Algoritmos e Estruturas de Dados II

Capítulo da Aula 18: Árvore Geradora Mínima (MST)

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

Fevereiro de 2026

1. Conectando Cidades com Custo Mínimo

Imagine que você é um engenheiro de telecomunicações. Você precisa passar cabos de fibra óptica conectando N cidades. Você sabe o custo para cabear entre qualquer par de cidades. **Objetivo:** Conectar **todas** as cidades gastando o mínimo possível.

A solução é uma subestrutura do grafo original que: 1. Contém todos os vértices. 2. É conexa. 3. Não tem ciclos (é uma árvore). 4. A soma dos pesos das arestas é mínima.

Isso é uma **MST**. Se o grafo tem V vértices, a MST terá exatamente $V - 1$ arestas.

2. Algoritmo de Kruskal (Abordagem Gulosa nas Arestas)

O algoritmo de Joseph Kruskal é intuitivo: "Sempre tente pegar a aresta mais barata disponível, desde que ela não forme um ciclo".

Passo a Passo

1. Crie uma lista com **todas** as arestas do grafo. 2. **Ordene** a lista por peso crescente. 3. Itere pela lista ordenada:

- Seja a aresta (u, v) com peso w.
- Se u e v já estão conectados (estão no mesmo conjunto), discarte a aresta (formaria ciclo).
- Se não, **adicione** a aresta à MST e **una** os conjuntos de u e v.

Para verificar conectividade e unir conjuntos rapidamente, usamos a estrutura **Union-Find (DSU)**.

Implementação Java (Kruskal)

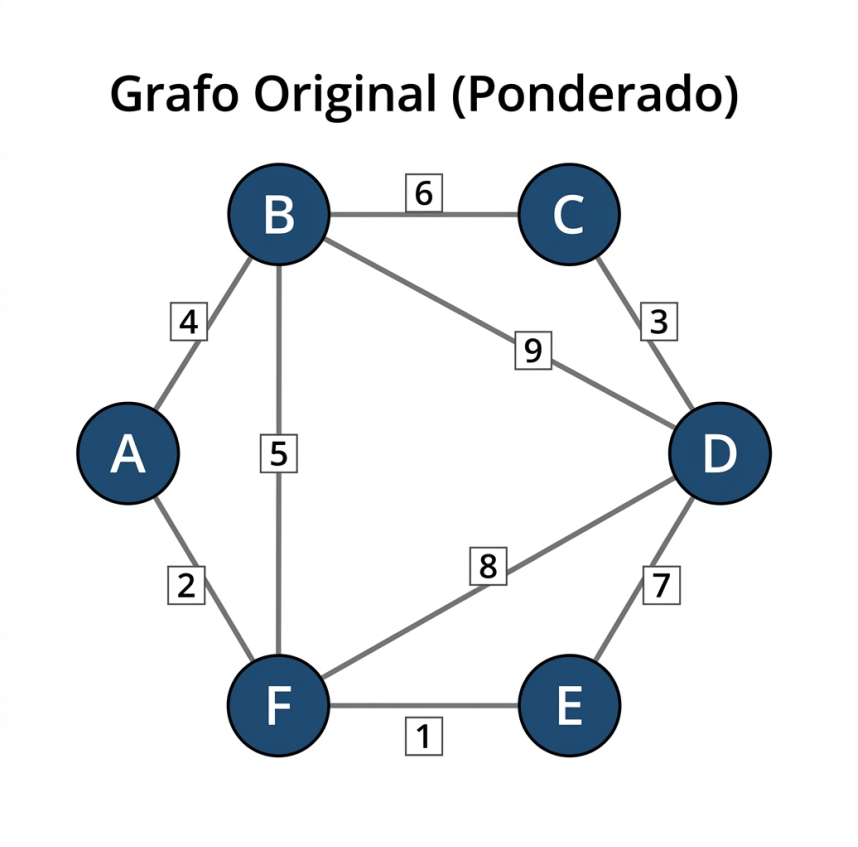

Figure 1: MST: subgrafo que conecta todos os vértices com soma mínima dos pesos das arestas.

```java
import java.util.*;

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

public class KruskalMST {

    // Union-Find Simplificado (veja tópico 19 para completo)
    static int[] parent;
    static int find(int i) {
        if (parent[i] == i) return i;
        return parent[i] = find(parent[i]);
    }
```

```
    static void union(int i, int j) {
        int rootI = find(i);
        int rootJ = find(j);
        if (rootI != rootJ) parent[rootI] = rootJ;
    }

    public static int kruskal(int V, List<Edge> edges) {
        Collections.sort(edges); // O(E log E)

        parent = new int[V];
        for(int i=0; i<V; i++) parent[i] = i;

        int mstWeight = 0;
        int edgesCount = 0;

        for (Edge e : edges) {
            if (find(e.u) != find(e.v)) {
                union(e.u, e.v);
                mstWeight += e.weight;
                edgesCount++;
            }
        }

        if (edgesCount != V - 1) return -1; // Grafo desconexo
        return mstWeight;
    }
}
```

Complexidade: $O(E \log E)$ devido à ordenação.

