Teoria Análise de Algoritmos P2

- Codigo do github do bossini
- Resumo via GPT

Algoritmo de Kruskal

Definição

O **algoritmo de Kruskal** é um algoritmo clássico da Teoria dos Grafos utilizado para encontrar uma **Árvore Geradora Mínima (AGM)** de um **grafo não direcionado e conectado**.

A **Árvore Geradora Mínima (AGM)** é um subconjunto das arestas do grafo que:

- **Conecta todos os vértices** (ou seja, é uma árvore — sem ciclos),
- E tem o **menor peso total possível** (soma dos pesos das arestas é mínima).

Etapas do Algoritmo de Kruskal

1. **Ordenar as arestas**
    - Todas as arestas do grafo são colocadas em uma lista.
    - A lista é ordenada em **ordem crescente de peso**.
2. **Inicializar o Union-Find**
    - Cada vértice começa em seu **próprio conjunto** (componente desconectado).
    - Utilizamos a estrutura **Union-Find** para rastrear quais vértices estão conectados.
3. **Percorrer as arestas em ordem crescente**

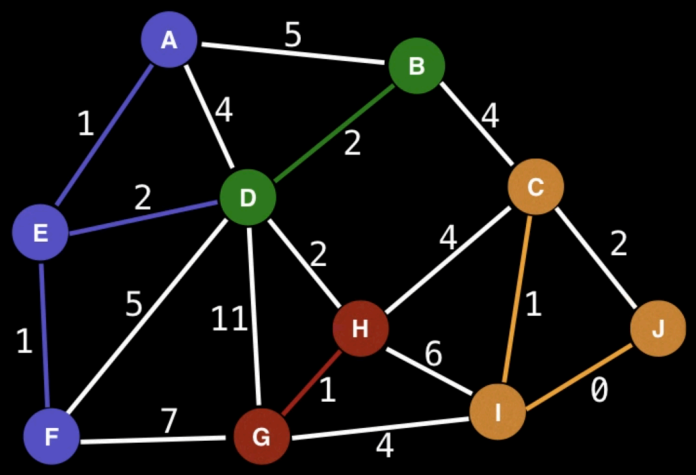
    - Para cada aresta `(u, v)`:
        - Verificamos se os vértices `u` e `v` estão em **conjuntos diferentes** (ou seja, ainda não estão conectados).
        - Se estiverem separados:
            - **Adicionamos essa aresta à árvore**
            - **Unimos os conjuntos** de `u` e `v` no Union-Find.
        - Se já estiverem conectados, **ignoramos a aresta** para evitar **ciclos**.
4. **Parar quando tiver `n - 1` arestas**
    - Uma árvore com `n` vértices sempre tem `n - 1` arestas.
    - Quando atingimos esse número, a árvore geradora mínima está completa.

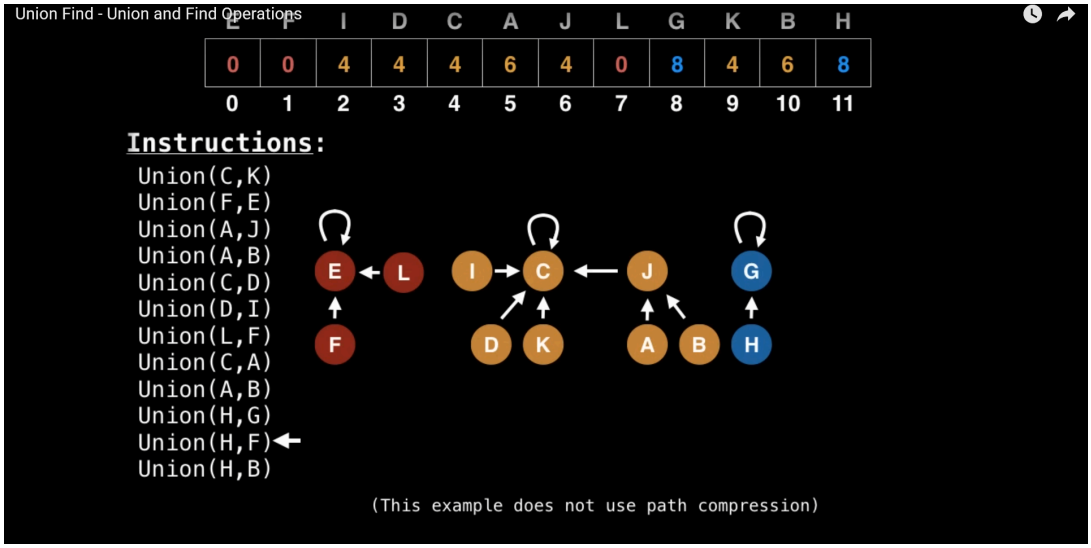
Características

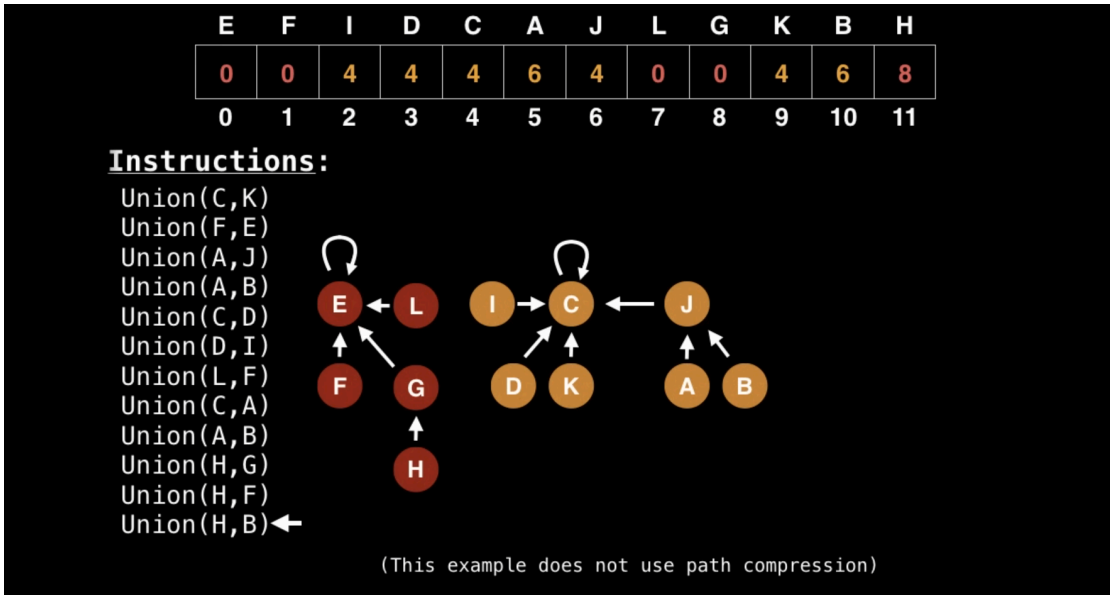
- **Complexidade**:
    - Ordenar arestas: O(E log E)
    - Operações Union-Find com compressão de caminho: praticamente O(1) por operação
    - Complexidade total: **O(E log E)**, onde `E` é o número de arestas.
- **Entrada esperada**:
    - Grafo **não direcionado**, **ponderado** e **conectado**.
- **Saída**:
    - Conjunto de arestas que formam a árvore geradora mínima e seu **peso total mínimo**.

