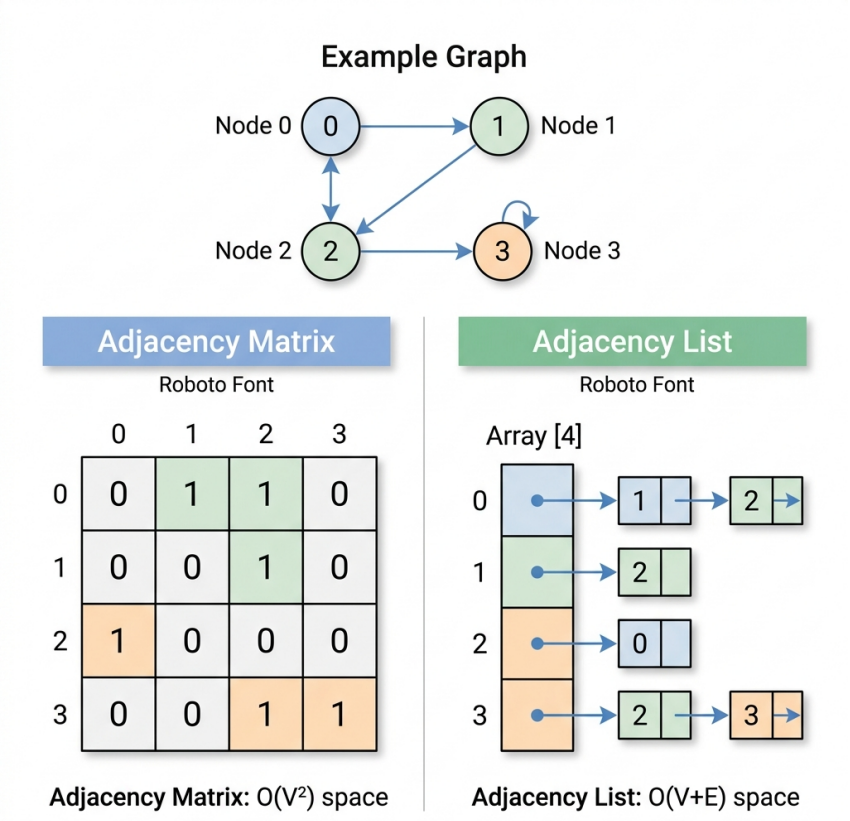

Figure 2: Matriz de adjacência vs lista de adjacência: trade-off memória e tempo de vizinhos.

```
import java.util.*;

public class Grafo {
    // Vértices rotulados de 0 a V-1
    private int V;
    private List<List<Integer>> adj;

    public Grafo(int V) {
        this.V = V;
        adj = new ArrayList<>(V);

        // Inicializa as listas vazias
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
    }

    // Adiciona aresta não-direcionada
    public void addEdge(int u, int v) {
        adj.get(u).add(v);
```

```
        adj.get(v).add(u); // Remova esta linha se for Direcionado
    }

    // Exibe o grafo
    public void printGraph() {
        for (int i = 0; i < V; i++) {
            System.out.print("Vértice " + i + ":");
            for (Integer vizinho : adj.get(i)) {
                System.out.print(" -> " + vizinho);
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Grafo g = new Grafo(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 3);

        g.printGraph();
    }
}
```

Grafo Ponderado (Com Pesos)

Se o grafo tem pesos, a lista de adjacência precisa armazenar pares `(vizinho, peso)`. Em Java, criamos uma classe `Edge`.

```
class Aresta {
    int destino;
    int peso;

    public Aresta(int d, int p) {
        this.destino = d;
        this.peso = p;
    }
}

// Na classe Grafo:
private List<List<Aresta>> adj;
```

4. Edge List (Lista de Arestas)

Às vezes (ex: Kruskal para MST, Bellman-Ford), não precisamos saber quem são os vizinhos de u. Só precisamos da lista de todas as arestas.

