**Batch- T5**

**Practical No. 7**

**Name: Siya Pondkule**

**PRN: 23520008**

**Greedy Method**

1) Implement Kruskal’s algorithm & Prim's algorithm to find Minimum Spanning Tree (*MST*) of the given an undirected, connected and weighted graph.



**Kruskals Algorithm**

**Steps:**

1. **Sort all edges:**
   1. Sort all edges in increasing order based on their weights.
2. **Initialize Disjoint Set (Union-Find):**
   1. Create a Union-Find data structure to keep track of connected components.
   2. Initially, each vertex is in its own set.
3. **Add the smallest edge:**
   1. For each edge (u, v) in the sorted edge list:
      1. If u and v belong to different components (sets), add the edge to the MST.
      2. Union the sets containing u and v to combine them into one set.
4. **Repeat the process:**
   1. Repeat step 3 until the number of edges in the MST equals V-1.
5. **Termination:**

5.1 When the number of edges in the MST equals V-1, stop.

* 1. Return the total cost of the MST.

import java.util.*\**;

class kruskals {

    class Edge implements Comparable<Edge> {

        int src, dest, weight;

        public int compareTo(Edge compareEdge) {

            return *this*.weight - compareEdge.weight;

        }

    }

    class Subset {

        int parent, rank;

    }

    int V, E;

    Edge[] edge;

    kruskals(int v, int e) {

        V = v;

        E = e;

        edge = new Edge[E];

        for (int i = 0; i < e; i++) {

            edge[i] = new Edge();

        }

    }

    int find(Subset[] subsets, int i) {

        if (subsets[i].parent != i)

            subsets[i].parent = find(subsets, subsets[i].parent);

        return subsets[i].parent;

    }

    void union(Subset[] subsets, int x, int y) {

        int xroot = find(subsets, x);

        int yroot = find(subsets, y);

        if (subsets[xroot].rank < subsets[yroot].rank)

            subsets[xroot].parent = yroot;

        else if (subsets[xroot].rank > subsets[yroot].rank)

            subsets[yroot].parent = xroot;

        else {

            subsets[yroot].parent = xroot;

            subsets[xroot].rank++;

        }

    }

    void kruskalMST() {

        Edge[] result = new Edge[V];

int e = 0;

int totalCost = 0

        Arrays.sort(edge);

        Subset[] subsets = new Subset[V];

        for (int v = 0; v < V; v++) {

            subsets[v] = new Subset();

            subsets[v].parent = v;

            subsets[v].rank = 0;

        }

        int i = 0;

        while (e < V - 1) {

            Edge next\_edge = edge[i++];

            int x = find(subsets, next\_edge.src);

            int y = find(subsets, next\_edge.dest);

            if (x != y) {

                result[e++] = next\_edge;

                union(subsets, x, y);

                totalCost += next\_edge.weight;

            }

        }

        System.out.println("Edges in the MST:");

        for (i = 0; i < e; i++)

            System.out.println(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);

        System.out.println("Minimum cost (total weight) of MST: " + totalCost);

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of vertices:");

        int V = sc.nextInt();

        System.out.println("Enter the number of edges:");

        int E = sc.nextInt();

        kruskals graph = new kruskals(V, E);

        System.out.println("Enter the edges with their respective weights (src dest weight):");

        for (int i = 0; i < E; i++) {

            graph.edge[i].src = sc.nextInt();