Union Find application: Kruskal's Minimum Spanning Tree

I to J = 0
A to E = 1
C to I = 1
E to F = 1
G to H = 1
B to D = 2
C to J = 2
D to E = 2
D to H = 2
A to D = 4
B to C = 4
C to H = 4
G to I = 4
A to B = 5
D to F = 5
H to I = 6
F to G = 7
D to G = 11








```java
package arvores_geradoras_minimas;
import java.util.*;

// Classe que representa uma aresta de um grafo
// Implementa a interface Comparable para poder ordenar as arestas por peso
class Aresta implements Comparable<Aresta> {
  int u, v, peso;

  // Construtor que define os vértices u, v e o peso da aresta
  Aresta(int u, int v, int peso) {
    this.u = u;
    this.v = v;
    this.peso = peso;
  }
```

```java
  // Método que permite comparar duas arestas com base no peso
  // Isso é usado para ordenar as arestas em ordem crescente de peso
  @Override
  public int compareTo(Aresta o) {
    if (this.peso < o.peso)
      return -1; // a aresta atual vem antes
    if (this.peso == o.peso)
      return 0;  // pesos iguais
    return 1;     // a aresta atual vem depois
    // Alternativamente poderia usar: return Integer.compare(this.peso, o.peso);
  }
}

// Estrutura de dados Union-Find (ou Disjoint Set)
// Serve para controlar quais vértices estão no mesmo componente
class UnionFind {
  int[] representantes; // Vetor onde cada índice aponta para seu representante

  // Construtor: cada vértice começa sendo seu próprio representante
  public UnionFind(int n) {
    representantes = new int[n];
    for (int i = 0; i < n; i++) {
      representantes[i] = i;
    }
  }

  // Método "find": encontra o representante de um vértice
  // Segue os ponteiros até encontrar o vértice que é seu próprio representante
  public int find(int x) {
    while (representantes[x] != x) {
      x = representantes[x];
    }
    return x;
  }

  // Método "union": une dois conjuntos (de x e y)
  // O representante de y passa a ser o representante de x
  public void union(int x, int y) {
    int rX = find(x);
    int rY = find(y);
    representantes[rY] = rX;
  }
}

// Classe principal que implementa o algoritmo de Kruskal
public class KruskalSimples {

  // Método que executa o algoritmo de Kruskal
  // Recebe o número de vértices (n) e a lista de arestas
  public List<Aresta> kruskal(int n, List<Aresta> arestas) {

    // Primeiro, ordenamos as arestas em ordem crescente de peso
```

```java
    Collections.sort(arestas);

    // Criamos a estrutura Union-Find para controlar os componentes conectados
    var uf = new UnionFind(n);

    // Lista onde vamos armazenar as arestas escolhidas para formar a árvore mínima
    var arvoreResultante = new ArrayList<Aresta>();

    // Iteramos sobre todas as arestas ordenadas
    for (var a : arestas) {
      // Verificamos se os vértices da aresta estão em componentes diferentes
      // Isso evita formar ciclos
      if (uf.find(a.u) != uf.find(a.v)) {
        // Se estiverem em componentes diferentes, unimos os conjuntos
        uf.union(a.u, a.v);
        // Adicionamos essa aresta na árvore geradora mínima
        arvoreResultante.add(a);
      }
    }

    // Retornamos a lista de arestas que compõem a árvore geradora mínima
    return arvoreResultante;
  }

  // Método principal: executa um exemplo do algoritmo de Kruskal
  public static void main(String[] args) {
    int n = 4; // Número de vértices no grafo

    // Lista de arestas com seus respectivos pesos
    // O grafo é não direcionado, ou seja, (0,1) é igual a (1,0)
    List<Aresta> arestas = Arrays.asList(
      new Aresta(0, 1, 10),
      new Aresta(0, 2, 6),
      new Aresta(0, 3, 5),
      new Aresta(1, 3, 15),
      new Aresta(2, 3, 4)
    );

    // Executamos o algoritmo de Kruskal e obtemos a árvore geradora mínima
    var arvoreResultante = new KruskalSimples().kruskal(n, arestas);

    int total = 0; // Variável para acumular o peso total da árvore geradora

    // Imprimimos cada aresta selecionada e somamos os pesos
    for (var a : arvoreResultante) {
      System.out.printf("(%d, %d, %d) ", a.u, a.v, a.peso);
      total += a.peso;
    }

    // Exibimos o peso total da árvore geradora mínima
    System.out.println("Peso total : " + total);
  }
```

```
}
```

Algoritmo Heap Sort

Definição

O **Heap Sort** é um algoritmo de ordenação baseado em uma estrutura de dados chamada **heap binário** — geralmente uma **max-heap**, onde o maior elemento está sempre na raiz (topo da árvore). Ele é um algoritmo **in-place** (não precisa de memória extra significativa), e tem complexidade de tempo **O(n log n)** em todos os casos.

Seu principal objetivo é **ordenar um array de números de forma eficiente e estável**, transformando-o primeiro em um heap e depois extraindo repetidamente o maior elemento para colocá-lo em sua posição correta no final do array.

Etapas do Heap Sort

1. Construir um Max-Heap

- O array é reorganizado para satisfazer a estrutura de **max-heap**, onde:
    - Cada pai é maior que seus filhos.
- Isso é feito usando o procedimento `maxHeapify` de baixo para cima.
- O índice 1 (ou 0 em versões baseadas em 0) será a **raiz** da heap.

2. Extrair o maior elemento

- O maior elemento (raiz da heap) é trocado com o **último elemento** do array.
- O tamanho da heap é reduzido (exclui-se a última posição, já ordenada).

- Aplica-se `maxHeapify` novamente para restaurar a propriedade de max-heap.

3. Repetir até o heap ficar vazio

- Continua trocando a raiz com o último elemento do heap restante.
- Heapifica novamente.
- Isso continua até que todos os elementos estejam ordenados.

Características

- **Complexidade de tempo**:
    - Construção do heap: **O(n)**
    - Cada extração com reestruturação: **O(log n)**
    - Total: **O(n log n)**
- **Complexidade de espaço**:
    - **O(1)** extra — é um algoritmo **in-place**
- **Estável?**
    - **Não** — elementos com valores iguais podem ter suas posições relativas trocadas.
- **Tipo de ordenação**:
    - Normalmente retorna array em **ordem crescente**, usando **max-heap**.

Exemplo de Aplicações

- Quando se precisa de **eficiência garantida**, mesmo nos piores casos.
- Em **sistemas embarcados** ou ambientes com restrição de memória (por ser in-place).
- Para **ordenar grandes volumes de dados** com desempenho previsível.
- Em algoritmos que exigem acesso rápido ao maior (ou menor) elemento, como filas de prioridade.

```
package heap;

// Classe principal com implementação do algoritmo de ordenação HeapSort
public class HeapSort {

  // Retorna o índice do nó pai de um nó na posição i
  public static int parent(int i) {
    return i / 2;
  }

  // Retorna o índice do filho esquerdo de um nó na posição i
  public static int left(int i) {
    return 2 * i;
  }

  // Retorna o índice do filho direito de um nó na posição i
  public static int right(int i) {
    return 2 * i + 1;
  }

  // Função auxiliar para trocar dois elementos no array
  private static void swap(int[] A, int i, int j) {
    var temp = A[i];
    A[i] = A[j];
    A[j] = temp;
  }

  // Garante que o sub-árvore com raiz em A[i] obedeça a propriedade de max-heap
  // i é o índice do nó atual e n é o tamanho do heap
  public static void maxHeapify(int[] A, int i, int n) {
    // Encontra os índices dos filhos esquerdo e direito do nó atual
    var l = left(i);
    var r = right(i);
    int largest;

    // Verifica se o filho esquerdo é maior que o pai
    if (l <= n && A[l] > A[i])
      largest = l;
    else
      largest = i;

    // Verifica se o filho direito é maior do que o maior entre pai e filho esquerdo
```

```
    if (r <= n && A[r] > A[largest])
      largest = r;

    // Se o maior não for o próprio pai, faz a troca e continua heapificando recursivamente
    if (largest != i) {
      swap(A, i, largest); // troca pai com maior filho
      maxHeapify(A, largest, n); // reaplica heapify na posição trocada
    }
  }

  // Constrói uma max-heap a partir de um array desorganizado
  public static void buildMaxHeap(int[] A, int n) {
    // Começa do meio do array até o início
    // Isso porque a partir de n/2 todos os nós são folhas e já estão em heap
    for (int i = n / 2; i >= 1; i--) {
      maxHeapify(A, i, n); // ajusta cada sub-árvore
    }
  }

  // Função principal do algoritmo de ordenação HeapSort
  public static void heapSort(int[] A) {
    // Assumimos que o índice 0 é ignorado (heap começa no índice 1)
    int n = A.length - 1;

    // Primeiro, construímos o max-heap
    buildMaxHeap(A, n);

    // Repetidamente removemos o maior elemento (raiz) e colocamos no final
    for (int i = n; i >= 2; i--) {
      swap(A, 1, n); // coloca a raiz no final do heap
      n--; // reduz o tamanho do heap
      maxHeapify(A, 1, n); // restaura a propriedade de max-heap
    }
  }
}
```

