Estruturas de Dados

Aula 15 — Árvore Geradora Mínima

Prof Ana Carolina Sokolonski

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

2026

Agenda

Motivação e Definição

Algoritmo de Kruskal

Estrutura Union-Find

Algoritmo de Prim

MST vs Caminho Mínimo

Implementação em C

Exercícios

Motivação e Definição

Minimum Spanning Tree — MST

Por quê Árvore Geradora Mínima?

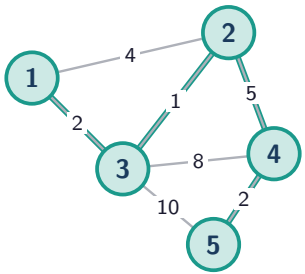
Problema real

Uma empresa precisa cabrear n escritórios com fibra óptica. Existem possíveis conexões diretas entre pares de escritórios, cada uma com um custo. Como conectar **todos os escritórios** gastando o **mínimo total de cabo**?

Aplicações

- Redes de telecomunicação e energia elétrica
- Pipelines de água/gás
- Agrupamento (*clustering*) em aprendizado de máquina
- Roteamento em redes *broadcast*
- Aproximação do Problema do Caixeiro Viajante

Grafo com custos de conexão



Arestas em verde = MST custo total = **10**

Definição Formal de MST

Grafo de entrada

Seja G **conexo** e **ponderado**, $G = (V, E, w)$
V = conjunto de vértices
E = conjunto de arestas (não-dirigidas)
$w : E \to \mathbb{R}$ = função de peso

Árvore Geradora

Um subgrafo $T \subseteq E$ tal que:
T **conecta** todos os vértices de V, sem ciclos.
$|T| = |V| - 1$ arestas

Árvore Geradora *Mínima*

A árvore geradora T^* que **minimiza**:

$$w(T^*) = \sum_{(u,v) \in T^*} w(u, v)$$

Propriedades importantes

- Uma MST sempre existe se G for conexo
- Se todos os pesos são distintos, a MST é **única**
- Se pesos se repetem, pode haver **múltiplas MSTs** com mesmo custo
- Remoção de qualquer aresta de T^* desconecta o grafo

Propriedade do corte

Para qualquer corte $(S, V \setminus S)$ do grafo, a aresta de **menor peso** que cruza o corte **pertence a alguma MST**. Esta propriedade fundamenta os algoritmos gulosos.

Algoritmo de Kruskal

Arestas em ordem crescente de peso

Kruskal — Estratégia Gulosa

Ideia central

Ordenar **todas as arestas** por peso crescente. Percorrer a lista e, para cada aresta (u, v):
Aceitar: se u e v estão em **componentes distintos** (não forma ciclo).
Rejeitar: se u e v já estão **no mesmo componente** (formaria ciclo).

Pseudocódigo

Entrada: $G = (V, E, w)$ conexo não-dirigido

1. Ordenar E por peso crescente
2. Inicializar Union-Find com $|V|$ componentes
3. Para cada $(u, v) \in E$ (em ordem):
   se $\textsc{Find}(u) \neq \textsc{Find}(v)$:
   Adicionar (u, v) à MST
   $\textsc{Union}(u, v)$
4. Retornar MST

Complexidade

- Ordenação das arestas: $\mathcal{O}(E \log E)$
- Union-Find com compressão e rank: $\mathcal{O}(E \cdot \alpha(V)) \approx \mathcal{O}(E)$
- **Total:** $\mathcal{O}(E \log E)$
  equivalente a $\mathcal{O}(E \log V)$
  pois $E \leq V^2$

Quando usar?

Kruskal é eficiente para **grafos esparsos** ($E \ll V^2$). A ordenação domina o tempo total. Muito usado quando as arestas já chegam ordenadas ou são poucas.