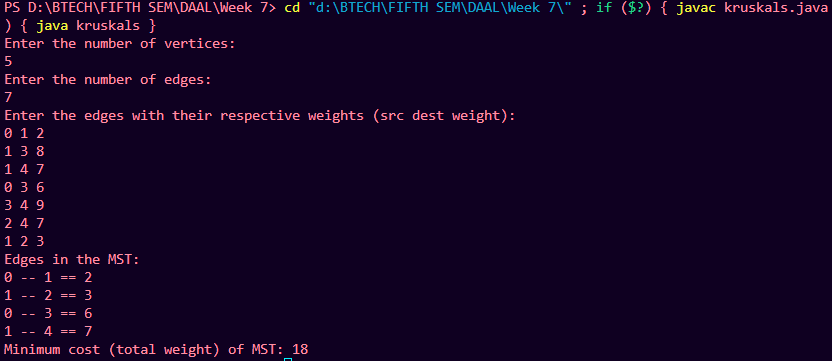
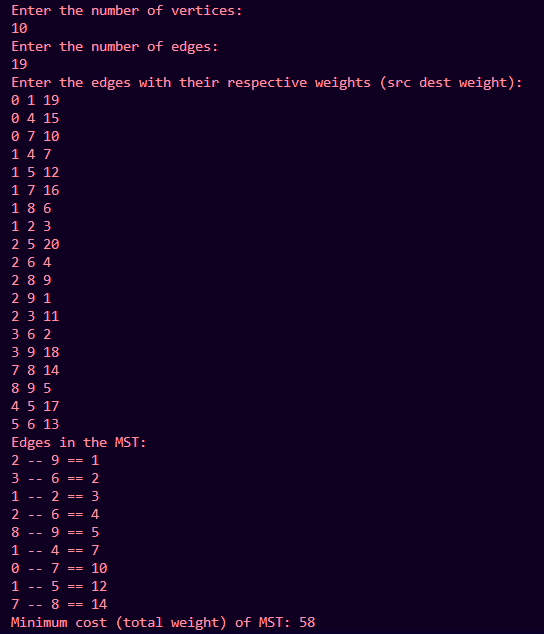
            graph.edge[i].dest = sc.nextInt();

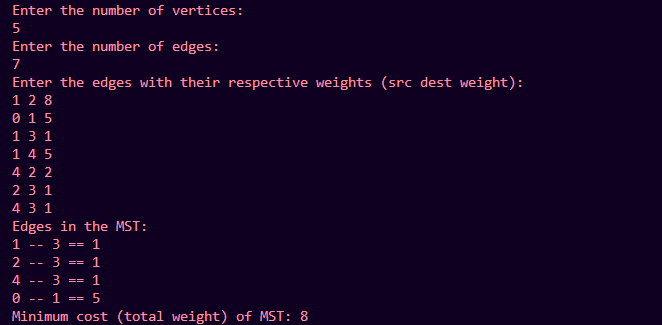
            graph.edge[i].weight = sc.nextInt();

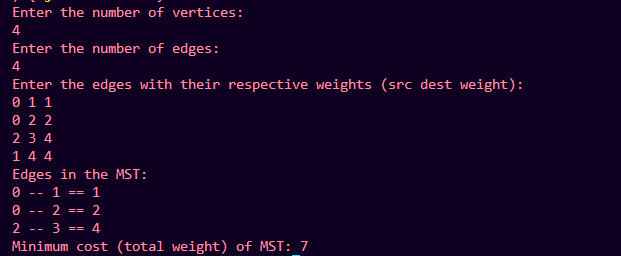
        }

        graph.kruskalMST();

    }

}  






**Prims Algorithm**

**Steps:**

1. **Initialization:**
   1. Create a priority queue (min-heap) to select the smallest edge.
   2. Initialize a boolean visited[] array of size V to keep track of visited nodes.
   3. Initialize MST cost = 0 to store the cost of the MST.
2. **Start from an arbitrary node:**
   1. Choose a node (e.g., node 0) as the starting node.
   2. Mark node 0 as visited in the visited[] array.
   3. Add all edges from node 0 to the priority queue.
3. **Add the smallest edge:**
   1. While there are unvisited nodes and the priority queue is not empty:
      1. Extract the edge with the smallest weight from the priority queue
      2. If the connected node is already visited, skip this edge.
      3. Otherwise:
   2. Mark the connected node as visited.
   3. Add the edge’s weight to MST cost.
   4. Add all adjacent edges from the newly visited node to the priority queue.
4. **Repeat the process:**
   1. Repeat step 3 until all vertices are visited or the number of edges in the MST equals V-1.
5. **Termination:**
   1. When the number of edges added to the MST equals V-1, stop.
   2. Return the total MST cost.