Algoritmo de Menor Caminho com BFS (Busca em Largura)

Definição

A **Busca em Largura (BFS)** é um algoritmo de travessia em grafos que visita os vértices em **camadas**, começando por um vértice de origem e explorando todos os seus vizinhos antes de passar para os vizinhos dos vizinhos.

Quando aplicada a um **grafo não ponderado** (onde todas as arestas têm o mesmo "peso" ou custo), a BFS é capaz de encontrar o **menor caminho** entre dois vértices, no sentido de **menor número de arestas**.

Objetivo

O objetivo é encontrar **quantos passos (arestas)** são necessários para ir de um vértice **origem** até um vértice **destino**, em um grafo **não ponderado**.

Etapas do Algoritmo BFS para Menor Caminho

1. Inicializar estruturas auxiliares

- Um vetor `distancias[]` para armazenar a distância de cada vértice até a origem.
- Um vetor `visitados[]` para marcar quais vértices já foram visitados.
- Uma fila `fila` (FIFO) para processar os vértices por nível.

2. Configurar o ponto de partida

- Marcar a **origem** como visitada.
- Colocar a **origem** na fila.
- Definir `distancia[origem] = 0`.

3. Executar a BFS

- Enquanto a fila não estiver vazia:
    - Retirar o próximo vértice da fila.

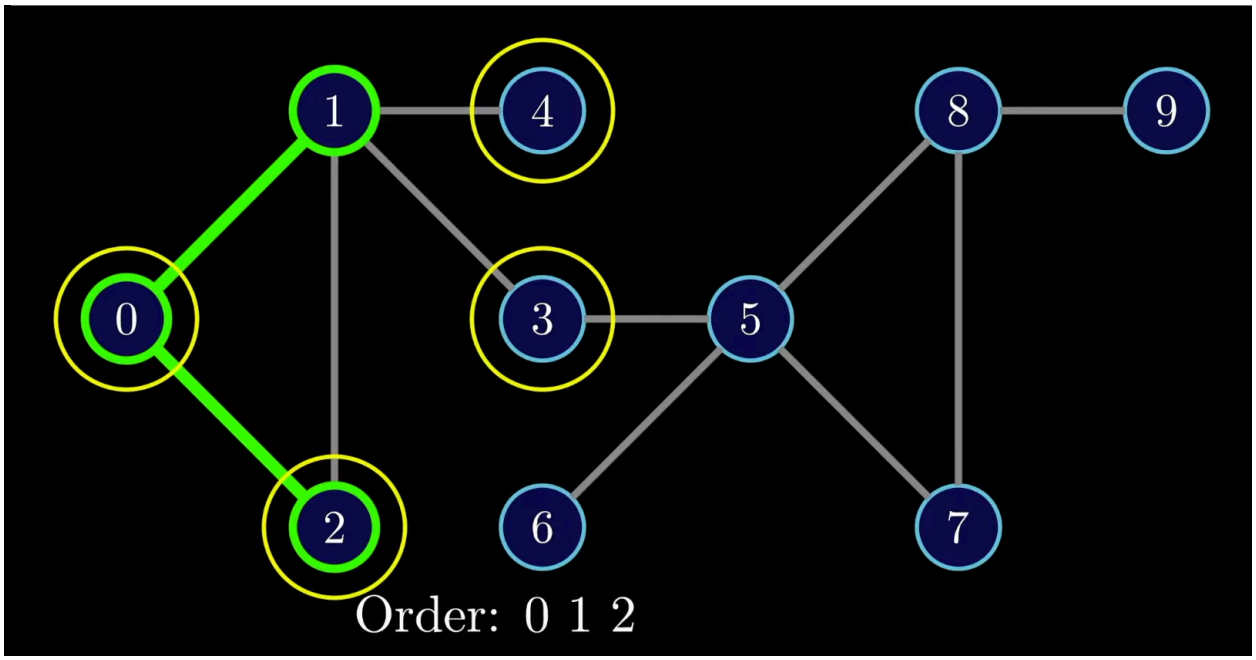
- Para cada **vizinho** desse vértice:
    - Se ainda não foi visitado:
        - Marcar como visitado.
        - Colocar o vizinho na fila.
        - Atualizar a distância: `distancia[vizinho] = distancia[atual] + 1`.

4. Finalizar

- Ao final, `distancia[destino]` contém o número mínimo de arestas entre a origem e o destino.

Características

- **Tipo de grafo**: Não direcionado ou direcionado, **não ponderado**.
- **Tempo de execução**: O(V + E), onde:
    - V = número de vértices
    - E = número de arestas
- **Complexidade de espaço**: O(V) para armazenar distâncias, visitados e fila.
- **Resultado**: Distância mínima entre dois vértices (em número de arestas).




```
def floodFill(img, row, col, p):
    start = img[row][col]
    queue = [(row, col)]
    visited = set()
    while len(queue) > 0:
        row, col = queue.pop(0)
        visited.add((row, col))
        img[row][col] = p
        for row, col in neighbors(img, row, col, start):
            if (row, col) not in visited:
                queue.append((row, col))
    return img
def neighbors(img, row, col, start):
    indices = [(row - 1, col), (row + 1, col), (row, col - 1), (row, col + 1)]
    return [(row, col) for row, col in indices if isValid(img, row, col) and
                        img[row][col] == start]
def isValid(img, row, col):
    return row ≥ 0 and col ≥ 0 and row < len(img) and col < len(img[0])
```