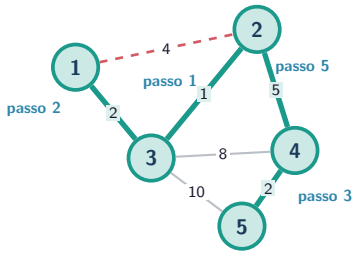
Kruskal — Passo a Passo

Arestas ordenadas: $(2,3)$:1 $(1,3)$:2 $(4,5)$:2 $(1,2)$:4 $(2,4)$:5 $(3,4)$:8 $(3,5)$:10

Aresta	Peso	Ação	Custo MST
$(2,3)$	1	Aceita	1
$(1,3)$	2	Aceita	3
$(4,5)$	2	Aceita	5
$(1,2)$	4	Ciclo!	5
$(2,4)$	5	Aceita	10
$(3,4)$	8	*(pronto)*	10

MST $= \{(2,3),(1,3),(4,5),(2,4)\}$

Custo total = 10



Tracejado = aresta rejeitada (formaria ciclo)

Estrutura Union-Find

Detecção eficiente de ciclos

O que é?

Estrutura de dados para gerenciar uma coleção de **conjuntos disjuntos**. Suporta duas operações:

- `find(x)`: retorna o **representante** (raiz) do conjunto de x
- `union(x,y)`: **une** os conjuntos de x e y

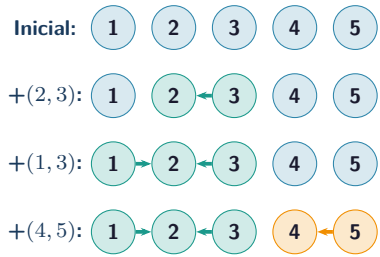
Duas otimizações essenciais

1. Compressão de caminho: em `find`, aponta todos os nós diretamente para a raiz.

2. União por rank: sempre anexa a árvore menor sob a maior. Garante altura $\mathcal{O}(\log n)$.

Com ambas: operações em $\mathcal{O}(\alpha(n)) \approx \mathcal{O}(1)$.

Evolução dos componentes em Kruskal



Union-Find — Implementação em C

```c
#define MAX 100
int pai[MAX], rank[MAX];
/* Inicializa: cada vértice é seu próprio representante */
void inicializar(int n) {
    for(int i=0; i<n; i++){
        pai[i]  = i;
        rank[i] = 0;
    }
}
/* Find com compressão de caminho */
int find(int x) {
    if(pai[x] != x) pai[x]=find(pai[x]);//compressão
    return pai[x];
}
/* Union por rank */
void unite(int x, int y) {
    int rx = find(x), ry = find(y);
    if (rx == ry) return; //já no mesmo conjunto
    if (rank[rx]<rank[ry])  pai[rx]=ry;
    else if(rank[rx]>rank[ry]) pai[ry]=rx;
    else {pai[ry] = rx; rank[rx]++;}
}
```

Algoritmo de Prim

Crescimento vértice a vértice

Prim — Estratégia Gulosa

Ideia central

Cresce uma única árvore a partir de um vértice inicial. Em cada passo, adiciona a aresta de **menor peso** que conecta um vértice **dentro** da árvore a um vértice **fora** dela.

Pseudocódigo

1. $chave[v] \leftarrow \infty$ para todo v; $chave[s] \leftarrow 0$
2. Inserir todos os vértices na fila de prioridade
3. Enquanto fila não vazia:
   $u \leftarrow$ ExtractMin(fila)
   Para cada vizinho v de u:
   se $v \notin$ MST e $w(u, v) < chave[v]$:
   $chave[v] \leftarrow w(u, v)$
   $pai[v] \leftarrow u$

Complexidade

- Com vetor simples (adjacência): $\mathcal{O}(V^2)$
- Com fila de prioridade + lista: $\mathcal{O}(E \log V)$
- Com Heap de Fibonacci: $\mathcal{O}(E + V \log V)$

Kruskal vs Prim

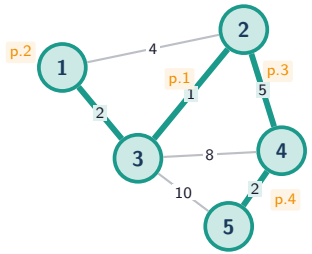
- **Kruskal**: melhor para grafos **esparsos** (E pequeno)
- **Prim**: melhor para grafos **densos** ($E \approx V^2$)
- Ambos produzem uma MST (possivelmente diferente se houver empates de peso)

Prim — Passo a Passo (início no vértice 1)

Passo	1	2	3	4	5	Adicionado
Inicial	**0**	∞	∞	∞	∞	vértice 1
1	–	4	**2**	∞	∞	(1,3):2
2	–	**1**	–	8	10	(3,2):1
3	–	–	–	**5**	10	(2,4):5
4	–	–	–	–	**2**	(4,5):2

chave$[v]$: menor peso de aresta que conecta v à árvore.