import java.util.*\**;

class PrimsAlgorithm {

    class Edge {

        int dest, weight;

        Edge(int dest, int weight) {

*this*.dest = dest;

*this*.weight = weight;

        }

    }

    class Node implements Comparable<Node> {

        int vertex, key;

        Node(int vertex, int key) {

*this*.vertex = vertex;

*this*.key = key;

        }

        public int compareTo(Node other) {

            return *this*.key - other.key;

        }

    }

    int V;

    List<List<Edge>> adj;

    PrimsAlgorithm(int V) {

*this*.V = V;

        adj = new ArrayList<>();

        for (int i = 0; i < V; i++) {

            adj.add(new ArrayList<>());

        }

    }

    void addEdge(int u, int v, int weight) {

        adj.get(u).add(new Edge(v, weight));

        adj.get(v).add(new Edge(u, weight));    }

    void primMST() {

        boolean[] mstSet = new boolean[V];

        int[] key = new int[V];

        int[] parent = new int[V];

        int totalCost =0;

PriorityQueue<Node> pq = new PriorityQueue<>(V);

        Arrays.fill(key, Integer.MAX\_VALUE);

        key[0] = 0;

        pq.add(new Node(0, key[0]));

        while (!pq.isEmpty()) {

            int u = pq.poll().vertex;

            mstSet[u] = true;

            totalCost += key[u];

            for (Edge edge : adj.get(u)) {

                int v = edge.dest;

                int weight = edge.weight;

                if (!mstSet[v] && weight < key[v]) {

                    key[v] = weight;

                    pq.add(new Node(v, key[v]));

                    parent[v] = u;

                }

            }

        }

        System.out.println("Edges in the MST:");

        for (int i = 1; i < V; i++) {

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

        }

        System.out.println("Minimum cost (total weight) of MST: " + totalCost);

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of vertices:");

        int V = sc.nextInt();

        PrimsAlgorithm graph = new PrimsAlgorithm(V);

        System.out.println("Enter the number of edges:");

        int E = sc.nextInt();

        System.out.println("Enter the edges with their respective weights (src dest weight):");

        for (int i = 0; i < E; i++) {

            int u = sc.nextInt();

            int v = sc.nextInt();

            int weight = sc.nextInt();

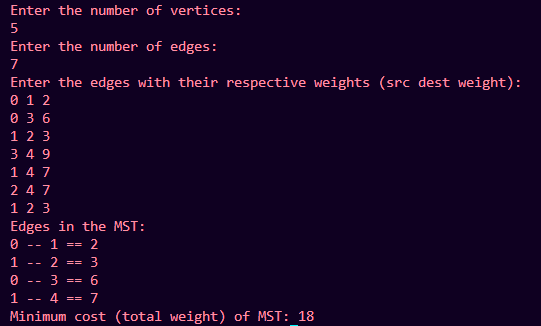
            graph.addEdge(u, v, weight);

