Green University of Bangladesh
Department of Computer Science and Engineering (CSE)
Faculty of Sciences and Engineering
Semester: (Spring, Year: 2023), B.Sc. in CSE (Day)
Lab Report: 02
Course Title: Algorithm Lab
Course Code: CSE 206 Section: DA

Lab Report Name: Detect Cycle using BFS, topological sort, Find level use BFS, Find depth of Tree use BFS, Kruskal and Prim's find the total number of minimum spanning tree.

Student Details

Name	ID
Pankaj Mahanta	213902002

Lab Date : **08- 03 - 2023**
Submission Date : **03 – 17 - 2023**
Course Teacher's Name : Md. Sultanul Islam Ovi

Lab Report Status	
Marks: ………………………………	**Signature: ....**
Comments:	**Date: ....**

Task-1: Write a program to detect the cycle in a graph using BFS.

The given problem implements the topological sorting algorithm to detect cycles in a directed graph. Here's a step-by-step explanation of the code:

Then, it initializes an adjacency list adj with a size of N to store the directed edges between vertices.It also initializes an InDegree vector of size N with all elements initialized to 0, to store the indegree of each vertex.

The addEdges method takes two arguments u and v representing the source and destination vertices of the edge respectively, and updates the adj and InDegree vectors accordingly. It adds the destination vertex to the adjacency list of the source vertex and increments the indegree of the destination vertex.

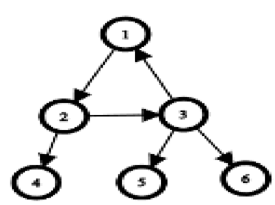
The CheckCycle method takes an integer n as an argument, representing the total number of vertices in the graph. It first creates an empty queue q.It pushes all the vertices that have an indegree of 0 to the queue.It then enters into a loop where it dequeues a vertex from the queue, increments a counter count, and prints the vertex value.

For each adjacent vertex of the dequeued vertex, the method decreases its indegree by 1.If the indegree of any adjacent vertex becomes 0, it is pushed to the queue.The loop continues until the queue is empty.

If count is equal to n, it means all vertices have been traversed, and there is no cycle in the graph. Thus, the method returns false. Otherwise, the method returns true, indicating the presence of a cycle.

Overall, the code performs the topological sorting algorithm by keeping track of the indegree of each vertex and using a queue to traverse the graph in a topological order. If a cycle is detected, it returns true, otherwise false.

Graph:



Code:

Lab_Report_2 > MyCode > tast_1_Detect_Cycle_Bfs.c++ > InDegree

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 999;
// initialize adjacency list
vector<int> adj[N];
// initialize indegree 0
vector<int> InDegree(N, 0);
// add edges method
void addEdges(int u, int v)
{
    adj[u].push_back(v);
    InDegree[v]++;
}

// topological method
bool CheckCycle(int n)
{
    // create a queue
    queue<int> q;
    for (int i = 0; i < n; i++)
    {
        if (InDegree[i] == 0)
        {
            q.push(i);
        }
    }
    // this process is perform when the queue is empty
    // just initialize count variable
    int count = 0;
    while (!q.empty())
    {
        int node = q.front();
        q.pop();
        count++;
       // cout << node << " ";
        // jara node ar sathe connected tader degree 1 kore komay dibo
        for (auto it : adj[node])
        {
            InDegree[it]--;
            if (InDegree[it] == 0)
            { // again check oi element ar moddhe kader indgree 0 tader ke
                // push it
                q.push(it);
            }
        }
    }
    if (count == n)
        return false;
    return true;
}
int main()
{
    // input vertex and edge
    int n, m;
    cin >> n >> m;
```

```cpp
}
int main()
{
    // input vertex and edge
    int n, m;
    cin >> n >> m;
    // input edges
    for (int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        addEdges(u, v);
    }
    // work the main funda topological sort
    bool isCycle;
    isCycle = CheckCycle(n);

    if (isCycle)
    {
        cout << "\nCycle Detect!!\n";
    }
    else
    {
        cout << "\nNo Cycle\n";
    }

    return 0;
}
```

Output:

```
PS C:\Users\Admin\Desktop\Code\C_C++\C++> cd "c:\Users\Admin\De
t_Cycle_Bfs } ; if ($?) { .\tast_1_Detect_Cycle_Bfs }
6 6
1 2
2 3
3 1
3 5
3 6
2 4

Cycle Detect!!
PS C:\Users\Admin\Desktop\Code\C_C++\C++\Lab_Report_2\MyCode>
```

Task-2: Write a program to find the level of each node using BFS

This problem aims to find the level of each node in an undirected graph. The level of a node is defined as the minimum number of edges required to reach that node from a given source node. In this code, the source node is assumed to be node 1.

A vector of vectors is used to represent the graph. The vector adj[] stores the adjacency list of each node. A vector vis[] is used to keep track of visited nodes during BFS traversal. A vector level[] is used to store the level of each node.

addEdge() function takes two nodes u and v as input and adds an undirected edge between them by adding v to the adjacency list of u and vice versa.

printAdj() function takes the number of nodes n as input and prints the adjacency list of each node.

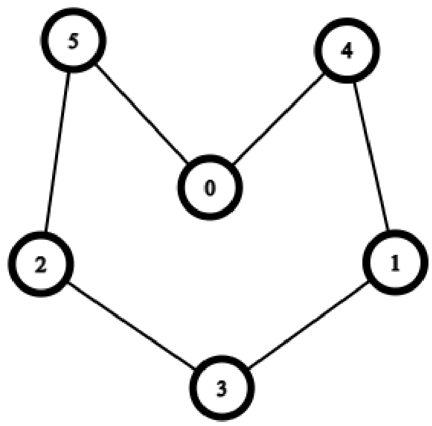
bfs() function takes the source node as input and performs a Breadth First Search (BFS) traversal starting from the source node. A queue of pairs is used to keep track of nodes and their corresponding levels during traversal. Initially, the source node is pushed into the queue with level 0 and marked as visited.While the queue is not empty, a node is popped from the front of the queue and its level is updated in the level[] vector.

For each unvisited neighbor of the current node, its level is updated as level[current_node] + 1, and the neighbor is pushed into the queue with its level.

Finally, it calls the bfs() function with the source node as input and prints the level of each node using the level[] vector.

Overall, this problem performs BFS traversal of the given graph starting from a source node and calculates the level of each node by counting the minimum number of edges required to reach each node from the source node.

Graph:



Code:

Lab_Report_2 > MyCode > task_2_level_Bfs.c++ > bfs(int)

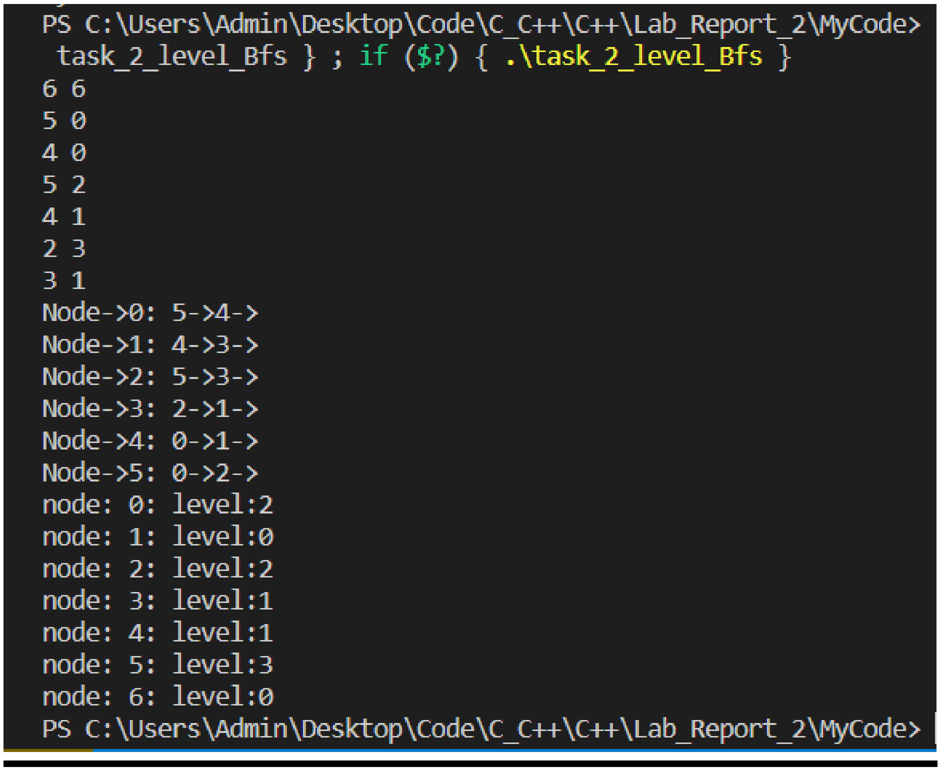
```cpp
// find the level
#include <bits/stdc++.h>
using namespace std;
const int N = 999;
vector<int> adj[N];
vector<int> vis(N);
vector<int> level(N);
void addEdge(int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}
void printAdj(int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << "Node->" << i << ": ";
        for (int it : adj[i])
        {
            cout << it << "->";
        }
        cout << "\n";
    }
}
void bfs(int source)
{ // create pair first position is node and second position is level
    queue<pair<int, int>> q;
    // node,level
    q.push({source, 0});
    // starting node is visited;
    vis[source] = 1;
    while(!q.empty()){
        int node=q.front().first;
        int l=q.front().second;
        q.pop();
        level[node]=l;
        for(int it: adj[node]){
            if(vis[it]==0){
                q.push({it,l+1});
                vis[it]=1;
            }
        }
    }
}

