**Problem Statement**

Imagine you are working on a project that involves analyzing various interconnected data points, represented as nodes, with connections between them, represented as edges.

Your task is to find the minimum spanning tree (MST) of this network, which is a subset of edges that connects all nodes without forming any cycles and has the minimum possible total edge weight.

To achieve this, you need to write a program that uses Prim's Algorithm to find the MST of a given graph.

**Input format :**

The first line of input consists of an integer **V,**representing the number of nodes.

The following **V** lines contain **V** space-separated integers each, representing the weighted connections between nodes.

The weight between node i and node j is provided in the ith row and jth column of the input.

**Output format :**

The output should print the edges and their corresponding weights in the minimum spanning tree (MST).

**Refer to the sample output for the exact format.**

**Code constraints :**

1 ≤ V ≤ 10

0 ≤ weight[i][j] ≤ 50

weight[i][i] = 0

**Sample test cases :**

**Input 1 :**

3

0 2 3

2 0 4

3 4 0

**Output 1 :**

Edge Weight

0 - 1 2

0 - 2 3

**Input 2 :**

5

0 2 0 6 0

2 0 3 8 5

0 3 0 0 7

6 8 0 0 9

0 5 7 9 0

**Output 2 :**

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

import java.util.Arrays; // Import Arrays utility class

import java.util.PriorityQueue; // Import PriorityQueue class for min-heap implementation

import java.util.Scanner; // Import Scanner class for input

// Graph class to represent the graph and its operations

class Graph {

    private int V; // Number of vertices

    private int[][] graph; // Adjacency matrix representation of the graph

    // Constructor to initialize the graph

    public Graph(int V, int[][] graph) {

        this.V = V; // Set number of vertices

        this.graph = graph; // Set adjacency matrix

    }

    // Function to find the MST using Prim's algorithm

    public void primMST() {

        int[] parent = new int[V]; // Array to store the constructed MST

        int[] key = new int[V]; // Key values used to pick minimum weight edge in cut

        boolean[] mstSet = new boolean[V]; // To represent set of vertices included in MST

        Arrays.fill(key, Integer.MAX\_VALUE); // Initialize all keys as INFINITE

        key[0] = 0; // Make key 0 so that this vertex is picked as first vertex

        parent[0] = -1; // First node is always root of MST

        // PriorityQueue to pick the minimum key vertex

        PriorityQueue<Pair> pq = new PriorityQueue<>((a, b) -> a.key - b.key);

        pq.add(new Pair(0, 0)); // Add first vertex to the priority queue

        while (!pq.isEmpty()) {

            int u = pq.poll().vertex; // Extract the vertex with the smallest key value

            mstSet[u] = true; // Include the picked vertex in MST set

            // Update key value and parent index of the adjacent vertices of the picked vertex

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

                // graph[u][v] is non-zero only for adjacent vertices of u

                // mstSet[v] is false for vertices not yet included in MST

                // Update the key only if graph[u][v] is smaller than key[v]

                if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {

                    parent[v] = u; // Update parent to the current vertex u

                    key[v] = graph[u][v]; // Update key to the weight of edge u-v

                    pq.add(new Pair(v, key[v])); // Add the vertex to the priority queue

                }

            }

        }

        // Print the constructed MST

        printMST(parent);

    }

    // A utility function to print the constructed MST stored in parent[]

    private void printMST(int[] parent) {

        System.out.println("Edge   Weight");

        for (int i = 1; i < V; i++) {

            System.out.println(parent[i] + " - " + i + "    " + graph[i][parent[i]]);

        }

    }

    // A class to represent a pair of vertex and key for priority queue

    class Pair {

        int vertex; // Vertex number

        int key; // Key value

        // Constructor to initialize Pair

        Pair(int vertex, int key) {

            this.vertex = vertex;

            this.key = key;

        }

    }

}

// Main class to test the above Graph class

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in); // Scanner object for input