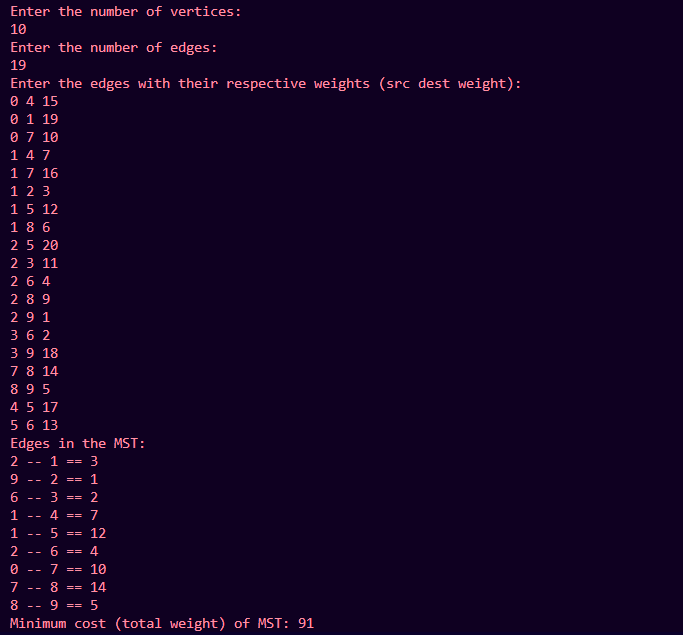
        }

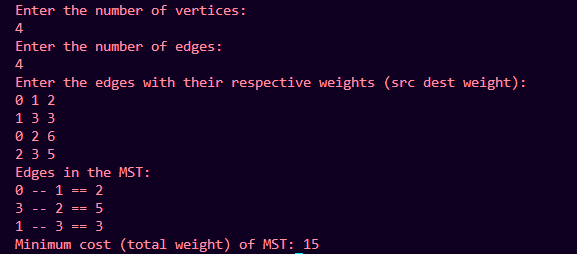
        graph.primMST();

    }

}







Q) How many edges does a minimum spanning tree for above example?

Total edges = 19

Q) In a graph *G*. let the edge *u v* have the least weight. is it true that *u v* is always part of any minimum spanning tree of *G*?.Justify your answers.

Yes, it is **true** that the edge u−vu - vu−v with the **least weight** will always be part of **any Minimum Spanning Tree (MST)** of a connected graph GGG

Jusification:

Both **Kruskal's algorithm** and **Prim's algorithm**, which are used to find the MST of a graph, are **greedy algorithms** that build the MST step by step by always adding the least weight edge that doesn't form a cycle or violate the properties of the MST.

* **Kruskal's Algorithm**: It starts by sorting all the edges by their weights and includes the smallest edge in the MST if it does not create a cycle. Since u−vu - vu−v is the edge with the least weight, Kruskal's algorithm will **always pick** this edge first because no smaller edge exists, and adding it will not create a cycle initially.
* **Prim's Algorithm**: Prim’s algorithm starts from an arbitrary vertex and keeps expanding the MST by choosing the smallest-weight edge that connects the current MST to a new vertex. If u−vu - vu−v is the smallest-weight edge in the entire graph, Prim's algorithm will pick this edge as soon as it encounters either vertex uuu or vvv, ensuring that the edge u−vu - vu−v becomes part of the MST.

Q) Let *G* be a graph and T be a minimum spanning tree of *G*. Suppose that the weight of an edge e is decreased. How can you find the minimum spanning tree of the modified graph? What is the runtime of your solution?

When the weight of an edge eee in the graph GGG is decreased, the Minimum Spanning Tree (MST) of the modified graph can change. The steps to update the MST after decreasing the weight of eee depend on whether eee is already part of the MST or not.

Case 1: Edge eee is already in the MST

If the edge eee is already part of the MST and its weight is decreased, the MST remains valid because reducing the weight of an edge that is already in the tree will not create any cycles or violate the MST properties. Therefore, the MST will not change, and no further steps are needed.

Runtime: The runtime in this case is O(1) since no recomputation is needed.

Case 2: Edge eee is not in the MST

If eee is not in the MST, decreasing its weight might make it the cheapest edge for some cut in the graph, meaning it could now become part of the MST. To handle this case, follow these steps:

Step 1: Add Edge eee to the MST

Include the newly lightened edge eee in the MST. Adding this edge may create a cycle because it connects two vertices already connected by a path in the current MST.

Step 2: Remove the Heaviest Edge in the Cycle

Once the new edge eee is added, a cycle will be formed. You need to remove the heaviest edge in the cycle to restore the tree property of the MST. This step ensures that the new tree remains a valid MST, as removing the heaviest edge will ensure the minimum total weight.

The key observation here is that by adding the lighter edge eee and removing the heaviest edge in the cycle, you are effectively minimizing the weight of the tree.

Finding the Cycle: Since the MST is a tree, finding the cycle created by adding the edge eee is straightforward. You can perform a depth-first search (DFS) or breadth-first search (BFS) to trace the cycle, which takes O(V) time, where VVV is the number of vertices.