1	0	2	2	0
0	2	0	2	0
2	2	2	2	2
0	0	2	0	2
1	0	0	0	0

(row, col): $(2, 2)$ $p : 2$

```
package menor_caminho;
import java.util.*;

// Classe que representa um grafo não direcionado
public class Grafo {
```

```java
// Número total de vértices no grafo
private int vertices;

// Lista de adjacência: cada posição do array contém uma lista de vértices vizinhos
private LinkedList<Integer>[] adjacencias;

// Construtor: inicializa o grafo com 'vertices' vértices
Grafo(int vertices) {
  this.vertices = vertices;

  // Cria o array de listas de adjacência
  this.adjacencias = new LinkedList[this.vertices];

  // Inicializa cada lista individualmente
  for (int i = 0; i < this.vertices; i++) {
    this.adjacencias[i] = new LinkedList<Integer>();
  }
}

// Método para encontrar o menor caminho (em número de arestas) entre dois vértices
// Isso NÃO é Dijkstra -- é uma BFS (Busca em Largura) para grafos não ponderados
void menorCaminho(int origem, int destino) {
  int[] distancias = new int[vertices]; // Vetor com as distâncias a partir da origem
  boolean[] visitados = new boolean[vertices]; // Marca quais vértices já foram visitados

  Queue<Integer> fila = new LinkedList<>(); // Fila para BFS
  fila.offer(origem); // Adiciona o vértice de origem à fila
  distancias[origem] = 0; // Distância da origem para ela mesma é 0
  visitados[origem] = true; // Marca a origem como visitada

  // Enquanto houver vértices na fila, continua a busca
  while (!fila.isEmpty()) {
    var atual = fila.poll(); // Remove o primeiro da fila

    // Para cada vizinho do vértice atual
    for (int vizinho : adjacencias[atual]) {

      // Se ainda não foi visitado, é o caminho mais curto até ele
      if (!visitados[vizinho]) {
        fila.offer(vizinho); // Adiciona o vizinho à fila para explorar depois
        visitados[vizinho] = true; // Marca como visitado
        distancias[vizinho] = distancias[atual] + 1; // Atualiza a distância até ele
      }
    }
  }

  // Ao final da busca, imprime a distância da origem ao destino
  System.out.printf(
    "Distância de %d até %d: %d\n",
    origem, destino, distancias[destino]
  );
}
```

```java
  // Método para adicionar uma aresta não direcionada entre dois vértices
  void adicionarAresta(int a1, int a2) {
    this.adjacencias[a1].add(a2);
    this.adjacencias[a2].add(a1);
  }

  // Método principal: lê os argumentos, cria o grafo, adiciona arestas e executa a busca
  // Exemplo de uso:
  // java Grafo 7 0 1 0 2 0 5 1 3 2 4 4 6 5 6 0 5
  // Aqui: 7 vértices, várias arestas, origem 0, destino 5
  public static void main(String[] args) {
    int vertices = Integer.parseInt(args[0]); // Número de vértices
    Grafo grafo = new Grafo(vertices);

    // Adiciona as arestas usando os argumentos passados
    for (int i = 1; i < args.length - 2; i++) {
      grafo.adicionarAresta(
        Integer.parseInt(args[i]),
        Integer.parseInt(args[i + 1])
      );
    }

    // Executa a busca do menor caminho entre os dois últimos argumentos
    grafo.menorCaminho(
      Integer.parseInt(args[args.length - 2]),
      Integer.parseInt(args[args.length - 1])
    );
  }
}
```

Algoritmo de Dijkstra — Menores Caminhos a Partir de Uma Única Origem

Definição

O **algoritmo de Dijkstra** é um algoritmo clássico da Teoria dos Grafos que resolve o problema de **menores caminhos a partir de uma única origem** em um grafo **ponderado e direcionado**, onde **todos os pesos das arestas são não negativos**.

Ele calcula o **menor custo** (ou distância) entre um **vértice de origem** e todos os outros vértices do grafo, armazenando o custo mínimo e o caminho correspondente.

Objetivo

- Encontrar o **menor caminho** (em termos de soma de pesos) do vértice de origem para todos os outros vértices.
- Retornar tanto a **distância mínima** quanto o **predecessor** de cada vértice no caminho ótimo.

Etapas do Algoritmo de Dijkstra

1. Inicialização

- Definir a **distância da origem como 0**.
- Definir a **distância dos outros vértices como infinito**.
- Inicializar todos os vértices sem predecessores.
- Adicionar todos os vértices em um conjunto `Q` de não processados.

2. Laço principal (enquanto houver vértices em Q):

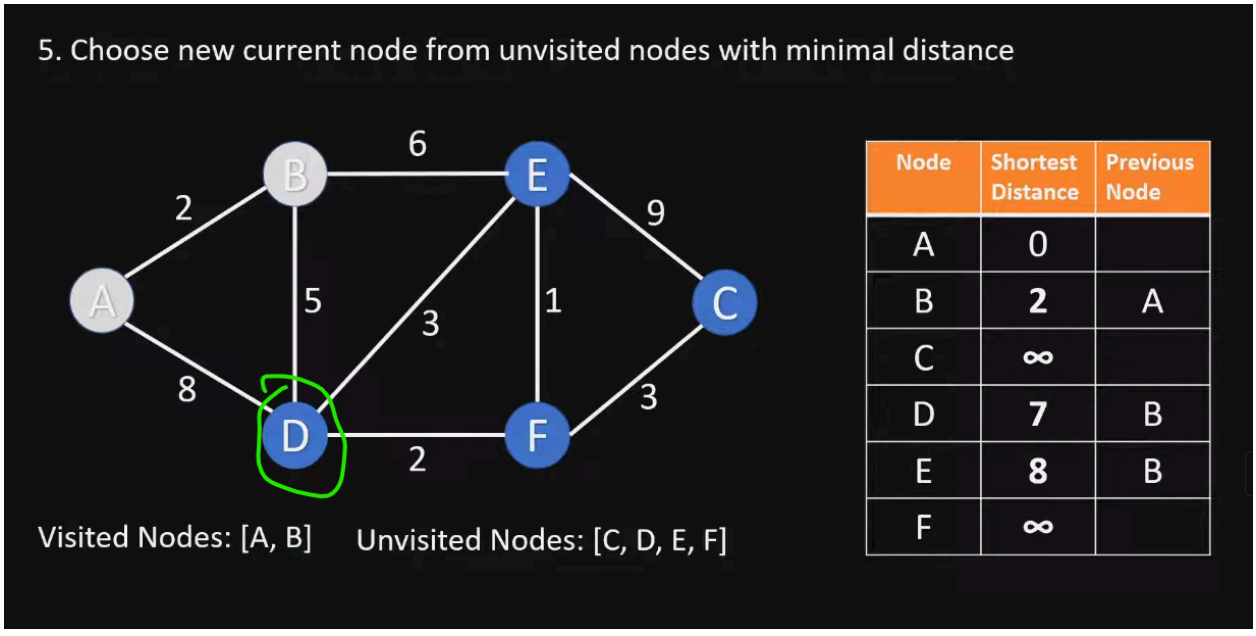
- Escolher o vértice `u` em `Q` com a **menor distância estimada** (`extrairMinimo`).
- Para cada vizinho `v` de `u`:
    - Calcular o custo de chegar a `v` passando por `u`:
      `novaDistancia = distancia[u] + peso(u, v)`
    - Se esse caminho for **menor que o conhecido atualmente**, atualize:
        - `distancia[v] = novaDistancia`
        - `predecessor[v] = u`

3. Repetir até todos os vértices serem processados

- Ao final, cada vértice terá:
    - A **menor distância** desde a origem.
    - O **predecessor** para reconstruir o caminho mínimo.

