Maximum Spanning Tree using Prim's Algorithm

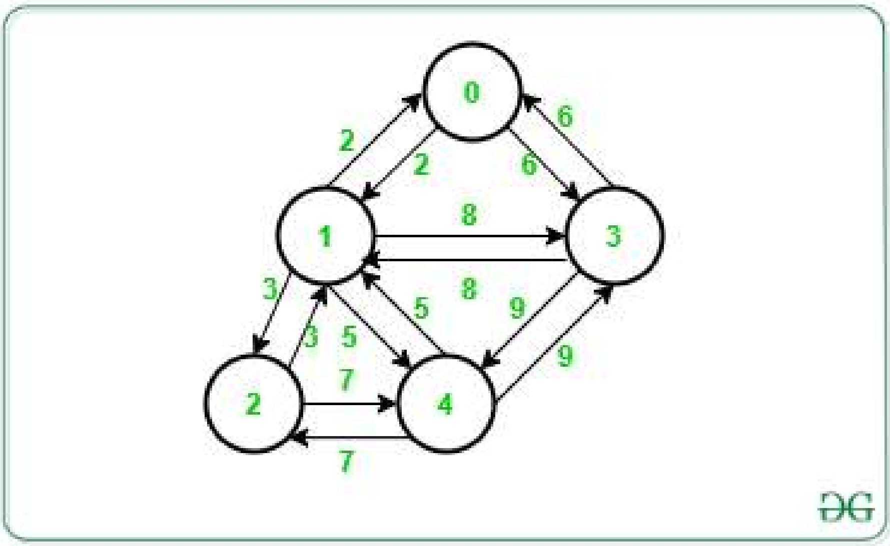
mishrapriyanshu557

Given [undirected weighted graph] **G**, the task is to find the **Maximum Spanning Tree** of the Graph using [Prim's Algorithm]

> *Prims algorithm is a Greedy algorithm which can be used to find the Minimum Spanning Tree (MST) as well as the* ***Maximum Spanning Tree*** *of a Graph.*

Examples:

> ***Input:*** *graph[V][V] = {{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:***
> *The total weight of the Maximum Spanning tree is 30.*
> *Edges Weight*
> *3 – 1 8*
> *4 – 2 7*
> *0 – 3 6*
> *3 – 4 9*
> ***Explanation:***
> *Choosing other edges won't result in maximum spanning tree.*



Maximum Spanning Tree:

Given an undirected weighted graph, a maximum spanning tree is a spanning tree having maximum weight. It can be easily computed using Prim's algorithm. The goal here is to find the spanning tree with the maximum weight out of all possible spanning trees.

Prim's Algorithm:

Prim's algorithm is a greedy algorithm, which works on the idea that a spanning tree must have all its vertices connected. The algorithm works by building the tree one vertex at a time, from an arbitrary starting vertex, and adding the most expensive possible connection from the tree to another vertex, which will give us the **Maximum Spanning Tree (MST)**.

Follow the steps below to solve the problem:

- Initialize a visited array of boolean datatype, to keep track of vertices visited so far. Initialize all the values with **false**.
- Initialize an array **weights[]**, representing the maximum weight to connect that vertex. Initialize all the values with some minimum value.
- Initialize an array **parent[]**, to keep track of the **maximum spanning tree**.
- Assign some large value, as the weight of the first vertex and parent as **-1**, so that it is picked first and has no parent.
- From all the unvisited vertices, pick a vertex **v** having a maximum weight and mark it as visited.
- Update the weights of all the unvisited adjacent vertices of **v**. To update the weights, iterate through all the unvisited neighbors of **v**. For every adjacent vertex **x**, if the

weight of the edge between **v** and **x** is greater than the previous value of **v**, update the value of **v** with that weight.

Below is the implementation of the above algorithm:

C++

```
// C++ program for the above algorithm

#include <bits/stdc++.h>
using namespace std;
#define V 5

// Function to find index of max-weight
// vertex from set of unvisited vertices
int findMaxVertex(bool visited[], int weights[])
{

    // Stores the index of max-weight vertex
    // from set of unvisited vertices
    int index = -1;

    // Stores the maximum weight from
    // the set of unvisited vertices
    int maxW = INT_MIN;

    // Iterate over all possible
    // nodes of a graph
    for (int i = 0; i < V; i++) {

        // If the current node is unvisited
        // and weight of current vertex is
        // greater than maxW
        if (visited[i] == false
            && weights[i] > maxW) {

            // Update maxW
            maxW = weights[i];

            // Update index
            index = i;
        }
    }
    return index;
}

// Utility function to find the maximum
// spanning tree of graph
void printMaximumSpanningTree(int graph[V][V],
                              int parent[])
{

    // Stores total weight of
    // maximum spanning tree
```

```
    // of a graph
    int MST = 0;

    // Iterate over all possible nodes
    // of a graph
    for (int i = 1; i < V; i++) {

        // Update MST
        MST += graph[i][parent[i]];
    }

    cout << "Weight of the maximum Spanning-tree "
         << MST << '\n'
         << '\n';

    cout << "Edges \tWeight\n";

    // Print the Edges and weight of
    // maximum spanning tree of a graph
    for (int i = 1; i < V; i++) {
        cout << parent[i] << " - " << i << " \t"
             << graph[i][parent[i]] << " \n";
    }
}

// Function to find the maximum spanning tree
void maximumSpanningTree(int graph[V][V])
{

    // visited[i]:Check if vertex i
    // is visited or not
    bool visited[V];

    // weights[i]: Stores maximum weight of
    // graph to connect an edge with i
    int weights[V];

    // parent[i]: Stores the parent node
    // of vertex i
    int parent[V];

    // Initialize weights as -INFINITE,
    // and visited of a node as false
    for (int i = 0; i < V; i++) {
        visited[i] = false;
        weights[i] = INT_MIN;
    }

    // Include 1st vertex in
    // maximum spanning tree
    weights[0] = INT_MAX;
    parent[0] = -1;

    // Search for other (V-1) vertices
    // and build a tree
    for (int i = 0; i < V - 1; i++) {
```

```
            // Stores index of max-weight vertex
            // from a set of unvisited vertex
            int maxVertexIndex
                = findMaxVertex(visited, weights);

            // Mark that vertex as visited
            visited[maxVertexIndex] = true;

            // Update adjacent vertices of
            // the current visited vertex
            for (int j = 0; j < V; j++) {

                // If there is an edge between j
                // and current visited vertex and
                // also j is unvisited vertex
                if (graph[j][maxVertexIndex] != 0
                    && visited[j] == false) {

                    // If graph[v][x] is
                    // greater than weight[v]
                    if (graph[j][maxVertexIndex] > weights[j]) {

                        // Update weights[j]
                        weights[j] = graph[j][maxVertexIndex];

                        // Update parent[j]
                        parent[j] = maxVertexIndex;
                    }
                }
            }
        }

        // Print maximum spanning tree
        printMaximumSpanningTree(graph, parent);
    }

    // Driver Code
    int main()
    {

        // Given graph
        int graph[V][V] = { { 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 } };

        // Function call
        maximumSpanningTree(graph);

        return 0;
    }
```

Java