Removing the Heaviest Edge: You also need to track the heaviest edge in the cycle, which can be done during the traversal of the cycle, taking O(V) time.

Final MST

Once the cycle is resolved by removing the heaviest edge, you will have the new MST of the modified graph.

Q) Find order of edges for Kruskal's and Prim's?

Prims Algorithm:

Edges in the MST:

2 -- 1 == 3

9 -- 2 == 1

6 -- 3 == 2

1 -- 4 == 7

1 -- 5 == 12

2 -- 6 == 4

0 -- 7 == 10

7 -- 8 == 14

8 -- 9 == 5

Kruskals Algorithm:

Edges in the MST:

2 -- 9 == 1

3 -- 6 == 2

1 -- 2 == 3

2 -- 6 == 4

8 -- 9 == 5

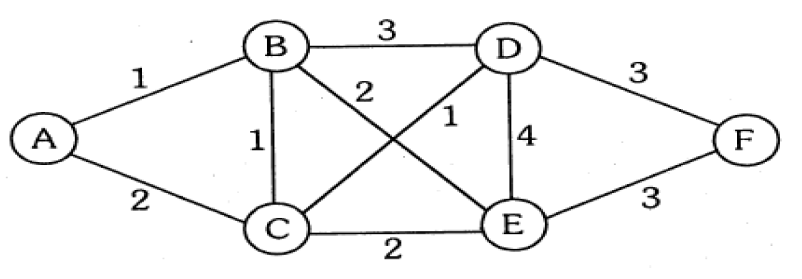
1 -- 4 == 7

0 -- 7 == 10

1 -- 5 == 12

7 -- 8 == 14

2) From a given vertex in a weighted connected graph, implement shortest path finding Dijkstra's algorithm.



**Algorithm:**

**Steps:**

1. **Initialization:**
   1. Create a distance[] array of size V and initialize all values to infinity (∞), except the source vertex, which is set to 0. This array keeps track of the shortest distance from the source to each vertex.
   2. Create a visited[] array of size V and initialize all values to false. This array keeps track of the vertices that have been processed.
   3. Use a priority queue (min-heap) to pick the next vertex with the smallest distance.
   4. Insert the source vertex into the priority queue with distance 0.
2. **While there are vertices to process in the priority queue:**
   1. Extract the vertex u from the priority queue with the smallest distance.
   2. Mark vertex u as visited: visited[u] = true.
   3. For each neighboring vertex v of u (i.e., each adjacent vertex of u):
      1. If v is not visited and the distance to v through u is shorter than the current known distance to v:
         1. Update the distance of v: distance[v] = distance[u] + weight(u, v).
         2. Add v to the priority queue with the updated distance.
3. **Repeat the process:**
   1. Continue extracting the vertex with the smallest distance from the priority queue and processing its neighbors, updating the distances and adding unvisited neighbors to the priority queue.
4. **Termination:**
   1. The algorithm terminates when all vertices have been processed or the priority queue is empty.
   2. The distance[] array now contains the shortest distances from the source vertex to every other vertex.

import java.util.*\**;

class DijkstraAlgorithm {

    private final int V;

    private final List<List<Edge>> adj;

    class Edge {

        int dest;

        int weight;

        Edge(int dest, int weight) {

*this*.dest = dest;

*this*.weight = weight;

        }

    }

    DijkstraAlgorithm(int V) {

*this*.V = V;

*this*.adj = new ArrayList<>();

        for (int i = 0; i < V; i++) {

            adj.add(new ArrayList<>());

        }

    }

    void addEdge(int u, int v, int weight) {

        adj.get(u).add(new Edge(v, weight));

        adj.get(v).add(new Edge(u, weight));

    }

    void dijkstra(int src) {

        int[] dist = new int[V];

        Arrays.fill(dist, Integer.MAX\_VALUE);

        dist[src] = 0;