Características

Característica	Valor
Tipo de grafo	Direcionado ou não, ponderado
Peso das arestas	**Não negativos**
Complexidade (lista + min linear)	$O(V^2)$
Complexidade (com heap)	O((V + E) log V) com PriorityQueue
Algoritmo	**Guloso** (greedy)
Resultado	Menor distância e predecessores



Iniciar grafo

```java
package menores_caminhos_de_origem_unica;
import java.util.*;

// Classe que representa um Grafo dirigido e ponderado
class Grafo {

  // Vetor de vértices do grafo
  private final Vertice[] vertices;

  // Lista de adjacência: cada vértice tem uma lista de arestas (vizinhos com pesos)
  private List<List<Aresta>> adjacencias;

  // Construtor: recebe os nomes dos vértices como parâmetro
  Grafo(String[] nomesVertices) {
    this.adjacencias = new ArrayList<>();
    this.vertices = new Vertice[nomesVertices.length];

    // Para cada nome, cria um vértice e inicializa a lista de adjacência correspondente
    for (int i = 0; i < nomesVertices.length; i++) {
      this.adjacencias.add(new ArrayList<>());
      this.vertices[i] = new Vertice(nomesVertices[i], i);
    }
  }

  // Método para adicionar uma aresta direcionada com peso entre dois vértices
  public void adicionarAresta(int origem, int destino, int peso) {
    adjacencias.get(origem).add(new Aresta(destino, peso));
  }

  // Retorna a lista de vizinhos (arestas) de um vértice u
  public List<Aresta> vizinhos(int u) {
    return adjacencias.get(u);
  }

  // Retorna todos os vértices do grafo
  public Vertice[] getVertices() {
    return vertices;
  }

  // Retorna o número de vértices
  public int quantidadeVertices() {
    return vertices.length;
  }

  // Classe interna que representa um vértice do grafo
  static class Vertice {
    String nome;         // Nome do vértice (ex: "s", "t")
    int indice;          // Índice no vetor
    int distancia;       // Distância mínima estimada até a origem
```

```
    Vertice predecessor; // Vértice anterior no menor caminho

    Vertice(String nome, int indice) {
      this.nome = nome;
      this.indice = indice;
      this.distancia = Integer.MAX_VALUE; // Inicia com "infinito"
      this.predecessor = null;
    }
  }

  // Classe interna que representa uma aresta do grafo
  static class Aresta {
    int destino; // Índice do vértice de destino
    int peso;    // Peso da aresta

    Aresta(int destino, int peso) {
      this.destino = destino;
      this.peso = peso;
    }
  }
}
```

Algoritmo

```
public class Dijkstra {

  // Executa o algoritmo de Dijkstra no grafo g, a partir do vértice de índice s
  public void executar(Grafo g, int s) {
    inicializarFonteUnica(g, s); // Inicializa distâncias e predecessores
    List<Grafo.Vertice> q = new ArrayList<>(List.of(g.getVertices())); // Conjunto Q de
vértices ainda não processados

    while (!q.isEmpty()) {
      var u = extrairMinimo(q); // Extrai o vértice com menor distância estimada
      for (Grafo.Aresta aresta : g.vizinhos(u.indice)) {
        // Para cada vizinho v de u, tenta melhorar o caminho
        relaxar(u, g.getVertices()[aresta.destino], aresta.peso);
      }
    }
  }

  // Relaxa uma aresta (u, v) com peso w
  // Se passar por u até v for melhor, atualiza distância e predecessor
  private void relaxar(Grafo.Vertice u, Grafo.Vertice v, int w) {
    if (u.distancia + w < v.distancia) {
      v.distancia = u.distancia + w;
      v.predecessor = u;
    }
  }

  // Inicializa todos os vértices com distância infinita e sem predecessor
```

```java
// Apenas o vértice de origem tem distância 0
private void inicializarFonteUnica(Grafo g, int s) {
  for (Grafo.Vertice v : g.getVertices()) {
    v.distancia = Integer.MAX_VALUE;
    v.predecessor = null;
  }
  g.getVertices()[s].distancia = 0;
}

// Extrai o vértice com a menor distância estimada da lista q
private Grafo.Vertice extrairMinimo(List<Grafo.Vertice> q) {
  int indiceMin = 0;
  for (int i = 1; i < q.size(); i++) {
    if (q.get(i).distancia < q.get(indiceMin).distancia) {
      indiceMin = i;
    }
  }
  return q.remove(indiceMin); // Remove da fila e retorna o vértice
}

// Método principal: constrói o grafo, executa Dijkstra e imprime resultados
public static void main(String[] args) {
  System.out.println("oi");

  // Nomes dos vértices
  String[] nomes = {"s", "t", "x", "y", "z"};

  // Cria o grafo
  Grafo g = new Grafo(nomes);

  // Adiciona as arestas com pesos (grafo dirigido e ponderado)
  g.adicionarAresta(0, 1, 10); // s → t
  g.adicionarAresta(0, 3, 5);  // s → y
  g.adicionarAresta(1, 2, 1);  // t → x
  g.adicionarAresta(1, 3, 2);  // t → y
  g.adicionarAresta(2, 4, 4);  // x → z
  g.adicionarAresta(3, 1, 3);  // y → t
  g.adicionarAresta(3, 2, 9);  // y → x
  g.adicionarAresta(3, 4, 2);  // y → z
  g.adicionarAresta(4, 2, 6);  // z → x

  // Executa o algoritmo de Dijkstra a partir de "s" (índice 0)
  var dijkstra = new Dijkstra();
  dijkstra.executar(g, 0);

  // Imprime as distâncias mínimas e predecessores para cada vértice
  for (Grafo.Vertice v : g.getVertices()) {
    System.out.printf("Distância de s a %s: %d\n", v.nome, v.distancia);
    System.out.printf(
      "%s\n", v.predecessor != null
        ? "Predecessor de " + v.nome + ": " + v.predecessor.nome
        : "Não tem predecessor"
```

```
      );
    }
  }
}
```

Algoritmo de Bellman-Ford — Menores Caminhos com Pesos Negativos

Definição

O **algoritmo de Bellman-Ford** é um algoritmo da Teoria dos Grafos utilizado para resolver o problema de **menores caminhos a partir de uma única origem** em **grafos direcionados e ponderados**, incluindo aqueles com **pesos negativos** nas arestas.

Diferente de Dijkstra, Bellman-Ford pode lidar com **arestas de custo negativo** e também pode detectar **ciclos negativos** no grafo (ciclos cujo custo total é menor que zero).

Objetivo

- Encontrar o **menor caminho** (menor soma de pesos) do vértice de origem para **todos os outros vértices**.
- Identificar se o grafo contém **ciclos de peso negativo**.

Etapas do Algoritmo Bellman-Ford

1. Inicializar

- Definir a **distância da origem como 0**.
- Definir a **distância dos outros vértices como infinito**.
- Inicializar todos os **predecessores como `null`**.

2. Relaxar todas as arestas (V - 1) vezes

- Para cada vértice `u`:
    - Para cada aresta `u → v` com peso `w`:
        - Se `dist[u] + w < dist[v]`, atualize:
            - `dist[v] = dist[u] + w`
            - `predecessor[v] = u`

Essa etapa garante que a menor distância será descoberta mesmo com caminhos complexos.

