DFS

#include<iostream>

#include<vector>

using namespace std;

void DFS(const vector<vector<int>>&graph, vector<bool>&visited,vector<char>&V, int v) {

visited[v] = true;

cout << V[v];

for (int i = 0; i < graph.size(); i++) {

if (graph[v][i]) {

if (!visited[i]) {

DFS(graph, visited, V, i); //递归，以未遍历过的邻接结点为遍历初始顶点

}

}

}

}

int main() {

//顶点的集合

vector<char> V = { 'A','B','C','D' };

//标记向量

int n = V.size();

vector<bool> visited (n, false);

//图（邻接表）

vector<vector<int>> graph = {

{0, 1, 0, 0},

{1, 0, 1, 1},

{0, 1, 0, 0},

{0, 1, 0, 0}

};

DFS(graph, visited,V, 1);

}

BFS

#include<iostream>

#include<queue>

#include<vector>

using namespace std;

void BFS(vector<vector<int>>&graph, vector<bool>&visited, vector<char>&V, int v) {

visited[v] = true;

queue<int> Q;

Q.push(v); //不用太死板，这里可以直接把下标入队

while (!Q.empty()) {

v = Q.front(); //将v更新为队列中的第一个顶点的下标

cout << V[v] << " ";

Q.pop();

for (int i = 0; i < graph.size(); i++) {

if (graph[v][i] && !visited[i]) { //把与出队顶点邻接的顶点全部入队

visited[i] = true;

Q.push(i);

}

}

}

}

int main() {

//顶点的集合

vector<char> V = { 'A','B','C','D','E','F','G','H' };

//标记向量

int n = V.size();

vector<bool> visited (n, false);

//图（邻接表）

vector<vector<int>> graph = {

{0, 1, 0, 1,1,0,1,0},

{1, 0, 1, 0,1,1,0,0},

{0,1,0,1,0,1,0,1},

{1,0,1,0,0,0,1,1},

{1,1,0,0,0,0,0,0},

{0,1,1,0,0,0,0,0},

{1,0,0,1,0,0,0,0},

{0,0,1,1,0,0,0,0}

};

BFS(graph, visited,V, 0);

}

判断邻接表表示的无向图能否构成树

#include <iostream>

#include <vector>

using namespace std;

bool DFS(const vector<vector<int>>& graph, vector<bool>& visited, int node, int parent) {

visited[node] = true;

for (int adj = 0; adj < graph.size(); ++adj) {

if (graph[node][adj]) {

if (!visited[adj]) {

if (!DFS(graph, visited, adj, node))

return false;

} else if (adj != parent) {

// 发现环

return false;

}

}

}

return true;

//DFS的过程中主要是检验有没有环得到false也是因为发现了环 最后visited数组的作用是检验每一个顶点有没有全部遍历

}

bool isTree(const vector<vector<int>>& graph) {

int n = graph.size();

vector<bool> visited(n, false);

int edgeCount = 0;

// 计算边的数量

for (int i = 0; i < n; ++i) {

for (int j = i + 1; j < n; ++j) {

if (graph[i][j]) edgeCount++;

}

}

// 边的数量不是 n-1

if (edgeCount != n - 1) return false;

// 检查连通性和环

if (!DFS(graph, visited, 0, -1)) return false;

// 检查是否所有顶点都被访问（连通性）

for (bool v : visited) {

if (!v) return false;

}

return true;

}

int main() {

vector<vector<int>> graph = {

{0, 1, 0, 0},

{1, 0, 1, 1},

{0, 1, 0, 0},

{0, 1, 0, 0}

};

cout << (isTree(graph) ? "Graph is a tree" : "Graph is not a tree") << endl;

return 0;

}

//统计连通分量的个数

#include <iostream>

#include <vector>

using namespace std;

class Graph {

int V; // 顶点数

vector<vector<int>> adj; // 邻接矩阵

void DFSUtil(int v, vector<bool> &visited); // DFS的辅助函数

public:

Graph(int V); // 构造函数

void addEdge(int v, int w); // 添加边

int findConnectedComponents(); // 寻找连通分量

};

Graph::Graph(int V) {

this->V = V;

adj.resize(V, vector<int>(V, 0));

}

void Graph::addEdge(int v, int w) {

// 由于是无向图，所以需要双向连接

adj[v][w] = 1;

adj[w][v] = 1;

}

void Graph::DFSUtil(int v, vector<bool> &visited) {

// 标记当前节点为已访问

visited[v] = true;

// 对于所有相邻的顶点进行DFS

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

if (adj[v][i] == 1 && !visited[i]) {

DFSUtil(i, visited);

}

}

}

int Graph::findConnectedComponents() {

vector<bool> visited(V, false);

int count = 0;

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

if (!visited[v]) {

DFSUtil(v, visited);

count++;

}

}

return count;

}

int main() {

int n;

cin >> n;

Graph g(n);

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

int edge;

cin >> edge;

if (edge == 1) {

g.addEdge(i, j);

}

}

}

cout << g.findConnectedComponents() << endl;

return 0;

}