int main()
{
```

```cpp
int main()
{
    // add node and edge
    int n, m;
    cin >> n >> m;
    // add every edges
    for (int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        addEdge(u, v);
    }
    // present adjacency list
    printAdj(n);
    bfs(1);
  // print the level for every node

  for(int i=0;i<=n;i++){
    cout<<"node: "<<i<<": "<<"level:"<<level[i]<<endl;
  }
    return 0;
}
```

<u>**Output:**</u>



```
PS C:\Users\Admin\Desktop\Code\C_C++\C++\Lab_Report_2\MyCode>
 task_2_level_Bfs } ; if ($?) { .\task_2_level_Bfs }
6 6
5 0
4 0
5 2
4 1
2 3
3 1
Node->0: 5->4->
Node->1: 4->3->
Node->2: 5->3->
Node->3: 2->1->
Node->4: 0->1->
Node->5: 0->2->
node: 0: level:2
node: 1: level:0
node: 2: level:2
node: 3: level:1
node: 4: level:1
node: 5: level:3
node: 6: level:0
PS C:\Users\Admin\Desktop\Code\C_C++\C++\Lab_Report_2\MyCode>
```

Task-3: Write a program to perform topological sort using BFS.

This problem performs topological sorting on a directed acyclic graph (DAG). Topological sorting is a way of ordering the vertices of a graph such that for every directed edge (u, v), vertex u comes before vertex v in the ordering.

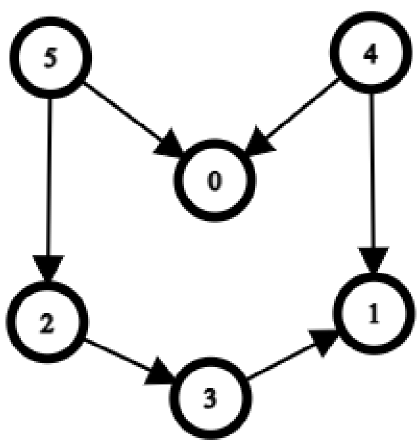
The code uses an adjacency list to represent the graph, where each index of the vector represents a vertex, and the values stored in the vector are the adjacent vertices. The InDegree vector keeps track of the number of incoming edges for each vertex.

The addEdges method adds an edge to the graph by adding the destination vertex to the adjacency list of the source vertex and incrementing the indegree of the destination vertex.

The topological method performs the topological sort. It starts by pushing all vertices with an indegree of 0 to a queue. Then, it processes the vertices in the queue one by one, printing them out and reducing the indegree of their adjacent vertices by 1. If the indegree of an adjacent vertex becomes 0, it is added to the queue. This process continues until the queue becomes empty.

In summary, topological sorting performs on a DAG represented by an adjacency list and indegree vector.

Graph:



Code:

```cpp

#include <bits/stdc++.h>
using namespace std;
const int N = 999;
// initialize adjacency list
vector<int> adj[N];
// initialize indegree 0
vector<int> InDegree(N, 0);

// add edges method
void addEdges(int u, int v)
{
    adj[u].push_back(v);
    InDegree[v]++;
}

// topological method
void topological(int n)
{
    // create a queue
    queue<int> q;
    // for(int i=0;i<n;i++){
    //     for(auto in:adj[i]){
    //         InDegree[in]++;
    //     }
    // }
    // push all the node which indegree is 0
    for(int i=0;i<n;i++){
        if(InDegree[i]==0){
            q.push(i);
        }
    }
    // this process is perform when the queue is empty
    while(!q.empty()){
        int node=q.front();
        q.pop();
        cout<<node<<" ";
        // jara node ar sathe connected tader degree 1 kore komay dibo
        for(auto it:adj[node]){
            InDegree[it]--;
            if(InDegree[it]==0){ //again check oi element ar moddhe kader indgree 0 tader ke
            //push it
                q.push(it);
            }
        }
```

```
}
int main()
{
    // input vertex and edge
    int n, m;
    cin >> n >> m;
    // input edges
    for (int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        addEdges(u, v);
    }
    // work the main funda topological sort
    topological(n);

    return 0;
}
```

Output:

```
6 6
5 0
4 0
5 2
4 1
2 3
3 1
4 5 0 2 3 1
PS C:\Users\Admin\Desktop\Code\C_C++\C++\Lab_Report_2\MyCode>
```

Task-4: Write a program to find the minimum depth of a binary tree.

This problem is used to find the minimum depth of a binary tree.

First, a binary tree is defined using the Node structure. The structure contains three fields: data, left and right. The data field stores the value of the node, the left field points to the left child node and the right field points to the right child node.

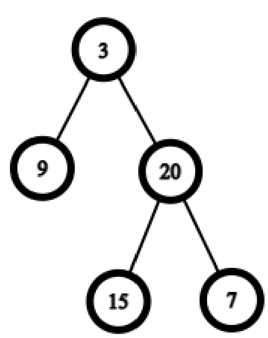
The minDepth() function takes the root node of the binary tree as input and returns the minimum depth of the tree. If the root node is NULL, the function returns 0. A queue is created using the queue this queue type is Node pointer. The queue stores nodes of the binary tree.

The root node is pushed into the queue. The variable depth is initialized to 1. It will be used to keep track of the current depth of the tree. A while loop is executed until the queue becomes empty. Inside the while loop, the size of the queue is stored in the variable size.

Another while loop is executed until size becomes 0. Inside this while loop, the front node of the queue is removed and stored in the variable root. If the left and right child nodes of the root node are both NULL, it means that the root node is a leaf node. Therefore, the function returns the current depth.

If the left child node of the root node is not NULL, it is pushed into the queue. If the right child node of the root node is not NULL, it is pushed into the queue. After processing all nodes at the current level, the depth variable is incremented. After the while loop completes, the function returns the final depth of the tree.

Graph:



Code:

```cpp
// minimum depth
#include <bits/stdc++.h>
using namespace std;
struct Node
{
    int data;
    Node *left, *right;
    Node(int value){
        value=data;
        left=right=NULL;
    }
};
int minDepth(Node *root){
    if(root==NULL)
    return 0;
    // create queue which type is node
    queue<Node *>q;
    // push root node first
    q.push(root);
    // initialize depth is 1
    int depth=1;
    while(!q.empty()){
        int size=q.size();
      //  cout<<size<<endl;
        while(size-->0){
            Node *root=q.front();
            q.pop();
         //   cout<<"root: "<<root->data<<endl;
            if(root->left==NULL && root->right==NULL)
              // return depth find for leaf node
              return depth;
              // check left node element is exist if exist then push it
            if(root->left!=NULL)
            q.push(root->left);
            // check right node element is exist if exist then push it
            if(root->right!=NULL)
            q.push(root->right);
        }
        // increment depth after processing all nodes at the current level
        depth++;
    }
    // return final depth
    return depth;
 }
int main()
{   Node *root;
     root=new Node(3);
     root->left=new Node(9);
```