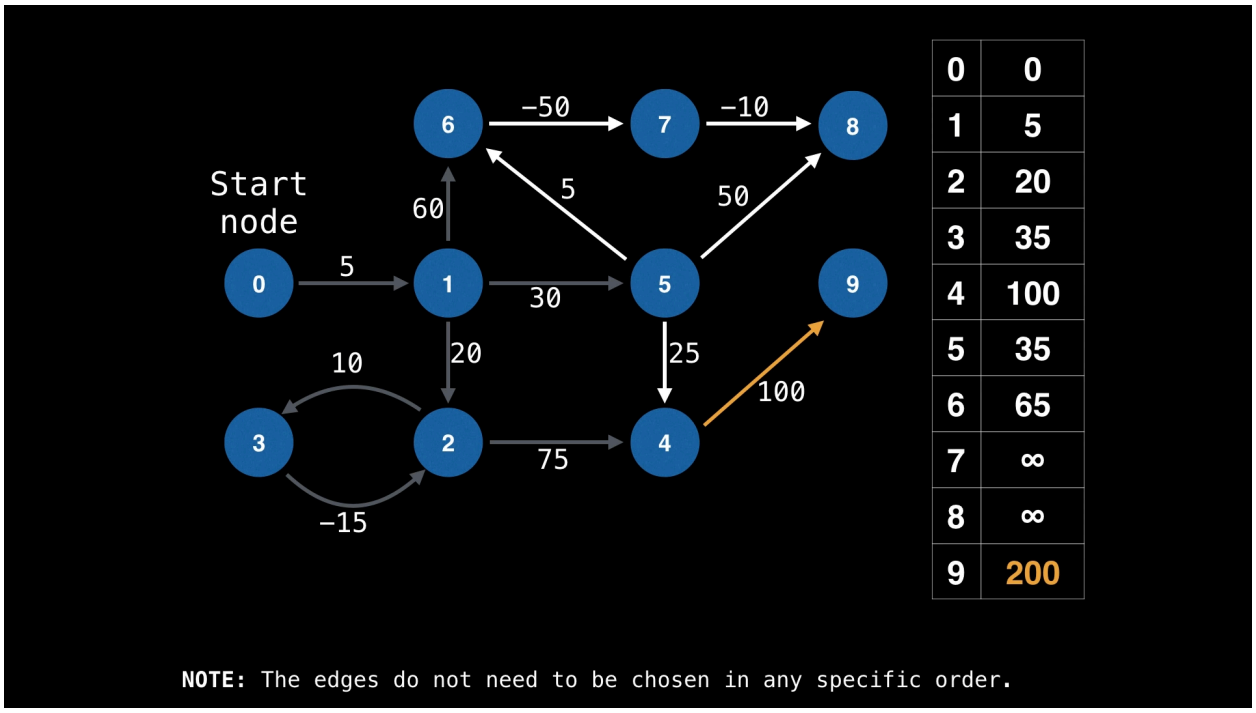
3. Verificar ciclos negativos

- Para cada aresta `u → v` com peso `w`:
    - Se `dist[u] + w < dist[v]`, então **existe um ciclo negativo** acessível a partir da origem.

Características

Propriedade	Valor
Tipo de grafo	Direcionado e ponderado
Suporta pesos negativos	Sim
Detecta ciclos negativos	Sim
Complexidade de tempo	**O(V × E)**
Complexidade de espaço	O(V)
Tipo de algoritmo	Programação dinâmica

Estável para caminhos reais	Sim



0	0
1	5
2	20
3	35
4	100
5	35
6	65
7	∞
8	∞
9	200

Grafo

```java
package menores_caminhos_de_origem_unica;
import java.util.*;

// Classe que representa o grafo
class Grafo {

  private final Vertice[] vertices; // Vetor de vértices
  private List<List<Aresta>> adjacencias; // Lista de adjacência

  // Construtor do grafo: recebe um vetor de nomes de vértices
  Grafo(String[] nomesVertices) {
    this.adjacencias = new ArrayList<>();
    this.vertices = new Vertice[nomesVertices.length];

    // Inicializa cada vértice e sua lista de adjacência
    for (int i = 0; i < nomesVertices.length; i++) {
      this.adjacencias.add(new ArrayList<>()); // cria lista para vértice i
      this.vertices[i] = new Vertice(nomesVertices[i], i); // cria vértice
    }
  }

  // Adiciona uma aresta dirigida de origem → destino com peso
  public void adicionarAresta(int origem, int destino, int peso) {
```

```
    adjacencias.get(origem).add(new Aresta(destino, peso));
  }

  // Retorna vizinhos (arestas) de um vértice
  public List<Aresta> vizinhos(int u) {
    return adjacencias.get(u);
  }

  // Retorna o vetor de vértices
  public Vertice[] getVertices() {
    return vertices;
  }

  // Retorna o número total de vértices
  public int quantidadeVertices() {
    return vertices.length;
  }

  // Classe interna que representa um vértice do grafo
  static class Vertice {
    String nome;
    int indice;
    int distancia;         // Distância mínima estimada desde a origem
    Vertice predecessor;   // Antecessor no caminho mais curto

    Vertice(String nome, int indice) {
      this.nome = nome;
      this.indice = indice;
      this.distancia = Integer.MAX_VALUE; // Começa com "infinito"
      this.predecessor = null;
    }
  }

  // Classe interna que representa uma aresta com destino e peso
  static class Aresta {
    int destino;
    int peso;

    Aresta(int destino, int peso) {
      this.destino = destino;
      this.peso = peso;
    }
  }
}
```

Algoritmo

```java
public class BellmanFord {

  // Executa o algoritmo de Bellman-Ford no grafo g a partir do vértice s (índice)
  public boolean executar(Grafo g, int s) {
    // Inicializa as distâncias e predecessores
    inicializarFonteUnica(g, s);

    // Relaxa todas as arestas V-1 vezes
    for (int i = 0; i < g.quantidadeVertices() - 1; i++) {
      for (var u : g.getVertices()) {
        for (var a : g.vizinhos(u.indice)) {
          var v = g.getVertices()[a.destino];
          relaxar(u, v, a.peso);
        }
      }
    }

    // Verifica presença de ciclos negativos
    for (var u : g.getVertices()) {
      for (var a : g.vizinhos(u.indice)) {
        var v = g.getVertices()[a.destino];
        // Se ainda for possível relaxar, então há ciclo negativo
        if (v.distancia > u.distancia + a.peso)
          return false;
      }
    }

    return true; // Não há ciclos negativos
  }

  // Aplica a operação de relaxamento na aresta (u → v)
  private void relaxar(Grafo.Vertice u, Grafo.Vertice v, int w) {
    // Se passar por u melhora a distância de v, atualiza
    if (u.distancia + w < v.distancia) {
      v.distancia = u.distancia + w;
      v.predecessor = u;
    }
  }

  // Inicializa as distâncias e predecessores para todos os vértices
  private void inicializarFonteUnica(Grafo g, int s) {
    for (Grafo.Vertice v : g.getVertices()) {
      v.distancia = Integer.MAX_VALUE; // Infinito
      v.predecessor = null;
    }
    g.getVertices()[s].distancia = 0; // Origem tem distância 0
  }

  // Função principal para testar o algoritmo
  public static void main(String[] args) {
    System.out.println("oi");
```

```
    // Define os nomes dos vértices
    String[] nomes = {"s", "t", "x", "y", "z"};

    // Cria o grafo com os nomes
    Grafo g = new Grafo(nomes);

    // Adiciona as arestas (grafo dirigido e ponderado)
    g.adicionarAresta(0, 1, 10);  // s → t
    g.adicionarAresta(0, 3, 5);   // s → y
    g.adicionarAresta(1, 2, 1);   // t → x
    g.adicionarAresta(1, 3, 2);   // t → y
    g.adicionarAresta(2, 4, 4);   // x → z
    g.adicionarAresta(3, 1, 3);   // y → t
    g.adicionarAresta(3, 2, 9);   // y → x
    g.adicionarAresta(3, 4, 2);   // y → z
    g.adicionarAresta(4, 2, 6);   // z → x

    // Executa Bellman-Ford a partir de 's' (índice 0)
    var bellmanFord = new BellmanFord();
    boolean sucesso = bellmanFord.executar(g, 0);

    // Exibe resultados
    if (sucesso) {
      for (Grafo.Vertice v : g.getVertices()) {
        System.out.printf("Distância de s a %s: %d\n", v.nome, v.distancia);
        System.out.printf(
          "%s\n", v.predecessor != null
            ? "Predecessor de " + v.nome + ": " + v.predecessor.nome
            : "Não tem predecessor"
        );
      }
    } else {
      System.out.println("O grafo contém um ciclo de peso negativo.");
    }
  }
}
```

BFS — Busca em Largura

Definição

A **Busca em Largura** percorre o grafo **em camadas**, partindo de um vértice inicial e visitando todos os vizinhos mais próximos antes de avançar para os mais distantes. É ideal para encontrar o **menor caminho em número de arestas** em grafos **não ponderados**.

