|  |  |
| --- | --- |
|  | **DEPARTMENT OF COMPUTER ENGINEERING** |

**Experiment No. 06**

|  |  |
| --- | --- |
| Semester | B.E. Semester VII – Computer Engineering |
| Subject | Big Data Analysis |
| Subject Professor In-charge | Prof. Pankaj Vanvari |
| Lab Professor In-charge | Dr. Umesh Kulkarni |
| Academic Year | 2024-25 |

|  |  |
| --- | --- |
| Student Name | Deep Salunkhe |
| Roll Number | 21102A0014 |

**Title:** Community detection algorithm (Girvan Newman Algorithm)

#include <iostream>

#include <vector>

#include <queue>

#include <map>

#include <set>

#include <algorithm>

#include <limits>

using namespace std;

typedef pair<int, int> Edge;

*// Function to perform BFS and calculate edge betweenness*

map<Edge, double> calculateEdgeBetweenness(vector<vector<int>>& graph, int V) {

    map<Edge, double> edgeBetweenness;

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

*// BFS variables*

        vector<int> dist(V, -1), numShortestPaths(V, 0), parent(V, -1);

        vector<double> dependency(V, 0.0);

        queue<int> q;

        vector<vector<int>> predecessors(V);

*// Start BFS*

        q.push(src);

        dist[src] = 0;

        numShortestPaths[src] = 1;

        vector<int> order; *// To maintain the BFS order*

        while (!q.empty()) {

            int node = q.front();

            q.pop();

            order.push\_back(node);

*// Traverse neighbors*

            for (int neighbor : graph[node]) {

*// Node found for the first time*

                if (dist[neighbor] == -1) {

                    dist[neighbor] = dist[node] + 1;

                    q.push(neighbor);

                }

*// Count shortest paths*

                if (dist[neighbor] == dist[node] + 1) {

                    numShortestPaths[neighbor] += numShortestPaths[node];

                    predecessors[neighbor].push\_back(node);

                }

            }

        }

*// Back-propagation of dependencies*

        for (int i = order.size() - 1; i >= 0; --i) {

            int node = order[i];

            for (int pred : predecessors[node]) {

                double partialDependency = ((double)numShortestPaths[pred] / numShortestPaths[node]) \* (1 + dependency[node]);

                dependency[pred] += partialDependency;

*// Track edge betweenness*

                Edge e = minmax(pred, node);

                edgeBetweenness[e] += partialDependency;

            }

        }

    }

*// Divide by 2 because each edge is counted twice*

    for (auto& eb : edgeBetweenness) {

        eb.second /= 2.0;

    }

    return edgeBetweenness;

}

*// DFS to find connected components (communities)*

void dfs(int node, vector<vector<int>>& graph, vector<bool>& visited, vector<int>& component) {

    visited[node] = true;

    component.push\_back(node);

    for (int neighbor : graph[node]) {

        if (!visited[neighbor]) {

            dfs(neighbor, graph, visited, component);

        }

    }

}

*// Function to find communities*

vector<vector<int>> findCommunities(vector<vector<int>>& graph, int V) {

    vector<vector<int>> communities;

    vector<bool> visited(V, false);

    for (int i = 0; i < V; ++i) {

        if (!visited[i]) {

            vector<int> component;

            dfs(i, graph, visited, component);

            communities.push\_back(component);

        }

    }

    return communities;

}

*// Remove edge from the graph*

void removeEdge(vector<vector<int>>& graph, int u, int v) {

    graph[u].erase(remove(graph[u].begin(), graph[u].end(), v), graph[u].end());

    graph[v].erase(remove(graph[v].begin(), graph[v].end(), u), graph[v].end());

}

*// Main function implementing Girvan-Newman*

void girvanNewman(vector<vector<int>>& graph, int V) {

    while (true) {

*// Calculate betweenness centrality*

        map<Edge, double> edgeBetweenness = calculateEdgeBetweenness(graph, V);

*// Find the edge with the maximum betweenness*

        Edge maxEdge = {-1, -1};

        double maxBetweenness = -numeric\_limits<double>::infinity();

        for (auto& eb : edgeBetweenness) {

            if (eb.second > maxBetweenness) {

                maxBetweenness = eb.second;

                maxEdge = eb.first;

            }

        }

*// No more edges to remove*

        if (maxEdge.first == -1) {

            break;

        }

*// Remove the edge*

        removeEdge(graph, maxEdge.first, maxEdge.second);

        cout << "Removed edge: (" << maxEdge.first << ", " << maxEdge.second << ")" << endl;

*// Find communities*

        vector<vector<int>> communities = findCommunities(graph, V);

*// Output the communities at this stage*

        cout << "Communities after removing edge (" << maxEdge.first << ", " << maxEdge.second << "):" << endl;

        for (const auto& community : communities) {

            for (int member : community) {

                cout << member << " ";

            }

            cout << endl;

        }

        cout << "Number of communities: " << communities.size() << endl;