```cpp
            depth++;
        }
        // return final depth
        return depth;
    }
int main()
{   Node *root;
     root=new Node(3);
     root->left=new Node(9);
     root->right=new Node(20);
     root->right->left=new Node(15);
     root->right->right=new Node(9);

    cout<<minDepth(root)<<endl;
    return 0;
}
/*
```

Output:

```
PS C:\Users\Admin\Desktop\Code\C_C++\C++\Lab_Report_2\MyCode> c
 task_4_MiniDepth } ; if ($?) { .\task_4_MiniDepth }
Minimum depth:
2
PS C:\Users\Admin\Desktop\Code\C_C++\C++\Lab_Report_2\MyCode>
```

Task-5: Write a program to find the number of distinct minimum spanning trees for a given weighted graph using Kruskal algorithm.

The code defines a struct Edge to represent an edge in the graph. The struct has three integer fields: u and v represent the endpoints of the edge, and w represents the weight of the edge.

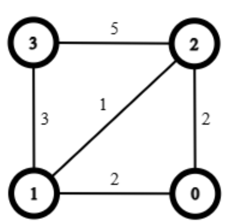
The code defines a function kruskal that takes a vector of Edge objects and the number of vertices in the graph, and returns a vector of Edge objects representing the MST of the graph. The function first initializes a vector mst to store the edges in the MST, and a vector parent to store the parent of each vertex in the MST. Initially, each vertex is its own parent. The function then sorts the input edges by weight using the sort function and a comparison function cmp. The function then iterates over the sorted edges, adding each edge to the MST if its endpoints have different parents. If an edge is added to the MST, the function updates the parent of all vertices in the subtree rooted at the endpoint with the old parent to the new parent.

The code defines a function count_msts that takes a vector of Edge objects and the number of vertices in the graph, and returns the number of distinct MSTs of the graph. The function first finds the MST of the input graph using the kruskal function. The function then initializes a variable count to 0, and iterates over each edge in the MST. For each edge, the function creates a copy of the input edges with the current edge removed, and finds the MST of the new graph using the kruskal function. The function then computes the weight of the new MST, and if it is equal to the weight of the original MST, the function increments count. Finally, the function returns count.

In the main function, the code defines a small example graph with 4 vertices and 5 edges, and calls the count_msts function to count the number of distinct MSTs of the graph. The function prints the result to the console.

In summary, this code demonstrates a technique for counting the number of distinct minimum spanning trees of a graph by using Kruskal's algorithm to find the MST and then iterating over all possible MSTs by removing each edge from the MST and finding the new MST.

Graph:



Code:

Lab_Report_2 > Others > kkn.c++ > ...

```cpp
#include <bits/stdc++.h>
using namespace std;
struct Edge {
    int u, v, w;
    Edge(int uu, int vv, int ww) : u(uu), v(vv), w(ww) {}
};

bool cmp(Edge& a, Edge& b) {
    return a.w < b.w;
}

vector<Edge> kruskal(vector<Edge>& edges, int n) {
    vector<Edge> mst;
    vector<int> parent(n);
    for (int i = 0; i < n; i++) {
        parent[i] = i;
    }
    sort(edges.begin(), edges.end(), cmp);
    for (Edge e : edges) {
        int pu = parent[e.u];
        int pv = parent[e.v];
        if (pu != pv) {
            mst.push_back(e);
            for (int i = 0; i < n; i++) {
                if (parent[i] == pv) {
                    parent[i] = pu;
                }
            }
        }
    }
    return mst;
}

int count_msts(vector<Edge>& edges, int n) {
    int mst_weight = 0;
    vector<Edge> mst = kruskal(edges, n);
    for (Edge e : mst) {
        mst_weight += e.w;
    }
    int count = 0;
    for (int i = 0; i < mst.size(); i++) {
        vector<Edge> new_edges = edges;
        new_edges.erase(new_edges.begin() + i);
        vector<Edge> new_mst = kruskal(new_edges, n);
        int new_weight = 0;
        for (Edge e : new_mst) {
            new_weight += e.w;
        }
```