Etapas do BFS

1. **Inicializar**
    - Criar uma lista (ou vetor) `visitados[]` e marcá-la como `false` para todos os vértices.
    - Criar uma fila (`Queue`) para armazenar os vértices a visitar.
    - Marcar o vértice de início como visitado e adicioná-lo à fila.
2. **Laço Principal**
    - Enquanto a fila não estiver vazia:
        - Remover o vértice da frente da fila (`poll()`).
        - Processar o vértice (ex: imprimir).
        - Para cada vizinho ainda não visitado:
            - Marcar como visitado.
            - Adicionar à fila.
3. **Fim**
    - Quando a fila estiver vazia, todos os vértices acessíveis a partir do vértice inicial foram visitados.

Complexidade

- **Tempo**: O(V + E), onde V = vértices e E = arestas.
- **Espaço**: O(V) para armazenar a fila e os visitados.

DFS — Busca em Profundidade

Definição

A **Busca em Profundidade** explora o grafo **em profundidade**, seguindo um caminho até o fim antes de retroceder. Utiliza **recursão** (ou pilha) para armazenar o estado da exploração.

Etapas do DFS

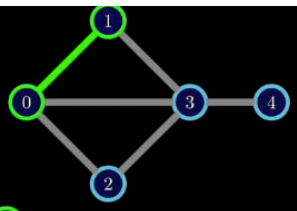
1. **Inicializar**
    - Criar uma lista `visitados[]` e marcar todos os vértices como `false`.
2. **Chamada Recursiva**
    - A partir do vértice inicial:
        - Marcar o vértice como visitado.
        - Processar o vértice (ex: imprimir).
        - Para cada vizinho não visitado:
            - Fazer chamada recursiva no vizinho.
3. **Retrocesso**
    - Quando um caminho não pode ser seguido, o algoritmo retorna para o último vértice com vizinhos não visitados.

Complexidade

- **Tempo**: O(V + E)
- **Espaço**: O(V) para recursão + visitados

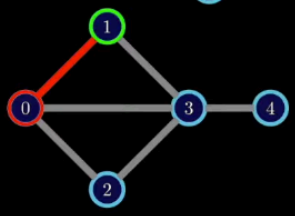
dfs(G, 0)

0 1



dfs(G, 1)

1



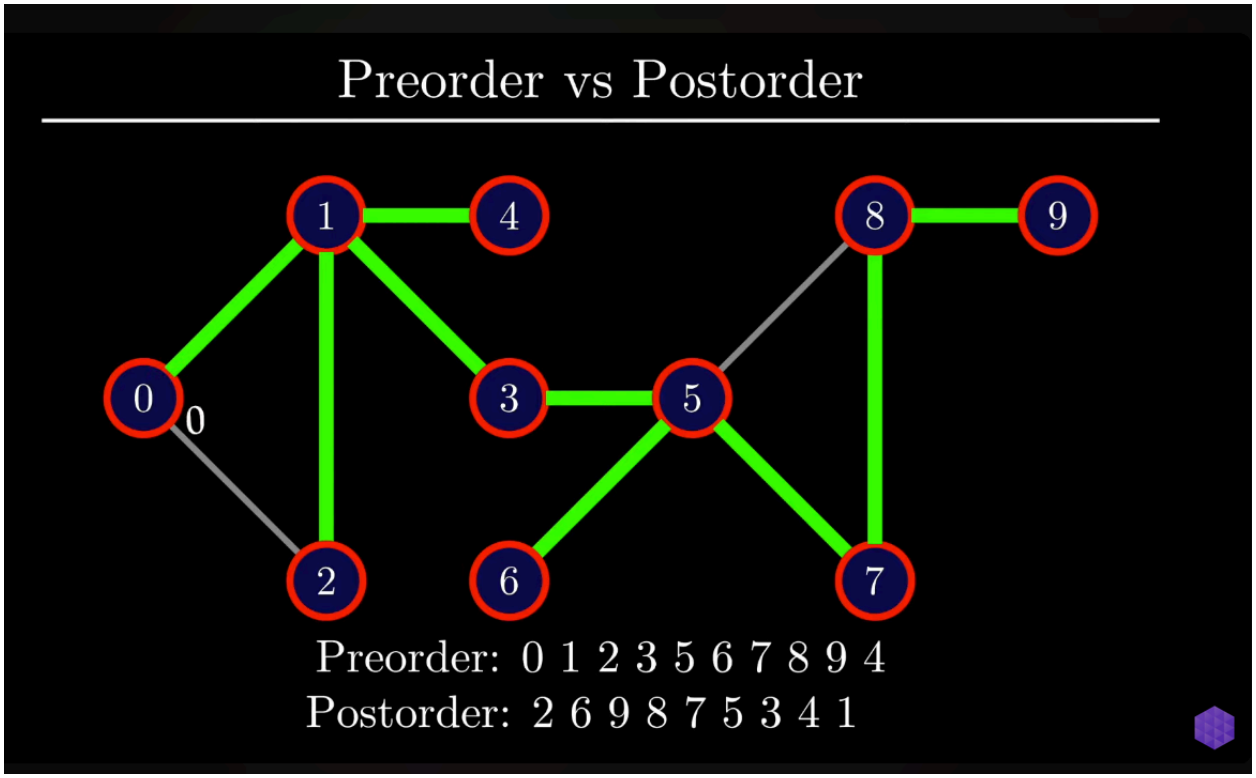

```
marked = [False] * G.size()
def dfs(G, v):
    visit(v)
    marked[v] = True
    for w in G.neighbors(v):
        if not marked[w]:
            dfs(G, w)
```

DFS Implementation Comparison

Both run in $O(V + E)$

```
marked = [False] * G.size()
def dfs(G, v):
    visit(v)
    marked[v] = True
    for w in G.neighbors(v):
        if not marked[w]:
            dfs(G, w)
```

```
marked = [False] * G.size()
def dfs_iter(G, v)
    stack = [v]
    while len(stack) > 0:
        v = stack.pop()
        if not marked[v]:
            visit(v)
            marked[v] = True
            for w in G.neighbors(v):
                if not marked[w]:
                    stack.append(w)
```



