🔧 Construindo a Ferrovia Mágica

Imagina que temos várias cidades (os **pinos**) e queremos conectá-las com trilhos de trem (**fios**). Mas trilhos são caros, então precisamos gastar o mínimo possível de material para ligar todas as cidades.

O problema é: **como conectar todas elas sem desperdiçar trilhos?**

🌟 Regras do Jogo

Para fazer isso, temos que seguir três regras importantes:

1. **Todas as cidades precisam estar conectadas no final** (senão algumas ficam isoladas).
2. **Não podemos fazer caminhos circulares** (senão estamos dando voltas desnecessárias).
3. **Queremos gastar o menor número possível de trilhos**.

Se fizermos isso certinho, teremos o que chamamos de **Árvore Geradora Mínima** (AGM).

📜 Dois jeitos de construir a ferrovia

Existem dois engenheiros brilhantes que propuseram maneiras de fazer isso:

🚂 Engenheiro Kruskal

Ele olha todas as possíveis conexões entre as cidades e sempre escolhe a mais barata. Mas ele tem uma regra: **nunca cria um ciclo**. Ele continua adicionando trilhos até que todas as cidades estejam conectadas.

🔹 Como funciona?

1. Ordenamos os trilhos do mais barato para o mais caro.
2. Vamos pegando os trilhos mais baratos e colocando no mapa.
3. Se um trilho criar um ciclo, a gente ignora.
4. Quando todas as cidades estiverem conectadas, terminamos.

Esse método é como se estivéssemos montando um quebra-cabeça peça por peça, começando pelas conexões mais baratas.

🚆 Engenheiro Prim

O Prim é mais metódico. Ele começa em uma cidade qualquer e vai expandindo a ferrovia escolhendo sempre a cidade mais próxima com o menor custo.

🔹 Como funciona?

1. Escolhemos uma cidade inicial.
2. Procuramos a cidade mais próxima que ainda não está na rede e colocamos o trilho mais barato.
3. Repetimos isso até que todas as cidades estejam conectadas.

Esse método é como construir uma teia de aranha, crescendo aos poucos e sempre pegando o fio mais curto primeiro.

✂️ A Arte de Escolher os Trilhos Certos

Mas como sabemos que estamos escolhendo os melhores trilhos? Aqui entra um conceito chamado **corte**.

💡 **Imagina que pegamos um grande lápis mágico e traçamos uma linha no mapa** dividindo as cidades em dois grupos. Sempre que traçamos essa linha, alguns trilhos ficam "cortados" no meio – esses são os trilhos que ligam os dois grupos.

A regra é simples: **sempre pegamos o trilho mais barato entre esses "cortados"**. Isso garante que estamos fazendo a escolha certa sem desperdiçar material.

Isso é garantido pelo **Teorema 23.1**, que basicamente diz:
✏️ **"O trilho mais barato que atravessa a linha sempre faz parte de pelo menos um ótimo jeito de conectar as cidades"**.

🏆 No Final das Contas...

Seguindo essas regras, conseguimos montar uma ferrovia ligando todas as cidades gastando o mínimo de material possível. E esse mesmo princípio funciona para **circuitos eletrônicos, redes de computadores, planejamento urbano e até mesmo na natureza!**

📚 Construindo a Melhor Rede: O GENERIC-MST e o Teorema 23.1

Imagine que você é um arquiteto encarregado de construir uma rede de estradas entre várias cidades. O seu objetivo é **conectar todas as cidades com o menor custo possível, sem criar rotas desnecessárias ou fechadas em círculos**. Essa é a essência do problema da Árvore Geradora Mínima (AGM)!

✨ O Segredo do Corte Mágico: O Teorema 23.1

Para garantir que estamos escolhendo sempre a melhor estrada para conectar as cidades, existe uma regra de ouro chamada **Teorema 23.1**.

⚡ Como funciona esse teorema?

1. Pegamos um mapa e traçamos uma linha imaginária nele, separando as cidades em dois grupos.
2. Algumas estradas cruzam essa linha – essas são as estradas candidatas para serem construídas.
3. O **Teorema 23.1 garante que a estrada mais barata que atravessa essa linha faz parte de pelo menos uma solução ótima**.
4. Se trocarmos qualquer outra estrada por essa mais barata, a rede continuará sendo a melhor possível.

🌟 O que isso significa na prática?

Isso nos ajuda a sempre fazer a escolha certa durante a construção da nossa rede, garantindo que nunca gastamos mais do que o necessário.

🛠️ O Método GENERIC-MST: O Planejamento da Construção

Agora que sabemos qual estrada escolher, precisamos de um **plano de construção**! Esse plano chama-se **GENERIC-MST**, que é um algoritmo abstrato que serve de base para os algoritmos de Kruskal e Prim.