```cpp
            {
                int u = parent[i];
                int v = i;
                int weight = 0;
                for (Edge &e : g.adj[u])
                {
                    if (e.v == v)
                    {
                        weight = e.weight;
                        break;
                    }
                }
                edges.push_back(Edge(v, weight));
            }

            sort(edges.begin(), edges.end(), [](Edge &a, Edge &b)
                 { return a.weight < b.weight; });

            vector<int> rank(g.V, 0);
            vector<int> parent(g.V, -1);
            for (int i = 0; i < g.V; i++)
            {
                parent[i] = i;
            }
            for (int i = 0; i < edges.size(); i++)
            {
                int u = edges[i].v;
                int v = parent[u];
                while (u != v)
                {
                    parent[u] = v;
                    u = parent[u];
                    v = parent[v];
                }
                if (parent[edges[i].v] != parent[edges[i - 1].v] || i == 0)
                {
                    count++;
                }
            }
        }
        return count;
    }

    int main()
    {
        int V = 4;
        Graph g;
        g.V = V;
        g.adj = vector<vector<Edge>>(V);

        g.adj[0].push_back(Edge(1, 2));
        g.adj[1].push_back(Edge(0, 2));

        g.adj[0].push_back(Edge(2, 3));
        g.adj[2].push_back(Edge(0, 3));

        g.adj[1].push_back(Edge(2, 1));
        g.adj[2].push_back(Edge(1, 1));

        g.adj[1].push_back(Edge(3, 4));
        g.adj[3].push_back(Edge(1, 4));

        g.adj[2].push_back(Edge(3, 5));
        g.adj[3].push_back(Edge(2, 5));

        int msts = count_msts(g);
        cout << "Number of distinct minimum spanning trees: " << msts << endl;

        return 0;
    }
```

```cpp
        int new_weight = 0;
        for (Edge e : new_mst) {
            new_weight += e.w;
        }
        if (new_weight == mst_weight) {
            count++;
        }
    }
    return count;
}

int main() {
    int n = 4;
    vector<Edge> edges;
    edges.push_back(Edge(0, 1, 2));
    edges.push_back(Edge(0, 2, 2));
    edges.push_back(Edge(1, 2, 1));
    edges.push_back(Edge(1, 3, 3));
    edges.push_back(Edge(2, 3, 5));
    int count = count_msts(edges, n);
    cout << "Number of distinct minimum spanning trees: " << count << endl;
    return 0;
}
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    COMMENTS

PS C:\Users\Admin\Desktop\Code\C_C++\C++> cd "c:\Users\Admin\D
Number of distinct minimum spanning trees: 2
PS C:\Users\Admin\Desktop\Code\C_C++\C++\Lab_Report_2\Others>
```

Task-6: Write a program to find the number of distinct minimum spanning trees for a given weighted graph using the Prim algorithm.

The first step is to define two data structures: Edge and Graph. Edge represents an edge in the graph and contains the destination vertex v and the weight of the edge. Graph contains the number of vertices V and an adjacency list representation of the graph. The count_msts function takes a Graph object g as input and returns an integer representing the number of distinct minimum spanning trees in the graph.

The function initializes a variable count to 0 to count the number of MSTs. It initializes three vectors: parent, dist, and visited, each with size V. parent stores the parent of each vertex in the MST dist stores the distance from the source vertex to each vertex visited is used to mark visited vertices.

It also initializes a priority queue pq to store the vertices to be processed. It is a min heap where the top element has the minimum distance. The function loops over all vertices in the graph and performs the following steps for each vertex:

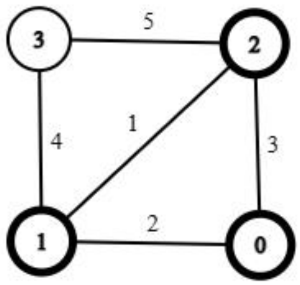
It assigns the distance from the source vertex to all vertices to be INT_MAX and marks all vertices as unvisited. It pushes the source vertex into the priority queue with distance 0. It loops while the priority queue is not empty:

It pops the vertex with the minimum distance from the priority queue. If the vertex is already visited, it continues to the next iteration of the loop. Otherwise, it marks the vertex as visited and updates the distance of all its neighbors in the priority queue if the new distance is smaller than the current distance.

After the loop, parent vector contains the parent of each vertex in the MST. For each MST, the function constructs a vector of Edge objects representing the edges in the MST. The function sorts the vector of edges in non-decreasing order of weight. It initializes two vectors rank and parent, each with size V rank stores the rank of each vertex in the union-find data structure.

parent stores the parent of each vertex in the union-find data structure. It sets the parent of each vertex to itself initially. The function loops over the sorted edges and performs the following steps for each edge: It finds the parent of the source vertex u and the parent of the destination vertex v. It performs union-by-rank by making the parent of the vertex with smaller rank the parent of the other vertex. If the parent of the destination vertex v is different from the parent of the source vertex u, it increments the count variable. The function returns the final value of the count variable.

Graph:



Code:

Lab_Report_2 > Others > ppn.c++ > ...