MST: $\{(1,3),(3,2),(2,4),(4,5)\}$, custo = **10**



Mesma MST que Kruskal (pesos distintos)

MST vs Caminho Mínimo

São objetivos diferentes!

MST $\neq$ Caminho Mínimo

Confusão frequente

MST e caminho mínimo usam a mesma estrutura (grafos ponderados) e ambos são gulosos, mas têm **objetivos opostos**:

MST: minimiza o **peso total da árvore** (todos os vértices conectados ao menor custo)

Dijkstra/BF: minimiza a **distância de s a cada v** (caminhos individuais, não uma árvore global)

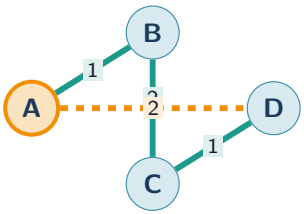
Exemplo concreto

$V = \{1, 2, 3, 4\}$, arestas:
$1-2{:}1,\ \ 1-3{:}2,\ \ 2-4{:}10,\ \ 3-4{:}3,\ \ 1-4{:}2$

MST: $\{(1,2){:}1,\ (1,3){:}2,\ (1,4){:}2\}$ custo 5

Caminho $1 \to 4$: $1 \to 4$ direto $= \mathbf{2}$
Na MST, $1 \to 4$ também usa $(1,4){:}2 = 2$

Quando a aresta mais curta NÃO está na MST



MST: A-B-C-D = 4

- **MST**: $\{AB{:}1,\ BC{:}2,\ CD{:}1\}$ custo $= 4$
- **Caminho mínimo** $A \to D$: $A \to D$ direto $= \mathbf{2}$
- Aresta $AD{:}2$ **não está** na MST pois adicioná-la criaria ciclo, mas é o caminho mais curto!

Tabela Comparativa

Característica	Kruskal	Prim	Dijkstra
Objetivo	MST	MST	Caminho mínimo
Estrutura central	Union-Find	Fila de prioridade	Fila de prioridade
Ordem de escolha	arestas globais	arestas do corte	vértices mais próximos
Complexidade	$\mathcal{O}(E \log E)$	$\mathcal{O}(E \log V)$	$\mathcal{O}(E \log V)$
Pesos negativos	Sim (MST)	Sim (MST)	Não (Dijkstra)
Resultado	$\lvert V \rvert - 1$ arestas	$\lvert V \rvert - 1$ arestas	$d[v]$ para cada v
Melhor para	Esparso	Denso	Fonte única

Tabela Comparativa

▪ Pergunta-chave

Quero conectar todos os vértices com custo mínimo?
⇒ Use **MST** (Kruskal ou Prim)

Quero ir de A a B pelo caminho mais barato?
⇒ Use **Dijkstra** (ou Bellman-Ford)

▪ MST $\neq$ Árvore de Caminhos Mínimos

A árvore de caminhos mínimos de Dijkstra a partir de s minimiza $d[s \to v]$ individualmente. A MST minimiza $\sum w(T)$ globalmente. Os dois resultados **geralmente diferem**.

Implementação em C

Kruskal e Prim

Kruskal — Implementação Completa

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 200
typedef struct{
    int u, v, w;
}Aresta;
int pai[MAX], rnk[MAX];
int find(int x){
    return (pai[x] == x) ? x : (pai[x] = find(pai[x]));
}
void unite(int x, int y) {
    int rx = find(x), ry = find(y);
    if (rx == ry) return;
    if (rnk[rx] < rnk[ry]) pai[rx] = ry;
    else if (rnk[rx] > rnk[ry]) pai[ry] = rx;
    else { pai[ry] = rx; rnk[rx]++; }
}
int cmp(const void *a, const void *b){
    return ((Aresta*)a)->w - ((Aresta*)b)->w;
}
//continua no próximo slide...
```

Kruskal — Implementação Completa