Diferenças Principais

Característica	BFS	DFS
Estrutura usada	Fila (Queue)	Recursão (ou pilha)
Ordem de visita	Em camadas	Em profundidade
Menor caminho	✅ Sim (em número de arestas)	❌ Não
Ideal para	Caminhos mínimos, largura	Exploração, ciclos, topologia
Tempo/esp. média	O(V + E)	O(V + E)

```
package bfsdfs;
```

```java
import java.util.*;

// Programa que implementa BFS e DFS em um grafo dirigido
// Exemplo de uso (linha de comando):
// java Grafo 5 0 1 1 2 2 3 3 4

public class Grafo {
  private int vertices; // Número de vértices no grafo
  private List<Integer>[] adjacencias; // Lista de adjacência para armazenar as arestas

  // Construtor: inicializa o grafo com a quantidade de vértices
  public Grafo(int vertices) {
    this.vertices = vertices;
    this.adjacencias = new LinkedList[vertices]; // Array de listas de adjacência
    for (int i = 0; i < vertices; i++) {
      adjacencias[i] = new LinkedList<>(); // Inicializa a lista para cada vértice
    }
  }

  // Adiciona uma aresta direcionada do vértice "origem" para o "destino"
  public void adicionarAresta(int origem, int destino) {
    adjacencias[origem].add(destino);
    // Se quiser que o grafo seja não direcionado, descomente a linha abaixo
    // adjacencias[destino].add(origem);
  }

  // Algoritmo de busca em largura (BFS)
  public void bfs(int inicio) {
    boolean[] visitados = new boolean[vertices]; // Marca os vértices visitados
    visitados[inicio] = true; // Marca o vértice inicial como visitado

    Queue<Integer> fila = new LinkedList<>(); // Fila para armazenar os próximos vértices a
visitar
    fila.add(inicio); // Começa pela raiz

    System.out.println("BFS:");
    while (!fila.isEmpty()) {
      int atual = fila.poll(); // Pega o próximo vértice da fila
      System.out.println(atual + " "); // Imprime o vértice visitado

      // Percorre todos os vizinhos do vértice atual
      for (int vizinho : adjacencias[atual]) {
        if (!visitados[vizinho]) {
          visitados[vizinho] = true; // Marca como visitado
          fila.add(vizinho);         // Adiciona à fila para visitar depois
        }
      }
    }
    System.out.println(); // Pula linha após a BFS
  }
```

```java
  // Algoritmo de busca em profundidade (DFS) -- versão pública que chama o recursivo
  public void dfs(int inicio) {
    boolean[] visitados = new boolean[vertices]; // Marca os vértices visitados
    System.out.println("DFS:");
    dfsRecursivo(inicio, visitados); // Inicia a chamada recursiva
    System.out.println(); // Pula linha após a DFS
  }

  // Método auxiliar recursivo para o DFS
  private void dfsRecursivo(int atual, boolean[] visitados) {
    visitados[atual] = true; // Marca o vértice como visitado
    System.out.println(atual + " "); // Imprime o vértice

    // Visita recursivamente os vizinhos não visitados
    for (int vizinho : adjacencias[atual]) {
      if (!visitados[vizinho]) {
        dfsRecursivo(vizinho, visitados);
      }
    }
  }

  // Método principal: cria o grafo a partir dos argumentos e executa BFS e DFS
  public static void main(String[] args) {
    int vertices = Integer.parseInt(args[0]); // Número de vértices
    var grafo = new Grafo(vertices);

    // Lê as arestas do grafo a partir dos argumentos
    for (int i = 1; i < args.length; i += 2) {
      int origem = Integer.parseInt(args[i]);
      int destino = Integer.parseInt(args[i + 1]);
      grafo.adicionarAresta(origem, destino); // Adiciona cada aresta ao grafo
    }

    grafo.bfs(0); // Executa BFS a partir do vértice 0
    grafo.dfs(0); // Executa DFS a partir do vértice 0
  }
}
```

Algoritmo de Prim — Árvore Geradora Mínima

Definição

O **algoritmo de Prim** é um algoritmo **guloso (greedy)** utilizado para encontrar uma **Árvore Geradora Mínima (AGM)** de um **grafo conectado, não direcionado e ponderado**.

A AGM é um subconjunto de arestas que conecta todos os vértices do grafo, **sem formar ciclos**, com o **menor custo total possível** (soma dos pesos das arestas).

Objetivo

Selecionar um conjunto de arestas que:

- Conecte todos os vértices do grafo.
- Tenha o menor **peso total** possível.
- Forme uma estrutura **sem ciclos** (ou seja, uma árvore).

Etapas do Algoritmo de Prim

1. Inicialização

- Escolha um vértice de origem.
- Marque todos os outros vértices como **não visitados**.
- Crie uma **fila de prioridade (min-heap)** para armazenar as arestas candidatas, priorizando as de menor peso.

2. Laço Principal

Enquanto a fila não estiver vazia:

1. **Remova a aresta de menor peso** da fila.
2. Se o vértice de destino **ainda não foi visitado**:
    - Marque-o como visitado.
    - Adicione o peso da aresta ao **custo total da árvore**.
    - Adicione à fila todas as arestas **que saem desse novo vértice** e levam a vértices ainda não visitados.

3. Finalização

- Quando todos os vértices estiverem visitados, a AGM estará completa.
- O algoritmo retorna o **custo total da AGM** (e, opcionalmente, as arestas usadas).

Características

Característica	Valor
Tipo de grafo	Não direcionado e ponderado
Requisitos	Grafo conectado
Complexidade (com heap)	O((V + E) log V)
Estratégia	**Gulosa (greedy)**
Resultado	Árvore Geradora Mínima (AGM)

```java
import java.util.*;

// Classe que representa uma aresta no grafo (vizinho + peso)
class ArestaPrim {
  int destino, peso;

  ArestaPrim(int destino, int peso) {
    this.destino = destino;
    this.peso = peso;
  }
}

// Classe auxiliar que representa um par (vértice, peso) para a fila de prioridade
class Par implements Comparable<Par> {
  int vertice, peso;

  Par(int vertice, int peso) {
    this.vertice = vertice;
    this.peso = peso;
  }

  // Compara dois pares com base no peso (usado pela fila de prioridade)
  @Override
```

```
  public int compareTo(Par outro) {
    return Integer.compare(this.peso, outro.peso); // menor peso vem primeiro
  }
}

public class Prim {

  // Método principal que executa o algoritmo de Prim
  public int prim(int n, List<List<ArestaPrim>> grafo, int origem) {
    boolean[] visitado = new boolean[n]; // Vetor que marca quais vértices já estão na
árvore
    PriorityQueue<Par> fila = new PriorityQueue<>(); // Fila de prioridade para pegar a
menor aresta
    fila.offer(new Par(origem, 0)); // Começa pela origem com custo 0
    int custoTotal = 0; // Acumulador do custo total da AGM

    // Enquanto ainda houver vértices candidatos
    while (!fila.isEmpty()) {
      Par atual = fila.poll(); // Pega a aresta de menor peso da fila

      if (visitado[atual.vertice]) continue; // Se já foi visitado, ignora

      visitado[atual.vertice] = true; // Marca o vértice como incluído na árvore
      custoTotal += atual.peso;       // Soma o peso da aresta ao custo total

      // Explora todos os vizinhos do vértice atual
      for (ArestaPrim vizinho : grafo.get(atual.vertice)) {
        // Se o vizinho ainda não está na árvore
        if (!visitado[vizinho.destino]) {
          // Adiciona essa aresta como possível escolha futura
          fila.offer(new Par(vizinho.destino, vizinho.peso));
        }
      }
    }

    return custoTotal; // Retorna o custo total da árvore geradora mínima
  }

  public static void main(String[] args) {
    int n = 5; // Número de vértices no grafo

    // Criação do grafo como lista de adjacência
    List<List<ArestaPrim>> grafo = new ArrayList<>();
    for (int i = 0; i < n; i++) grafo.add(new ArrayList<>());

    // Adição das arestas (grafo não direcionado, por isso dupla inserção)
    grafo.get(0).add(new ArestaPrim(1, 10));
    grafo.get(1).add(new ArestaPrim(0, 10));

    grafo.get(0).add(new ArestaPrim(2, 6));
    grafo.get(2).add(new ArestaPrim(0, 6));
```

```
    grafo.get(0).add(new ArestaPrim(3, 5));
    grafo.get(3).add(new ArestaPrim(0, 5));

    grafo.get(1).add(new ArestaPrim(3, 15));
    grafo.get(3).add(new ArestaPrim(1, 15));

    grafo.get(2).add(new ArestaPrim(3, 4));
    grafo.get(3).add(new ArestaPrim(2, 4));

    // Instancia a classe e executa o algoritmo de Prim a partir do vértice 0
    var prim = new Prim();
    int custo = prim.prim(n, grafo, 0);

    // Imprime o custo total da Árvore Geradora Mínima
    System.out.println("Custo total da AGM (Prim): " + custo);
  }
}
```