```cpp
#include <bits/stdc++.h>
using namespace std;
struct Edge
{
    int v, weight;
    Edge(int v, int w) : v(v), weight(w) {}
};

struct Graph
{
    int V;
    vector<vector<Edge>> adj;
};
int count_msts(Graph &g)
{
    int count = 0;
    vector<int> parent(g.V, -1);
    vector<int> dist(g.V, INT_MAX);
    vector<bool> visited(g.V, false);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    for (int s = 0; s < g.V; s++)
    {
        dist.assign(g.V, INT_MAX);
        visited.assign(g.V, false);
        pq.push(make_pair(0, s));
        dist[s] = 0;

        while (!pq.empty())
        {
            int u = pq.top().second;
            pq.pop();
            if (visited[u])
            {
                continue;
            }
            visited[u] = true;
            for (Edge &e : g.adj[u])
            {
                int v = e.v;
                int weight = e.weight;
                if (!visited[v] && weight < dist[v])
                {
                    dist[v] = weight;
                    parent[v] = u;
                    pq.push(make_pair(dist[v], v));
                }
            }
        }

        vector<Edge> edges;
        for (int i = 1; i < g.V; i++)
        {
            int u = parent[i];
            int v = i;
```

```cpp
            int u = parent[i];
            int v = i;
            int weight = 0;
            for (Edge &e : g.adj[u])
            {
                if (e.v == v)
                {
                    weight = e.weight;
                    break;
                }
            }
            edges.push_back(Edge(v, weight));
        }

        sort(edges.begin(), edges.end(), [](Edge &a, Edge &b)
             { return a.weight < b.weight; });

        vector<int> rank(g.V, 0);
        vector<int> parent(g.V, -1);
        for (int i = 0; i < g.V; i++)
        {
            parent[i] = i;
        }
        for (int i = 0; i < edges.size(); i++)
        {
            int u = edges[i].v;
            int v = parent[u];
            while (u != v)
            {
                parent[u] = v;
                u = parent[u];
                v = parent[v];
            }
            if (parent[edges[i].v] != parent[edges[i - 1].v] || i == 0)
            {
                count++;
            }
        }
    }
    return count;
}

int main()
{
    int V = 4;
    Graph g;
    g.V = V;
    g.adj = vector<vector<Edge>>(V);

    g.adj[0].push_back(Edge(1, 2));
    g.adj[1].push_back(Edge(0, 2));

    g.adj[0].push_back(Edge(2, 3));
    g.adj[2].push_back(Edge(0, 3));

    g.adj[1].push_back(Edge(2, 1));
    g.adj[2].push_back(Edge(1, 1));

    g.adj[1].push_back(Edge(3, 4));
    g.adj[3].push_back(Edge(1, 4));

    g.adj[2].push_back(Edge(3, 5));
    g.adj[3].push_back(Edge(2, 5));

    int msts = count_msts(g);
    cout << "Number of distinct minimum spanning trees: " << msts << endl;

    return 0;
}
```

Output:

Output

```
/tmp/HWNf2VvhPY.o
Number of distinct minimum spanning trees: 12
```