```c
int kruskal(Aresta arestas[], int E, int V) {
    qsort(arestas, E, sizeof(Aresta), cmp);
    for(int i=0; i<V; i++) {
        pai[i] = i; rnk[i] = 0;
    }
    int custo = 0, cnt = 0;
    for(int i=0; i<E && cnt<V-1; i++){
        int u = arestas[i].u, v = arestas[i].v;
        if(find(u) != find(v)){
            unite(u, v);
            printf("MST: (%d,%d) peso %d\n", u, v, arestas[i].w);
            custo += arestas[i].w; cnt++;
        }
    }
    return custo;
}
```

Prim — Implementação $\mathcal{O}(V^2)$

```c
#include <stdio.h>
#include <limits.h>
#define MAX 100
#define INF INT_MAX
int G[MAX][MAX];   /* matriz de adjacência (0 = sem aresta) */
/* Retorna o vértice de menor chave fora da MST */
int minChave(int chave[], int inMST[], int V) {
    int min = INF, idx = -1;
    for (int v = 0; v < V; v++)
        if (!inMST[v] && chave[v] < min) {
            min = chave[v];
            idx = v;
        }
    return idx;
}
int prim(int V) {
    int chave[MAX], pai[MAX];
    int inMST[MAX];
    for (int i = 0; i < V; i++) {
        chave[i] = INF; inMST[i] = 0;
    }
//continua no próximo slide...
```

Prim — Implementação $\mathcal{O}(V^2)$

```
1
2       chave[0] = 0; pai[0] = -1;
3       int custo = 0;
4       for (int cnt = 0; cnt < V-1; cnt++) {
5           int u = minChave(chave, inMST, V);
6           inMST[u] = 1;
7           for (int v = 0; v < V; v++) {
8               if (G[u][v] && !inMST[v] && G[u][v] < chave[v]) {
9                   chave[v] = G[u][v];
10                  pai[v]   = u;
11              }
12          }
13      }
14      for (int v = 1; v < V; v++) {
15          printf("MST: (%d,%d) peso %d\n", pai[v], v, chave[v]);
16          custo += chave[v];
17      }
18      return custo;
19  }
```

Exercícios

Pratique os algoritmos

Exercícios

1. **Trace manual (Kruskal):** Aplique o algoritmo de Kruskal no grafo abaixo. Mostre a lista de arestas ordenadas, qual Union-Find aceita/rejeita e o custo total da MST.

   Arestas: (A, B):3, (A, C):1, (B, C):2, (B, D):4, (C, D):6, (C, E):5, (D, E):2

2. **Trace manual (Prim):** Aplique o algoritmo de Prim no mesmo grafo, partindo do vértice A. Monte a tabela de chave$[v]$ e pai$[v]$ a cada passo. Verifique se a MST obtida tem o mesmo peso total que no exercício anterior.

3. **Implementação:** Adapte o código de Kruskal para imprimir a lista de arestas da MST na ordem em que foram adicionadas e o custo total. Teste com o grafo da Aula 15 de Caminhos Mínimos.

4. **Análise:** Dado o grafo da questão 1, execute o algoritmo de Dijkstra a partir de A e determine se o caminho mínimo de A a E coincide com o caminho entre esses vértices na MST. Explique a diferença.

Fim da Aula 15

Dúvidas e próximos passos

Encerrando a Aula 15

Resumo da aula

- **MST**: conecta todos os vértices com custo total mínimo
- **Kruskal**: ordena arestas + Union-Find, $\mathcal{O}(E \log E)$, ideal para grafos esparsos
- **Union-Find**: compressão de caminho + rank $\Rightarrow \mathcal{O}(\alpha(n))$
- **Prim**: cresce a árvore pelo corte mínimo, $\mathcal{O}(V^2)$ ou $\mathcal{O}(E \log V)$
- **MST** $\neq$ **caminho mínimo**: objetivos, estruturas e resultados distintos

Próxima aula — Fluxo em Redes

- Redes de fluxo: capacidade e fluxo
- Algoritmo de Ford-Fulkerson
- Método de Edmonds-Karp (BFS + FF)
- Teorema Max-Flow Min-Cut
- Aplicações: bipartite matching, roteamento

carolsoko@ifba.edu.br

Dúvidas? Mande uma mensagem!