Analysis of Algorithm

Practical no 7 :

Prism & kruskal's Algorithm

Prism Algorithm

Code :

```
import java.util.*;

public class PrimMST {

  // Number of vertices in the graph

  static final int V = 4;

  // A utility function to find the vertex with minimum key value, from the set of vertices not yet included in MST

  static int minKey(int key[], boolean mstSet[]) {

    // Initialize min value

    int min = Integer.MAX_VALUE, min_index = -1;

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

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

        min = key[v];

        min_index = v;

      }

    }

    return min_index;

  }

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

  static void printMST(int parent[], int graph[][]) {

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

    for (int i = 1; i < V; i++) {

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

    }

  }
```

```
  // Function to construct and print MST for a graph represented using adjacency matrix representation
  static void primMST(int graph[][]) {
    int parent[] = new int[V];
    int key[] = new int[V];
    boolean mstSet[] = new boolean[V];
    // Initialize all keys as INFINITE and mstSet[] as false
    Arrays.fill(key, Integer.MAX_VALUE);
    Arrays.fill(mstSet, false);
    // Always include the first vertex in MST. Make key 0 so that this vertex is picked as the first vertex
    key[0] = 0;
    parent[0] = -1;
    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
      // Pick the minimum key vertex from the set of vertices not yet included in MST
      int u = minKey(key, mstSet);
      // Add the picked vertex to the MST Set
      mstSet[u] = true;
      // Update the key value and parent index of the adjacent vertices of the picked vertex.
      // Consider only those vertices which are not yet included in MST
      for (int v = 0; v < V; v++) {
        // graph[u][v] is non-zero only for adjacent vertices, 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;
          key[v] = graph[u][v];
        }
      }
    }
    // Print the constructed MST
```

```
    printMST(parent, graph);
  }
  // Driver code
  public static void main(String[] args) {
    int graph[][] = {
      { 0, 2, 0, 6},
      { 2, 0, 3, 8},
      { 0, 3, 0, 0},
      { 6, 8, 0, 0},
    };
    // Print the solution
    primMST(graph);
  }
}
```

Output :

```
Edge   Weight
0 - 1    2
1 - 2    3
0 - 3    6
```

Analysis :

 Graph Representation:

- The graph is represented as an **adjacency matrix**, which is a 2D array where graph[u][v] contains the weight of the edge between vertices u and v. If there's no edge between u and v, the entry will be 0 (indicating no direct connection).

 Key Concepts:

- parent[]: An array that stores the parent of each vertex in the MST.
- key[]: An array that stores the minimum edge weight required to include each vertex in the MST.
- mstSet[]: A boolean array that indicates whether a vertex has been included in the MST or not.

 Algorithm Flow:

- **Initialization**:
    - The key[] array is initialized to a very large value (Integer.MAX_VALUE), except for the starting vertex (vertex 0), which is set to 0 to ensure it is picked first.
    - mstSet[] is initialized to false for all vertices (indicating that none of the vertices are included in the MST at the start).
    - The parent of the starting vertex is set to -1 as it does not have a parent.
- **Main Loop** (The algorithm runs for V-1 iterations):
    - In each iteration, the algorithm selects the vertex u that has the minimum key value among the vertices not yet included in the MST (minKey() function).
    - Then, it marks vertex u as included in the MST (mstSet[u] = true).
    - For each adjacent vertex v, if the edge u-v is smaller than the current key value of vertex v, the algorithm updates the key of vertex v and sets u as its parent.
- **Printing the MST**: After constructing the MST, the printMST() function is called to display the edges of the MST.

Summary of Time and Space Complexities

1. **Best Case Time Complexity**:
    - **O(V^2)**: Even in the best case, the time complexity remains **O(V^2)** because the algorithm still needs to find the minimum key value for each vertex and process all adjacent vertices.
2. **Average Case Time Complexity**:
    - **O(V^2)**: For a graph with arbitrary edge weights, the time complexity remains **O(V^2)** because the number of iterations in both the main loop and the minKey() function is proportional to V.
3. **Worst Case Time Complexity**:
    - **O(V^2)**: In the worst case (dense graphs), where every vertex is connected to every other vertex, the algorithm still runs in **O(V^2)** time.
4. **Space Complexity**:
    - **O(V^2)**: This is due to the adjacency matrix representation of the graph.
    - **O(V)**: Additional space is used by the key[], parent[], and mstSet[] arrays, but this is dominated by the adjacency matrix.