⚡ Como funciona o GENERIC-MST?

1. Começamos sem nenhuma estrada.
2. Vamos adicionando as estradas mais baratas de forma segura, garantindo que elas nunca formem um círculo.

3. Cada nova estrada reduz o número de regiões separadas, até que todas as cidades estejam conectadas.

🔄 Garantia de uma Rede Perfeita

O GENERIC-MST funciona porque:

- Ele sempre mantém um conjunto de estradas que faz parte de pelo menos uma solução ótima.
- O mapa parcial construído sempre forma uma **floresta** (um conjunto de pequenas redes separadas, sem círculos).
- No final, essa floresta se transforma em uma única rede ótima.

🎨 Como Escolher a Melhor Estrada?

O **Corolário 23.2** nos dá uma regra simples:

> "Se tivermos um grupo de cidades e uma estrada mais barata conectando esse grupo a outra parte do mapa, essa estrada é segura para a solução ótima."

Isso significa que **sempre podemos adicionar a menor estrada entre duas regiões desconectadas**, garantindo que a nossa rede fique sempre a melhor possível.

🎯 Estratégias de Construção: Kruskal vs Prim

Agora que entendemos as regras, precisamos decidir **como aplicar o GENERIC-MST na prática**. Temos dois engenheiros principais: **Kruskal** e **Prim**.

🚂 O Engenheiro Kruskal: Construindo aos Poucos

O Kruskal segue uma abordagem **gulosa**, escolhendo primeiro as estradas mais baratas:

1. Começamos sem nenhuma estrada.
2. Ordenamos todas as estradas por custo.
3. Pegamos a estrada mais barata e verificamos se ela forma um ciclo.
4. Se não formar um ciclo, adicionamos.
5. Repetimos até todas as cidades estarem conectadas.

👨‍🌾 Como o Kruskal é tão eficiente?

Ele usa uma estrutura chamada **conjuntos disjuntos**:

- **FIND-SET(u):** Descobre em qual grupo uma cidade está.
- **UNION(u, v):** Junta duas cidades em um único grupo.
- **MAKE-SET(v):** Inicializa cada cidade como uma rede separada.

Isso garante que conseguimos unir as estradas sem formar ciclos de forma **rápida e eficiente**.

⏳ Análise de Complexidade do Kruskal

- Ordenar as estradas: **O(E log E)**
- Operações com conjuntos disjuntos: **O(E α(V))**
- No total: **O(E log E)** (ótimo para grafos esparsos!)

🚌 O Engenheiro Prim: Expandindo a Rede

O Prim funciona de maneira diferente. Ele **começa em uma cidade e vai expandindo a rede aos poucos**:

1. Escolhemos uma cidade inicial.
2. Adicionamos sempre a estrada mais barata que conecta a rede a uma nova cidade.
3. Repetimos até que todas as cidades estejam conectadas.

🧠 Como o Prim Escolhe as Estradas Certas?

O Prim usa uma **fila de prioridade**, onde mantemos uma lista das estradas disponíveis, sempre pegando a mais barata.

⏳ Análise de Complexidade do Prim

- Usando um heap binário:
  - EXTRACT-MIN: **O(log V)**
  - DECREASE-KEY: **O(log V)**
  - Total: **O(E log V)** (ótimo para grafos densos!)

- Usando um heap de Fibonacci:
  - EXTRACT-MIN: **O(log V) amortizado**
  - DECREASE-KEY: **O(1) amortizado**
  - Total: **O(E + V log V)** (ainda melhor!)

🏆 Qual Método Escolher?

Algoritmo	Estratégia	Melhor para
Kruskal	Ordena todas as estradas e adiciona as menores	Grafos esparsos (poucas conexões)
Prim	Expande uma única árvore aos poucos	Grafos densos (muitas conexões)

🎉 Conclusão: A Rede Perfeita

Agora sabemos como construir a melhor rede de estradas! Os ingredientes principais foram:

- **O Teorema 23.1**, garantindo que sempre pegamos a melhor estrada disponível.
- **O GENERIC-MST**, que nos ensinou como construir a rede de forma genérica.
- **O Corolário 23.2**, mostrando que a menor estrada entre duas partes do mapa sempre é segura.
- **Os algoritmos de Kruskal e Prim**, que aplicam essas ideias para construir a rede da forma mais eficiente possível.