        PriorityQueue<Node> pq = new PriorityQueue<>(V, new Node());

        pq.add(new Node(src, dist[src]));

        while (!pq.isEmpty()) {

            int u = pq.poll().vertex;

            for (Edge edge : adj.get(u)) {

                int v = edge.dest;

                int weight = edge.weight;

                if (dist[u] + weight < dist[v]) {

                    dist[v] = dist[u] + weight;

                    pq.add(new Node(v, dist[v]));

                }

            }

        }

        System.out.println("Vertex Distance from Source");

        for (int i = 0; i < V; i++) {

            System.out.println(i + "\t\t" + dist[i]);

        }

    }

    class Node implements Comparator<Node> {

        int vertex;

        int key;

        Node() {}

        Node(int vertex, int key) {

*this*.vertex = vertex;

*this*.key = key;

        }

        @Override

        public int compare(Node n1, Node n2) {

            return Integer.compare(n1.key, n2.key);

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of vertices:");

        int V = sc.nextInt();

        DijkstraAlgorithm graph = new DijkstraAlgorithm(V);

        System.out.println("Enter the number of edges:");

        int E = sc.nextInt();

        System.out.println("Enter the edges with their respective weights (src dest weight):");

        for (int i = 0; i < E; i++) {

            int u = sc.nextInt();

            int v = sc.nextInt();

            int weight = sc.nextInt();

            graph.addEdge(u, v, weight);

        }

        System.out.println("Enter the source vertex:");

        int source = sc.nextInt();

        graph.dijkstra(source);

    }

}







Q) Show that Dijkstra’s algorithm doesn’t work for graphs with negative weight edges

Dijkstra’s algorithm assumes that all edge weights are non-negative. When the graph contains negative weight edges, the algorithm may produce incorrect results.

Dijkstra's algorithm uses a greedy approach by selecting the vertex with the smallest distance from the source at each step. If a negative weight edge is encountered after selecting a vertex, Dijkstra's algorithm will not reconsider that vertex, potentially missing a shorter path that could be obtained through the negative edge.

import java.util.*\**;

class DijkstraAlgorithm {

    private final int V;

    private final List<List<Edge>> adj;

    class Edge {

        int dest;

        int weight;

        Edge(int dest, int weight) {

*this*.dest = dest;

*this*.weight = weight;

        }

    }

    DijkstraAlgorithm(int V) {

*this*.V = V;

*this*.adj = new ArrayList<>();

        for (int i = 0; i < V; i++) {

            adj.add(new ArrayList<>());

        }

    }

    void addEdge(int u, int v, int weight) {

        adj.get(u).add(new Edge(v, weight));

        adj.get(v).add(new Edge(u, weight));

    }

    void dijkstra(int src) {

        int[] dist = new int[V];

        Arrays.fill(dist, Integer.MAX\_VALUE);

        dist[src] = 0;

        PriorityQueue<Node> pq = new PriorityQueue<>(V, new Node());

        pq.add(new Node(src, dist[src]));

        while (!pq.isEmpty()) {

            int u = pq.poll().vertex;

            for (Edge edge : adj.get(u)) {

                int v = edge.dest;

                int weight = edge.weight;

                if (dist[u] + weight < dist[v]) {

                    dist[v] = dist[u] + weight;

                    pq.add(new Node(v, dist[v]));

                }

            }

        }

        System.out.println("Vertex Distance from Source");

        for (int i = 0; i < V; i++) {

            System.out.println(i + "\t\t" + dist[i]);

        }

    }

    class Node implements Comparator<Node> {

        int vertex;

        int key;

        Node() {}

        Node(int vertex, int key) {

*this*.vertex = vertex;

*this*.key = key;

        }

        @Override

        public int compare(Node n1, Node n2) {

            return Integer.compare(n1.key, n2.key);

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of vertices:");

        int V = sc.nextInt();

        DijkstraAlgorithm graph = new DijkstraAlgorithm(V);

        System.out.println("Enter the number of edges:");

        int E = sc.nextInt();

        System.out.println("Enter the edges with their respective weights (src dest weight):");

        for (int i = 0; i < E; i++) {

            int u = sc.nextInt();

            int v = sc.nextInt();

            int weight = sc.nextInt();

            graph.addEdge(u, v, weight);

        }

        System.out.println("Enter the source vertex:");

        int source = sc.nextInt();

        graph.dijkstra(source);

    }

}

Q) Modify the Dijkstra’s algorithm to find shortest path.