Kruskal's Algorithm

```
import java.util.Arrays;
import java.util.Comparator;
// Class to represent a graph edge
class Edge {
  int src, dest, weight;
  public Edge(int src, int dest, int weight) {
    this.src = src;
    this.dest = dest;
    this.weight = weight;
  }
}
// Disjoint-set (Union-Find) data structure
class DisjointSet {
  int[] parent, rank;
  public DisjointSet(int n) {
    parent = new int[n];
    rank = new int[n]
    for (int i = 0; i < n; i++) {
      parent[i] = i;
      rank[i] = 0;
    }
  }
  // Find the representative of the set containing x
  public int find(int x) {
    if (parent[x] != x) {
      parent[x] = find(parent[x]); // Path compression
    }
    return parent[x];
  }
  // Union of two sets
```

```
    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        // Union by rank
        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
}
public class KruskalAlgorithm {
    // Function to perform Kruskal's algorithm to find MST
    public static void kruskalMST(Edge[] edges, int V) {
        // Sort edges based on their weight
        Arrays.sort(edges, Comparator.comparingInt(e -> e.weight));
        // Create a Disjoint-set (Union-Find) data structure
        DisjointSet ds = new DisjointSet(V);
        System.out.println("Edges in the Minimum Spanning Tree:");

        // Traverse the sorted edges and add them to the MST if they don't form a cycle
        for (Edge edge : edges) {
            int x = ds.find(edge.src);
            int y = ds.find(edge.dest);
            // If adding this edge doesn't form a cycle
            if (x != y) {
```

```
            System.out.println(edge.src + " - " + edge.dest + " : " + edge.weight);
            ds.union(x, y);  // Union the sets
        }
      }
    }
    public static void main(String[] args) {
      int V = 4; // Number of vertices
      Edge[] edges = new Edge[] {
        new Edge(0, 1, 10),
        new Edge(0, 2, 6),
        new Edge(0, 3, 5),
        new Edge(1, 3, 15),
        new Edge(2, 3, 4)
      };
      kruskalMST(edges, V); // Run Kruskal's algorithm
    }
}
```

Output :

Edges in the Minimum Spanning Tree:

2 - 3 : 4

0 - 3 : 5

0 - 1 : 10

Analysis :

1. Edge Class:

The Edge class represents a graph edge with three attributes:

- src: the source vertex of the edge.
- dest: the destination vertex of the edge.
- weight: the weight of the edge.

The constructor initializes these values when an Edge object is created.

2. DisjointSet (Union-Find) Class:

This class is used for managing the connected components of the graph during the execution of Kruskal's algorithm.

- **Attributes**:
    - parent[]: An array where parent[i] holds the representative (or root) of the set containing vertex i.
    - rank[]: An array that holds the rank (or depth) of each tree. It is used to optimize the union operation by always attaching the smaller tree to the root of the larger tree (this is called **Union by Rank**).
- **Methods**:
    - find(x): Finds the representative of the set containing x. It uses **path compression**, which flattens the structure of the tree by making each node point directly to the root.
    - union(x, y): Merges the sets containing x and y. It uses **Union by Rank** to keep the tree structure flat, which helps in optimizing the algorithm.

3. Kruskal's Algorithm:

- **Sorting the edges**: The edges are sorted in non-decreasing order of their weights.
- **Union-Find**: We initialize a Union-Find data structure and use it to determine whether adding an edge would form a cycle. If not, the edge is added to the MST.
- **Cycle Detection**: The find method checks if the two vertices of the edge are in the same set. If they are, adding the edge would form a cycle, and the edge is skipped. If they are in different sets, the edge is added to the MST, and the sets are unified using the union method.

4. Execution in the main Method:

- The graph is represented as an array of Edge objects, each containing the source, destination, and weight of an edge.
- The kruskalMST method is called with the edges and the number of vertices (V = 4), and the MST is printed.

Summary of Time and Space Complexities

1. **Time Complexity**:
    - **Best Case**: O(ElogE)O(E \log E)O(ElogE)
    - **Average Case**: O(ElogE)O(E \log E)O(ElogE)
    - **Worst Case**: O(ElogE)O(E \log E)O(ElogE)
2. **Space Complexity**:
    - **Best Case**: O(E+V)O(E + V)O(E+V)
    - **Average Case**: O(E+V)O(E + V)O(E+V)
    - **Worst Case**: O(E+V)O(E + V)O(E+V)