    }

}

int main() {

    int V, E;

    cout << "Enter the number of vertices: ";

    cin >> V;

    cout << "Enter the number of edges: ";

    cin >> E;

    vector<vector<int>> graph(V);

    cout << "Enter the edges (u v):" << endl;

    for (int i = 0; i < E; ++i) {

        int u, v;

        cin >> u >> v;

        graph[u].push\_back(v);

        graph[v].push\_back(u);

    }

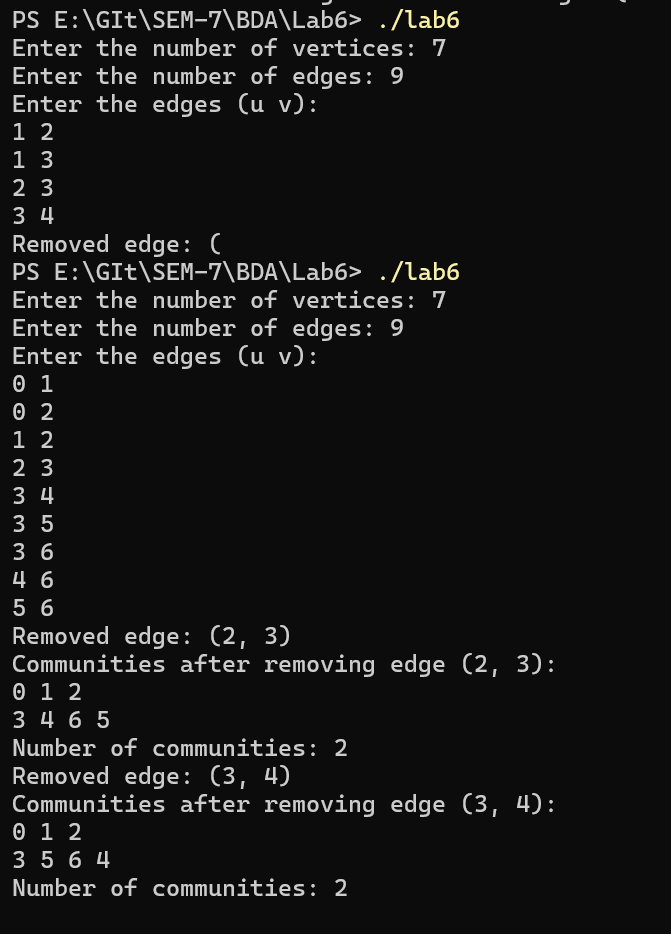
*// Run Girvan-Newman algorithm*

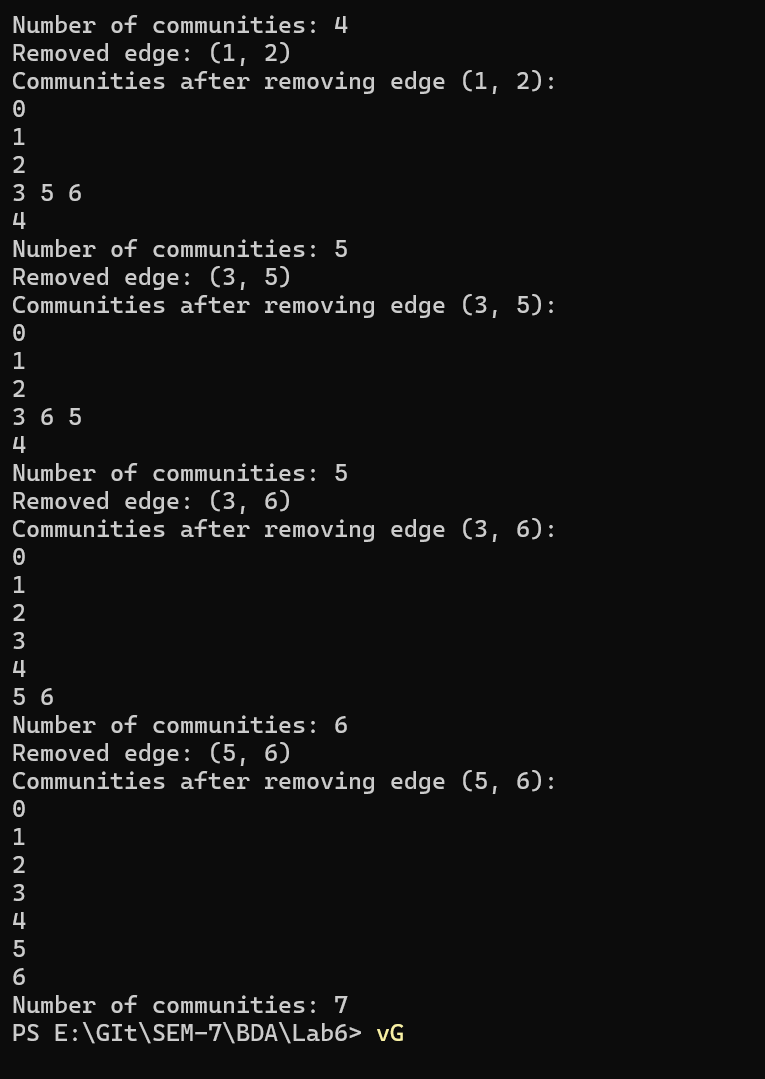
    girvanNewman(graph, V);

    return 0;

}

**Output:**

****

****