        int V = sc.nextInt(); // Read number of vertices

        int[][] graph = new int[V][V]; // Initialize adjacency matrix

        for (int i = 0; i < V; i++) {

            for (int j = 0; j < V; j++) {

                graph[i][j] = sc.nextInt(); // Read the adjacency matrix

            }

        }

        Graph g = new Graph(V, graph); // Create a Graph object

        g.primMST(); // Find and print the MST

        sc.close(); // Close the scanner

    }

}

**Explanation:**

1. **Imports**:
   * import java.util.Arrays;: Used to fill arrays with initial values.
   * import java.util.PriorityQueue;: Used to implement a priority queue, which helps in selecting the next minimum weight edge efficiently.
   * import java.util.Scanner;: Used for reading input from the user.
2. **Graph Class**:
   * Graph(int V, int[][] graph): Constructor to initialize the number of vertices V and the adjacency matrix graph.
   * primMST(): Function to find and print the Minimum Spanning Tree (MST) using Prim's algorithm.
     + int[] parent: Array to store the MST.
     + int[] key: Array to store the minimum weight to add each vertex to the MST.
     + boolean[] mstSet: Array to keep track of vertices included in the MST.
     + PriorityQueue<Pair> pq: Priority queue to pick the vertex with the minimum key value.
     + The algorithm initializes all keys as infinite, except the first vertex which is set to 0. It then continuously picks the smallest key vertex, adds it to the MST, and updates the key values of its adjacent vertices.
   * printMST(int[] parent): Function to print the MST edges and their weights.
   * Pair Class: Represents a vertex and its key value for the priority queue.
3. **Main Class**:
   * Reads the number of vertices and the adjacency matrix from the user.
   * Creates a Graph object and calls primMST() to find and print the MST.

**Problem Statement**

**Prim’s Algorithm for Minimum Spanning Tree (MST):**

Given an undirected weighted graph with V vertices and E edges, the task is to find the minimum spanning tree (MST) of the graph. ﻿

**Input format :**

The input of V lines containing V space-separated integers, representing the adjacency matrix of the graph.

Note: V is 5 for all the test cases.

**Output format :**

The minimum spanning tree in the form of edges and their weights is displayed.

**Refer to the sample output for format specifications.**

**Sample test cases :**

**Input 1 :**

0 2 0 6 0

2 0 3 8 5

0 3 0 8 7

6 8 0 0 9

0 5 7 9 0

**Output 1 :**

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

import java.util.Arrays; // Import Arrays utility class

import java.util.PriorityQueue; // Import PriorityQueue class for min-heap implementation

import java.util.Scanner; // Import Scanner class for input

// Graph class to represent the graph and its operations

class Graph {

    private int V; // Number of vertices

    private int[][] graph; // Adjacency matrix representation of the graph

    // Constructor to initialize the graph

    public Graph(int V, int[][] graph) {

        this.V = V; // Set number of vertices

        this.graph = graph; // Set adjacency matrix

    }

    // Function to find the MST using Prim's algorithm

    public void primMST() {

        int[] parent = new int[V]; // Array to store the constructed MST

        int[] key = new int[V]; // Key values used to pick minimum weight edge in cut

        boolean[] mstSet = new boolean[V]; // To represent set of vertices included in MST

        Arrays.fill(key, Integer.MAX\_VALUE); // Initialize all keys as INFINITE

        key[0] = 0; // Make key 0 so that this vertex is picked as first vertex

        parent[0] = -1; // First node is always root of MST

        // PriorityQueue to pick the minimum key vertex

        PriorityQueue<Pair> pq = new PriorityQueue<>((a, b) -> a.key - b.key);

        pq.add(new Pair(0, 0)); // Add first vertex to the priority queue

        while (!pq.isEmpty()) {

            int u = pq.poll().vertex; // Extract the vertex with the smallest key value

            mstSet[u] = true; // Include the picked vertex in MST set

            // Update key value and parent index of the adjacent vertices of the picked vertex

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

                // graph[u][v] is non-zero only for adjacent vertices of u

                // mstSet[v] is false for vertices not yet included in MST

                // Update the key only if graph[u][v] is smaller than key[v]

                if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {

                    parent[v] = u; // Update parent to the current vertex u

                    key[v] = graph[u][v]; // Update key to the weight of edge u-v

                    pq.add(new Pair(v, key[v])); // Add the vertex to the priority queue

                }

            }

        }

        // Print the constructed MST

        printMST(parent);

    }

    // A utility function to print the constructed MST stored in parent[]

    private void printMST(int[] parent) {

        System.out.println("Edge\tWeight");

        for (int i = 1; i < V; i++) {

            System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);

        }

    }

    // A class to represent a pair of vertex and key for priority queue

    class Pair {

        int vertex; // Vertex number

        int key; // Key value

        // Constructor to initialize Pair

        Pair(int vertex, int key) {

            this.vertex = vertex;

            this.key = key;

        }

    }

}

// Main class to test the above Graph class

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in); // Scanner object for input