**Algorithm:**

**Steps:**

1. **Initialization:**
   1. Create a distance[] array of size V and initialize all values to infinity (∞), except the source vertex, which is set to 0. This array keeps track of the shortest distance from the source to each vertex.
   2. Create a previous[] array of size V and initialize all values to -1. This array keeps track of the previous node for each vertex on the shortest path.
   3. Create a visited[] array of size V and initialize all values to false to keep track of processed nodes.
   4. Use a priority queue (min-heap) to pick the next vertex with the smallest distance.
   5. Insert the source vertex into the priority queue with distance 0.
2. **While there are vertices to process in the priority queue:**
   1. Extract the vertex u from the priority queue with the smallest distance.
   2. Mark vertex u as visited: visited[u] = true.
   3. For each neighboring vertex v of u (i.e., each adjacent vertex of u):
      1. If v is not visited and the distance to v through u is shorter than the current known distance to v:
         1. Update the distance of v: distance[v] = distance[u] + weight(u, v).
         2. Update the previous node for v: previous[v] = u.
         3. Add v to the priority queue with the updated distance.
3. **Repeat the process:**
   1. Continue extracting the vertex with the smallest distance from the priority queue and processing its neighbors, updating the distances and adding unvisited neighbors to the priority queue.
4. **Reconstruct the shortest path:**
   1. To find the shortest path from the source vertex to any vertex v, trace back from vertex v using the previous[] array.
   2. Start from v, and move to previous[v] until you reach the source.
5. **Termination:**
   1. The algorithm terminates when all vertices have been processed or the priority queue is empty.
   2. The distance[] array contains the shortest distances from the source vertex to all other vertices.
   3. The previous[] array can be used to reconstruct the shortest path from the source to any target vertex.

import java.util.*\**;

class Dijkstra {

    private final int V;

    private final List<List<Node>> adj;

    static class Node {

        int vertex, weight;

        Node(int vertex, int weight) {

*this*.vertex = vertex;

*this*.weight = weight;

        }

    }

    Dijkstra(int V) {

*this*.V = V;

*this*.adj = new ArrayList<>();

        for (int i = 0; i < V; i++) {

            adj.add(new ArrayList<>());

        }

    }

    void addEdge(int u, int v, int weight) {

        adj.get(u).add(new Node(v, weight));

        adj.get(v).add(new Node(u, weight));    }

    void shortestPath(int src) {

        int[] dist = new int[V];

        Arrays.fill(dist, Integer.MAX\_VALUE);

        dist[src] = 0;

        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(node -> node.weight));

        pq.add(new Node(src, 0));

        while (!pq.isEmpty()) {

            int u = pq.poll().vertex;

            for (Node neighbor : adj.get(u)) {

                int v = neighbor.vertex;

                int weight = neighbor.weight;

                if (dist[u] + weight < dist[v]) {

                    dist[v] = dist[u] + weight;

                    pq.add(new Node(v, dist[v]));

                }

            }

        }

        System.out.println("Vertex Distance from Source");

        for (int i = 0; i < V; i++) {

            System.out.println(i + "\t\t" + dist[i]);

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of vertices:");

        int V = sc.nextInt();

        Dijkstra graph = new Dijkstra(V);

        System.out.println("Enter the number of edges:");

        int E = sc.nextInt();

        System.out.println("Enter the edges with their respective weights (src dest weight):");

        for (int i = 0; i < E; i++) {

            int u = sc.nextInt();

            int v = sc.nextInt();

            int weight = sc.nextInt();

            graph.addEdge(u, v, weight);

        }

        System.out.println("Enter the source vertex:");

        int src = sc.nextInt();

        graph.shortestPath(src);

    }

}

