**Title: Implementing Prim's Algorithm to Resolve Looping Issues in Network Switches.**

**Objective Statement:** The objective of this project is to learn and apply Prim's algorithm to solve the looping problem in network switches by constructing a Minimum Spanning Tree (MST). This will involve understanding MST concepts, implementing the algorithm in C++, and using data structures like priority queues and adjacency lists to efficiently manage and optimize network paths.

**Problem Statement:** In network environments, redundant paths between switches can cause looping issues, resulting in network inefficiency and potential broadcast storms. This project aims to address these issues by using Prim's algorithm to construct a Minimum Spanning Tree (MST). By connecting switches with the minimal number of edges necessary, the MST will eliminate redundant paths, ensuring a loop-free network topology that maximizes performance.

**CODE:**

#include <iostream>

#include <vector>

#include <queue>

#include <utility>

#include <climits>

using namespace std;

typedef pair<int, int> Edge; // (weight, vertex)

// Function to perform Prim's algorithm for MST

void primMST(int n, vector<vector<Edge>>& adj) {

vector<int> key(n, INT\_MAX); // Minimum edge weight to reach each node

vector<int> parent(n, -1); // MST structure to store parent of each node

vector<bool> inMST(n, false); // Track nodes included in MST

// Min-heap to select the edge with the minimum weight

priority\_queue<Edge, vector<Edge>, greater<Edge>> pq;

// Starting the MST from node 0

key[0] = 0;

pq.push({0, 0});

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

// Skip if the node is already included in the MST

if (inMST[u]) continue;

inMST[u] = true;

// Traverse all adjacent vertices of the current node

for (auto &[weight, v] : adj[u]) {

// If v is not in MST and the weight of (u, v) is smaller than key[v]

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

key[v] = weight;

parent[v] = u;

pq.push({key[v], v});

}

}

}

// Print the edges of the MST and the total cost

int total\_cost = 0;

cout << "Edges in the MST:" << endl;

for (int v = 1; v < n; ++v) { // Skip the root node (0)

if (parent[v] != -1 && key[v] != INT\_MAX) { // Ensure it's a valid edge

cout << parent[v] << " - " << v << " : " << key[v] << endl;

total\_cost += key[v];

} else {

cout << "Node " << v << " is disconnected and not included in the MST." << endl;

}

}

cout << "Total cost of MST: " << total\_cost << endl;

}

int main() {

int n = 5; // Number of nodes (for example, representing switches in a network)

// Graph represented as an adjacency list

// Each node has a list of edges (weight, destination node)

vector<vector<Edge>> adj(n);

// Adding edges to the adjacency list

// The graph is undirected, so each edge is added in both directions

adj[0].push\_back({10, 1});

adj[1].push\_back({10, 0});

adj[0].push\_back({6, 2});

adj[2].push\_back({6, 0});

adj[0].push\_back({5, 3});

adj[3].push\_back({5, 0});

adj[1].push\_back({15, 3});

adj[3].push\_back({15, 1});

adj[2].push\_back({4, 3});

adj[3].push\_back({4, 2});

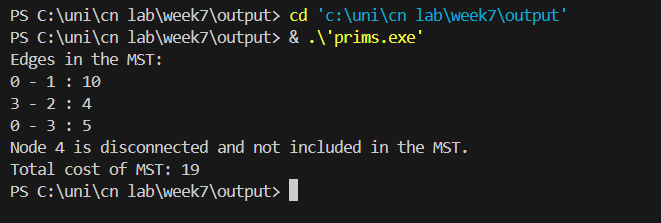
// Run Prim's algorithm to find and print the MST

primMST(n, adj);

return 0;

}

**OUTPUT:**

****

**Problem Faced:** One challenge encountered was handling disconnected components within the graph, as Prim’s algorithm only constructs the MST for a single connected component. This required ensuring the input graph was fully connected to avoid isolated nodes. Additionally, debugging issues with priority queue operations required close attention to ensure that the minimum edge was always selected.

**Conclusion:** Through this project, I developed a solid understanding of Prim's algorithm and its application to networking issues such as loop prevention in switches. I enhanced my C++ programming skills, especially in the use of priority queues and adjacency lists. The project also helped me improve problem-solving skills by handling graph edge cases and optimizing code for efficient MST construction.