        // Number of vertices is fixed at 5

        int V = 5;

        int[][] graph = new int[V][V]; // Initialize adjacency matrix

        for (int i = 0; i < V; i++) {

            for (int j = 0; j < V; j++) {

                graph[i][j] = sc.nextInt(); // Read the adjacency matrix

            }

        }

        Graph g = new Graph(V, graph); // Create a Graph object

        g.primMST(); // Find and print the MST

        sc.close(); // Close the scanner

    }

}

**Explanation:**

1. **Imports**:
   * import java.util.Arrays;: Used to fill arrays with initial values.
   * import java.util.PriorityQueue;: Used to implement a priority queue, which helps in selecting the next minimum weight edge efficiently.
   * import java.util.Scanner;: Used for reading input from the user.
2. **Graph Class**:
   * Graph(int V, int[][] graph): Constructor to initialize the number of vertices V and the adjacency matrix graph.
   * primMST(): Function to find and print the Minimum Spanning Tree (MST) using Prim's algorithm.
     + int[] parent: Array to store the MST.
     + int[] key: Array to store the minimum weight to add each vertex to the MST.
     + boolean[] mstSet: Array to keep track of vertices included in the MST.
     + PriorityQueue<Pair> pq: Priority queue to pick the vertex with the minimum key value.
     + The algorithm initializes all keys as infinite, except the first vertex which is set to 0. It then continuously picks the smallest key vertex, adds it to the MST, and updates the key values of its adjacent vertices.
   * printMST(int[] parent): Function to print the MST edges and their weights.
   * Pair Class: Represents a vertex and its key value for the priority queue.
3. **Main Class**:
   * Reads the number of vertices (fixed at 5) and the adjacency matrix from the user.
   * Creates a Graph object and calls primMST() to find and print the MST.

**Problem Statement**

Jane is managing a new railway network that connects several towns. She wants to find the minimum spanning tree (MST) of this network, which will help optimize the overall cost of constructing the railway connections between towns. Jane needs your help to implement Kruskal's algorithm to find the MST.

Write a program that takes input representing the railway network between towns and outputs the edges forming the MST.

**Input format :**

The first line contains an integer **N** representing the number of towns in the network.

The next **N** lines contain **N** space-separated integers each, cost[i][j] representing the cost of constructing the railway between towns. If there's no direct connection between two towns, the cost is represented as 0. If there's a connection, but the cost is unknown, it's represented as 9999.

**Output format :**

The output displays the edges forming the minimum spanning tree.

Each edge should be represented as "TownA -> TownB" where TownA and TownB are the indices of the towns (1-indexed) forming the edge.

**Refer to the sample output for the formatting specifications.**

**Code constraints :**

In this scenario, the given test cases will fall under the following constraints:

2 ≤ N ≤ 20

-50 ≤ cost[i][j] ≤ 50

cost[i][i] = 0

**Sample test cases :**

**Input 1 :**

3

0 10 20

10 0 15

20 15 0

**Output 1 :**

1 -> 2

2 -> 3

**Input 2 :**

4

0 10 20 30

10 0 15 25

20 15 0 35

30 25 35 0

**Output 2 :**

1 -> 2

2 -> 3

2 -> 4

import java.util.\*; // Import all necessary utilities

// Class to represent an edge in the graph

class Edge implements Comparable<Edge> {

    int src, dest, weight; // Source, destination, and weight of the edge

    // Constructor to initialize an edge

    public Edge(int src, int dest, int weight) {

        this.src = src;

        this.dest = dest;

        this.weight = weight;

    }

    // Method to compare edges based on their weight

    public int compareTo(Edge compareEdge) {

        return this.weight - compareEdge.weight;

    }

}

// Class to represent the graph

class Graph {

    private int V; // Number of vertices

    private List<Edge> edges; // List of all edges

    // Constructor to initialize the graph with given vertices

    public Graph(int V) {

        this.V = V;

        this.edges = new ArrayList<>(); // Initialize the list of edges

    }

    // Method to add an edge to the graph

    public void addEdge(int src, int dest, int weight) {

        edges.add(new Edge(src, dest, weight)); // Create and add an edge to the list

    }

    // Method to find the MST using Kruskal's algorithm

    public void kruskalMST() {

        List<Edge> result = new ArrayList<>(); // To store the resulting MST

        // Sort all the edges in non-decreasing order of their weight

        Collections.sort(edges);

        // Create subsets for union-find

        int[] parent = new int[V];

        int[] rank = new int[V];

        // Initialize each vertex as its own parent and rank as 0

        for (int i = 0; i < V; i++) {

            parent[i] = i;

            rank[i] = 0;

        }

        int e = 0; // Number of edges in MST

        int i = 0; // Index for sorted edges

        // Process edges in sorted order

        while (e < V - 1 && i < edges.size()) {

            Edge nextEdge = edges.get(i++); // Pick the smallest edge

            // Find the subsets of the vertices of the edge

            int x = find(parent, nextEdge.src);

            int y = find(parent, nextEdge.dest);