Algoritmo de Kruskal: Construindo o castelo com menos ouro

O **Kruskal** é como um engenheiro muito esperto que sempre escolhe primeiro os trilhos mais baratos. Ele segue esses passos:

1. **Pegue todos os trilhos e organize-os do mais barato para o mais caro.**
2. **Comece com uma cidade sem trilhos (nenhuma ligação entre elas).**
3. **Para cada trilho na lista:**
   - Se este trilho **não criar um ciclo**, construa-o!
   - Se criar um ciclo, ignore-a e passe para a próxima.
4. **Pare quando todas as cidades estiverem conectadas.**

Implementação de Kruskal em Java

Java

```java
import java.util.*;


class Aresta implements Comparable<Aresta> {

    int origem, destino, peso;


    public Aresta(int origem, int destino, int peso) {

        this.origem = origem;

        this.destino = destino;

        this.peso = peso;

    }


    public int compareTo(Aresta outra) {

        return this.peso - outra.peso; // Ordena por peso
crescente
```

```
    }
}

class ConjuntoDisjunto {
    int[] pai, rank;

    public ConjuntoDisjunto(int n) {
        pai = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            pai[i] = i;
            rank[i] = 0;
        }
    }

    public int find(int v) {
        if (pai[v] != v) {
            pai[v] = find(pai[v]);
        }
        return pai[v];
    }

    public void union(int v1, int v2) {
```

```
        int raiz1 = find(v1);
        int raiz2 = find(v2);

        if (raiz1 != raiz2) {
            if (rank[raiz1] > rank[raiz2]) {
                pai[raiz2] = raiz1;
            } else if (rank[raiz1] < rank[raiz2]) {
                pai[raiz1] = raiz2;
            } else {
                pai[raiz2] = raiz1;
                rank[raiz1]++;
            }
        }
    }
}

class Kruskal {
    public static List<Aresta> kruskal(int vertices,
List<Aresta> arestas) {
        List<Aresta> resultado = new ArrayList<>();
        Collections.sort(arestas); // Ordena as arestas por
peso

        ConjuntoDisjunto ds = new ConjuntoDisjunto(vertices);
```

```java
        for (Aresta aresta : arestas) {

            if (ds.find(aresta.origem) !=
ds.find(aresta.destino)) {

                resultado.add(aresta);

                ds.union(aresta.origem, aresta.destino);

            }

        }

        return resultado;

    }

}
```

Explicação do código:

1. Criamos uma classe **Aresta** para guardar as informações dos trilhos.
2. **Ordenamos** os trilhos do menor para o maior custo.
3. Usamos a estrutura **Conjunto Disjunto** para verificar se duas cidades estão na mesma árvore (evitar ciclos).
4. Se não formarem ciclo, **adicionamos o trilho**.

Algoritmo de Prim: Expansão de uma única árvore

Diferente do Kruskal, que começa com trilhos soltos, o **Prim** inicia de uma cidade e cresce, sempre adicionando o trilho mais barato que expande a árvore.

1. **Escolha uma cidade inicial.**
2. **Marque essa cidade como conectada.**
3. **Encontre o trilho mais barato que liga a árvore a uma nova cidade.**
4. **Marque essa nova cidade como conectada e repita.**
5. **Pare quando todas as cidades estiverem conectadas.**

Implementação de Prim em Java

Java

```
import java.util.*;


class Prim {

   public static void prim(int grafo[][], int vertices) {

        boolean[] visitado = new boolean[vertices];

        int[] chave = new int[vertices];

        int[] pai = new int[vertices];

        Arrays.fill(chave, Integer.MAX_VALUE);

        chave[0] = 0;

        pai[0] = -1;


        for (int i = 0; i < vertices - 1; i++) {

            int u = extrairMin(chave, visitado);

            visitado[u] = true;


            for (int v = 0; v < vertices; v++) {

                if (grafo[u][v] != 0 && !visitado[v] &&
grafo[u][v] < chave[v]) {

                    chave[v] = grafo[u][v];

                    pai[v] = u;

                }
```

```java
            }
        }
        imprimirAGM(pai, grafo, vertices);
    }

    private static int extrairMin(int[] chave, boolean[]
visitado) {
        int min = Integer.MAX_VALUE, indiceMin = -1;
        for (int v = 0; v < chave.length; v++) {
            if (!visitado[v] && chave[v] < min) {
                min = chave[v];
                indiceMin = v;
            }
        }
        return indiceMin;
    }

    private static void imprimirAGM(int[] pai, int[][] grafo,
int vertices) {
        System.out.println("Arestas da AGM:");
        for (int i = 1; i < vertices; i++) {
            System.out.println(pai[i] + " - " + i + " peso: "
+ grafo[i][pai[i]]);
        }
```

```
    }
}
```

Resumo do código:

- Começamos de uma cidade.
- A cada passo, adicionamos o menor trilho possível.
- Usamos uma estrutura de chaves para manter o menor custo para cada cidade.