3. Algoritmo de Prim (Abordagem Gulosa nos Vértices)

O algoritmo de Robert Prim (e Jarník) "cresce" a MST a partir de um vértice inicial, similar ao Dijkstra.

Passo a Passo

1. Escolha um vértice inicial arbitrário (ex: 0). 2. Adicione todas as arestas dele numa **Priority Queue** (Min-Heap). 3. Marque o vértice como visitado. 4. Enquanto a PQ não estiver vazia:

- Remova a aresta (u, v) de menor peso.
- Se v já foi visitado, ignore.
- Se não, esta aresta faz parte da MST! Adicione o custo.
- Marque v como visitado e adicione todas as arestas saindo de v para a PQ.

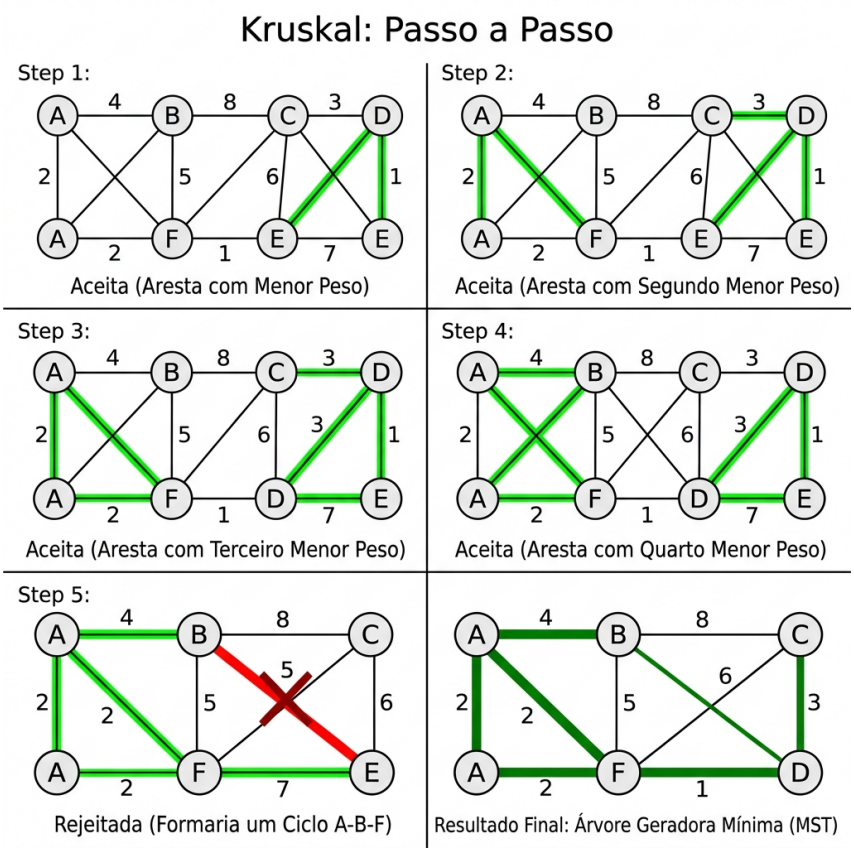

Figure 2: Kruskal: ordenar arestas por peso; acrescentar se não formar ciclo (Union-Find).

Implementação Java (Prim)

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

public static int prim(int V, List<List<Pair>> adj) {
    boolean[] visited = new boolean[V];
    PriorityQueue<Pair> pq = new PriorityQueue<>();

    // Começa do 0, custo 0
    pq.add(new Pair(0, 0));

    int mstWeight = 0;
    int nodesCount = 0;

    while (!pq.isEmpty()) {
        Pair current = pq.poll();
        int u = current.v;
```

```
            if (visited[u]) continue; // Lazy deletion

            visited[u] = true;
            mstWeight += current.weight;
            nodesCount++;

            for (Pair neighbor : adj.get(u)) {
                if (!visited[neighbor.v]) {
                    pq.add(new Pair(neighbor.v, neighbor.weight));
                }
            }
        }