            // If including this edge doesn't cause a cycle

            if (x != y) {

                result.add(nextEdge); // Include the edge in MST

                e++; // Increment the count of edges in MST

                union(parent, rank, x, y); // Union the sets

            }

        }

        // Print the resulting MST

        for (Edge edge : result) {

            System.out.println((edge.src + 1) + " -> " + (edge.dest + 1));

        }

    }

    // Utility method to find the subset of an element i using path compression

    private int find(int[] parent, int i) {

        if (parent[i] != i) {

            parent[i] = find(parent, parent[i]); // Recursively find the root and update parent

        }

        return parent[i];

    }

    // Utility method to do union of two sets x and y using union by rank

    private void union(int[] parent, int[] rank, int x, int y) {

        int rootX = find(parent, x); // Find root of x

        int rootY = find(parent, y); // Find root of y

        // Attach smaller rank tree under root of high rank tree

        if (rank[rootX] < rank[rootY]) {

            parent[rootX] = rootY;

        } else if (rank[rootX] > rank[rootY]) {

            parent[rootY] = rootX;

        } else {

            parent[rootY] = rootX; // If ranks are same, make one root and increment its rank

            rank[rootX]++;

        }

    }

}

// Main class to test the above Graph class

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in); // Scanner object for input

        // Read the number of towns (vertices)

        int N = sc.nextInt();

        // Create a graph with N vertices

        Graph graph = new Graph(N);

        // Read the adjacency matrix

        for (int i = 0; i < N; i++) {

            for (int j = 0; j < N; j++) {

                int cost = sc.nextInt();

                if (cost > 0 && cost != 9999) { // Consider only valid connections

                    graph.addEdge(i, j, cost); // Add edge to the graph

                }

            }

        }

        // Find and print the MST using Kruskal's algorithm

        graph.kruskalMST();

        sc.close(); // Close the scanner

    }

}

**Explanation:**

1. **Imports**:
   * import java.util.\*;: Imports all necessary utilities including ArrayList, Scanner, Collections, etc.
2. **Edge Class**:
   * Represents an edge with source (src), destination (dest), and weight (weight).
   * Implements Comparable<Edge> to compare edges based on their weight.
3. **Graph Class**:
   * **Constructor**: Initializes the graph with the given number of vertices V and an empty list of edges.
   * addEdge(int src, int dest, int weight): Adds an edge to the graph.
   * kruskalMST(): Implements Kruskal's algorithm to find the MST.
     + Sorts all edges in non-decreasing order of their weight.
     + Initializes parent and rank arrays for union-find operations.
     + Processes each edge and adds it to the MST if it does not form a cycle.
     + Uses the find method for path compression and union method for union by rank.
   * find(int[] parent, int i): Finds the subset (root) of an element i using path compression.
   * union(int[] parent, int[] rank, int x, int y): Performs the union of two sets using union by rank.
4. **Main Class**:
   * Reads the number of towns (vertices) from the user.
   * Reads the adjacency matrix representing the graph.
   * Creates a Graph object and adds edges to it based on the adjacency matrix.
   * Calls kruskalMST() to find and print the MST.

This program reads an adjacency matrix, constructs the graph, and then finds the Minimum Spanning Tree using Kruskal's algorithm, printing the edges of the MST in the specified format.

Let's break down the provided Java code, focusing on each component and its role in the implementation of Kruskal's algorithm for finding the Minimum Spanning Tree (MST) of a graph.

**1. Imports**

java

Copy code

import java.util.\*; // Import all necessary utilities

* **Purpose**: This imports Java's utility classes, including ArrayList, Scanner, and Collections, which are needed for managing lists, reading input, and sorting data.
* **Alternatives**: You could import specific classes individually if you only need a few of them, e.g., import java.util.ArrayList; import java.util.Scanner;.

**2. Edge Class**

java

Copy code

class Edge implements Comparable<Edge> {

int src, dest, weight; // Source, destination, and weight of the edge

public Edge(int src, int dest, int weight) {

this.src = src;

this.dest = dest;

this.weight = weight;

}

public int compareTo(Edge compareEdge) {

return this.weight - compareEdge.weight;

}

}

* **Purpose**:
  + **Attributes**:
    - src: The starting vertex of the edge.
    - dest: The ending vertex of the edge.
    - weight: The weight or cost associated with the edge.
  + **Constructor**: Initializes the edge with source, destination, and weight values.
  + **compareTo Method**: Allows edges to be sorted by weight. This is crucial for Kruskal's algorithm, which processes edges in ascending order of weight.
* **Alternatives**:
  + Instead of implementing Comparable<Edge>, you could use a separate Comparator<Edge> when sorting, which provides more flexibility.
* **Error Handling**:
  + **Without compareTo**: You would need to provide a custom comparator wherever sorting is required. Kruskal's algorithm relies on sorting edges by weight, so this method is essential.