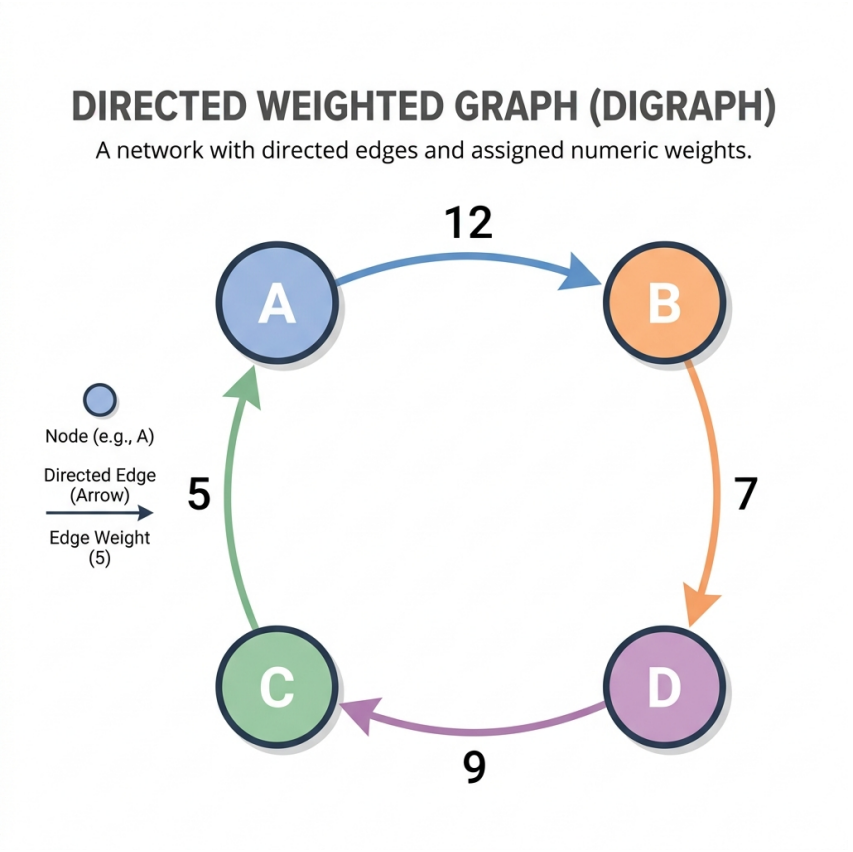

Figure 3: Grafo direcionado e ponderado: setas indicam sentido; números nas arestas são pesos.

```
class Edge implements Comparable<Edge> {
    int u, v, weight;
    // construtor...
    public int compareTo(Edge other) {
        return this.weight - other.weight;
    }
}
```

```
List<Edge> arestas = new ArrayList<>();
```

Isso simplifica a ordenação de arestas por peso.

5. Dicas de Entrada (Input)

Em problemas de juiz online, a entrada geralmente é dada assim:

```
5 6   (5 vértices, 6 arestas)
0 1
0 2
1 3
...
```

Template de Leitura:

```
Scanner sc = new Scanner(System.in);
int V = sc.nextInt();
int E = sc.nextInt();

// Inicializa List<List<Integer>>...

for(int i=0; i<E; i++) {
    int u = sc.nextInt(); // Se vértices forem 1-based, use (u-1)
    int v = sc.nextInt();
    addToGraph(u, v);
}
```

Atenção: Se os vértices forem rotulados de 1 a N, lembre-se de subtrair 1 para usar índices 0 a N-1 nos Lists.

• Teoria

Um grafo $G = (V, E)$ é uma estrutura matemática onde V é um conjunto finito de vértices e $E \subseteq V \times V$ é um conjunto de arestas. Em grafos não direcionados, $(u, v) \in E \Leftrightarrow (v, u) \in E$. Em grafos direcionados (digrafos), a ordem importa.

6. Aplicações em Engenharia de Computação

Grafos são fundamentais em:

- **Redes de Computadores:** Roteamento, protocolos de rede (OSPF, BGP), análise de topologia.
- **Sistemas Distribuídos:** Consenso distribuído, detecção de deadlock, ordenação de eventos.
- **Compiladores:** Análise de fluxo de dados, otimização de código, dependências entre variáveis.
- **Bancos de Dados:** Modelagem de relacionamentos (ER), otimização de consultas, transações.
- **Inteligência Artificial:** Redes neurais (grafos de computação), busca em espaço de estados, planejamento.
- **Sistemas Operacionais:** Escalonamento de processos, gerenciamento de recursos, detecção de ciclos.

▷ Exemplo

Em um sistema de GPS, o mapa rodoviário é modelado como um grafo ponderado onde vértices são cruzamentos e arestas são estradas com pesos representando distância ou tempo de viagem. Algoritmos de caminho mínimo (Dijkstra) são essenciais para calcular rotas otimizadas.

7. Exercícios

7.1. Exercícios Conceituais

1 Defina formalmente um grafo direcionado e um grafo não direcionado. Dê exemplos de problemas reais que são naturalmente modelados como cada tipo.

2 Compare matriz de adjacência com lista de adjacência em termos de:

- Complexidade de memória.
- Tempo para verificar se existe aresta (u, v).
- Tempo para iterar sobre vizinhos de um vértice.

Quando cada representação é preferível?

3 Explique a diferença entre grafo conexo, grafo fortemente conexo (para digrafos) e grafo completo. Dê exemplos práticos de cada um.

7.2. Exercícios Analíticos

1 Para um grafo com $V = 1000$ vértices e $E = 5000$ arestas:

- Calcule o espaço usado por matriz de adjacência vs lista de adjacência (assuma 4 bytes por inteiro).
- Determine qual representação é mais eficiente em memória.

2 Desenhe um grafo direcionado com 5 vértices que seja fortemente conexo. Depois, remova uma aresta e verifique se ainda é fortemente conexo.

3 Para um grafo completo K_n com n vértices, calcule:

- Número de arestas (não direcionado).
- Número de arestas (direcionado).
- Grau de cada vértice.

7.3. Exercícios de Programação

1 Implemente uma classe `Grafo` em Java que suporte:

- Construção a partir de número de vértices.
- Adição de arestas (direcionadas e não direcionadas).
- Adição de arestas ponderadas.
- Conversão entre matriz de adjacência e lista de adjacência.
- Cálculo do grau de um vértice.
- Verificação de existência de aresta.

2 Escreva um programa que leia um grafo de um arquivo (formato: primeira linha com V e E, seguido de E linhas com pares de vértices) e:

- Imprima a representação em lista de adjacência.
- Imprima a representação em matriz de adjacência.
- Calcule estatísticas (número de componentes conexos, vértice com maior grau, etc.).

3 Resolva problemas básicos de juiz online envolvendo grafos, como:

- Verificação de conectividade.
- Contagem de componentes conexos.
- Detecção de grafo bipartido simples.