        if (nodesCount != V) return -1;
        return mstWeight;
    }
```

Complexidade: $O(E \log V)$ usando Binary Heap.

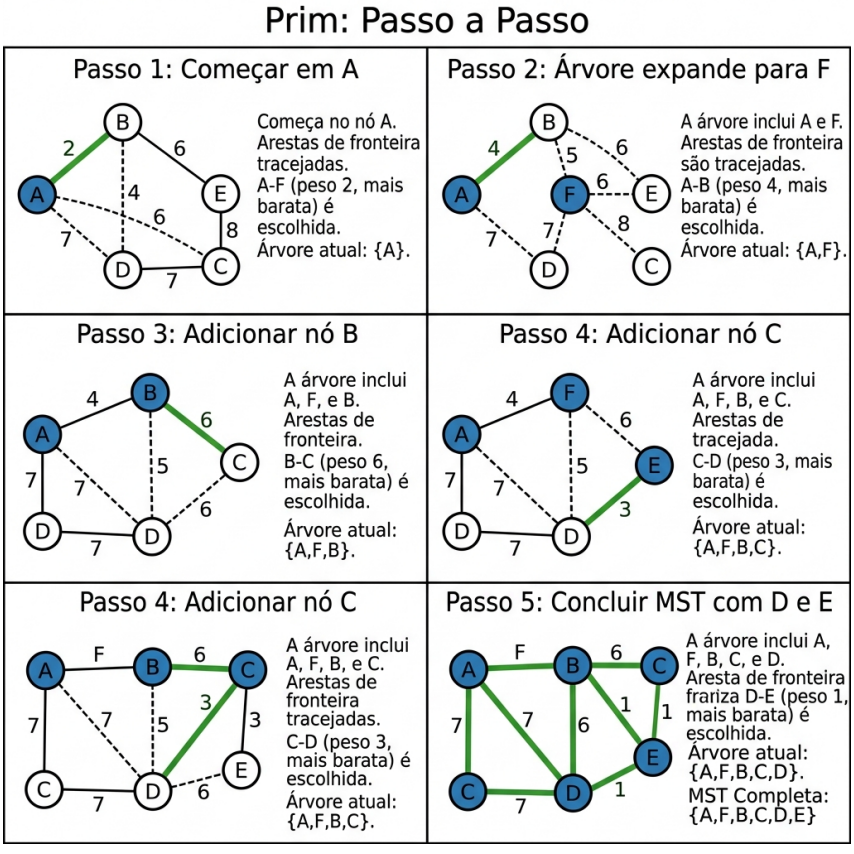

Figure 3: Prim: crescer uma árvore a partir de um vértice; sempre a aresta de menor peso para fora.

4. Kruskal vs Prim: O Veredito

| Característica | Kruskal | Prim | | :— | :— | :— | | **Lógica** | Ordena arestas. | Cresce árvore (Priority Queue). | | **Estrutura Auxiliar** | Union-Find (DSU). | Heap (Priority Queue) + Visited array. | | **Melhor para** | Grafos **Esparsos** (poucas arestas). | Grafos **Densos** (muitas arestas). | | **Facilidade** | Muito fácil se tiver DSU pronto. | Fácil se souber Dijkstra. |

Em competições, **Kruskal** é geralmente o favorito porque o DSU é muito curto de implementar e a lógica de ordenar arestas é menos propensa a bugs do que gerenciar a PQ.

• Teoria

Uma MST de um grafo conexo com V vértices sempre tem exatamente $V-1$ arestas. Se o grafo não for conexo, cada componente conexo terá sua própria MST, e o número total de arestas será $V-k$, onde k é o número de componentes conexos.

5. Aplicações em Engenharia de Computação

- **Redes de Computadores:** Projetar topologias de rede com custo mínimo de cabos/conexões.
- **Telecomunicações:** Planejamento de infraestrutura de fibra óptica conectando cidades.
- **Clustering:** Agrupamento hierárquico de dados usando MST como estrutura intermediária.
- **Visão Computacional:** Segmentação de imagens usando MST para conectar pixels similares.

6. Exercícios

6.1. Exercícios Conceituais

1 Prove que uma MST sempre existe em um grafo conexo. Por que a MST é única se todos os pesos são distintos?

2 Compare Kruskal e Prim em termos de estrutura de dados auxiliar, complexidade e quando cada um é mais eficiente.

3 Explique por que algoritmos gulosos funcionam para MST. Qual é a propriedade que garante a otimalidade?

6.2. Exercícios Analíticos

1 Dado o grafo com arestas e pesos:

```
(0,1): 4, (0,2): 1, (1,2): 2, (1,3): 5, (2,3): 3
```

Construa a MST usando Kruskal e Prim passo a passo.

2 Para um grafo completo K_n com n vértices, quantas arestas existem? Quantas arestas tem a MST? Qual é a complexidade de Kruskal e Prim neste caso?

3 Analise a complexidade de Kruskal quando usamos:

- Union-Find sem otimizações.
- Union-Find com path compression e union by rank.

6.3. Exercícios de Programação

1 Implemente Kruskal e Prim completos. Compare o desempenho em grafos esparsos ($E \approx V$) e densos ($E \approx V^2$).

2 Resolva problemas de juiz online relacionados a MST, como:

- Encontrar MST em grafos com pesos.
- Verificar se uma aresta específica faz parte de alguma MST.
- Encontrar a segunda melhor MST.

3 Estenda a implementação de MST para detectar se o grafo é conexo. Se não for, encontre MSTs para cada componente conexo.