**3. Graph Class**

java

Copy code

class Graph {

private int V; // Number of vertices

private List<Edge> edges; // List of all edges

public Graph(int V) {

this.V = V;

this.edges = new ArrayList<>(); // Initialize the list of edges

}

public void addEdge(int src, int dest, int weight) {

edges.add(new Edge(src, dest, weight)); // Create and add an edge to the list

}

public void kruskalMST() {

List<Edge> result = new ArrayList<>(); // To store the resulting MST

Collections.sort(edges); // Sort all the edges in non-decreasing order of their weight

int[] parent = new int[V];

int[] rank = new int[V];

for (int i = 0; i < V; i++) {

parent[i] = i;

rank[i] = 0;

}

int e = 0; // Number of edges in MST

int i = 0; // Index for sorted edges

while (e < V - 1 && i < edges.size()) {

Edge nextEdge = edges.get(i++); // Pick the smallest edge

int x = find(parent, nextEdge.src);

int y = find(parent, nextEdge.dest);

if (x != y) {

result.add(nextEdge); // Include the edge in MST

e++;

union(parent, rank, x, y); // Union the sets

}

}

for (Edge edge : result) {

System.out.println((edge.src + 1) + " -> " + (edge.dest + 1));

}

}

private int find(int[] parent, int i) {

if (parent[i] != i) {

parent[i] = find(parent, parent[i]); // Path compression

}

return parent[i];

}

private void union(int[] parent, int[] rank, int x, int y) {

int rootX = find(parent, x); // Find root of x

int rootY = find(parent, y); // Find root of y

if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

}

}

* **Purpose**:
  + **Attributes**:
    - V: Number of vertices in the graph.
    - edges: List of all edges in the graph.
  + **Methods**:
    - addEdge: Adds an edge to the list of edges.
    - kruskalMST: Implements Kruskal's algorithm to find the MST.
      * **Sorting**: Collections.sort(edges) sorts edges by weight.
      * **Union-Find**: find and union methods manage the connected components and prevent cycles.
    - **find**: Uses path compression to make subsequent finds faster.
    - **union**: Uses union by rank to keep the tree shallow.
* **Alternatives**:
  + **Edge List**: For the graph representation, an adjacency matrix could be used instead of an edge list, but this might be less efficient for Kruskal's algorithm.
  + **Sorting**: You could use a custom comparator or another sorting algorithm if needed.
* **Error Handling**:
  + **Without find and union**: Kruskal's algorithm would not be able to correctly manage cycles in the MST, leading to incorrect results.

**4. Main Class**

java

Copy code

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in); // Scanner object for input

int N = sc.nextInt(); // Read the number of vertices

Graph graph = new Graph(N); // Create a graph with N vertices

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

int cost = sc.nextInt();

if (cost > 0 && cost != 9999) { // Consider only valid connections

graph.addEdge(i, j, cost); // Add edge to the graph

}

}

}

graph.kruskalMST(); // Find and print the MST

sc.close(); // Close the scanner

}

}

* **Purpose**:
  + **Input Handling**: Reads the number of vertices and the adjacency matrix from the user.
  + **Graph Construction**: Creates a Graph object and populates it with edges based on the input matrix.
  + **MST Calculation**: Calls kruskalMST() to compute and print the MST.
  + **Scanner Closure**: Closes the Scanner object to free resources.
* **Alternatives**:
  + **Input Method**: You could use file input or another data source instead of reading from standard input.
* **Error Handling**:
  + **Without Proper Input Handling**: If the input format is incorrect or invalid, the program might throw exceptions or produce incorrect results.

**Summary**

* **Data Types**:
  + **List<Edge>**: Used to store edges in the graph.
  + **int[]**: Used for parent and rank arrays in the union-find structure.
  + **Edge Class**: Represents an edge with source, destination, and weight.
* **Methods and Functions**:
  + **compareTo**: Used for sorting edges.
  + **find**: Implements path compression to optimize union-find operations.
  + **union**: Implements union by rank to keep the union-find tree shallow.
* **Alternatives**:
  + **Edge List vs. Adjacency Matrix**: An edge list is more suitable for Kruskal's algorithm than an adjacency matrix.
  + **Sorting**: You could use different sorting mechanisms if needed.

Understanding these components and their roles is key to grasping how Kruskal's algorithm functions and why each part is necessary.