**01. BIỂU DIỄN ĐỒ THỊ**

6

1 1 0

3 0 0 2 0 4 0

1 1 0

1 4 0

3 1 0 3 0 5 0

1 4 0

typedef pair<int, int> ii;

typedef vector<ii> vii;

typedef vector<int> vi;

#define DFS\_WHITE -1

#define DFS\_BLACK 1

vector<vii> AdjList; Danh sách liền kề

int V, total\_neighbors, id, weight;

scanf("%d", &V);

AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList

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

scanf("%d", &total\_neighbors);

for (int j = 0; j < total\_neighbors; j++) {

scanf("%d %d", &id, &weight);

AdjList[i].push\_back(ii(id, weight));

}

}

**02. DFS (DEPTH FIRST SEARCH)**

vi dfs\_num; // this variable has to be global, we cannot put it in recursion

vi dfs\_discover;

int numCC;

void dfs(int u) { // DFS for normal usage: as graph traversal algorithm

printf(" %d", u); // this vertex is visited

dfs\_num[u] = DFS\_BLACK; // important step: we mark this vertex as visited

for (int j = 0; j < (int)AdjList[u].size(); j++) {

ii v = AdjList[u][j]; // v is a (neighbor, weight) pair

if (dfs\_num[v.first] == DFS\_WHITE) // important check to avoid cycle

dfs(v.first); // recursively visits unvisited neighbors v of vertex u

}

}

**03. BFS (BREATH FIRST SEARCH)**

map<int, int> dist; dist[s] = 0; // distance to source is 0 (default)

queue<int> q; q.push(s);

while (!q.empty()) {

int u = q.front(); q.pop(); // queue: layer by layer!

for (int j = 0; j < (int)AdjList[u].size(); j++) {

ii v = AdjList[u][j]; // for each neighbors of u

if (!dist.count(v.first)) { // dist.find(v.first) == dist.end() also ok

dist[v.first] = dist[u] + 1; // v unvisited + reachable

p[v.first] = u; // addition: the parent of vertex v->first is u

q.push(v.first); // enqueue v for next step

}

} }

Xem uva368 321 10047

**04. TÌM THÀNH PHẦN LIÊN THÔNG TRÊN ĐỒ THỊ VÔ HƯỚNG (CONNECTED COMPONENTS)**

vi dfs\_num; // this variable has to be global, we cannot put it in recursion

int numCC;

printThis("Standard DFS Demo (the input graph must be UNDIRECTED)");

numCC = 0;

dfs\_num.assign(V, DFS\_WHITE); // this sets all vertices' state to DFS\_WHITE

for (int i = 0; i < V; i++) // for each vertex i in [0..V-1]

if (dfs\_num[i] == DFS\_WHITE) // if that vertex is not visited yet

printf("Component %d:", ++numCC), dfs(i), printf("\n"); // 3 lines here!

printf("There are %d connected components\n", numCC);

**05. FLOOD FILL – LABELLING/COLOURING CONNECTED COMPONENTS**

printThis("Flood Fill Demo (the input graph must be UNDIRECTED)");

numCC = 0;

dfs\_num.assign(V, DFS\_WHITE);

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

if (dfs\_num[i] == DFS\_WHITE)

floodfill(i, ++numCC);

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

printf("Vertex %d has color %d\n", i, dfs\_num[i]);

**/\* Wetlands of Florida \*/**

// classic DFS flood fill

#include <iostream>

#include <string.h>

using namespace std;

#define REP(i, a, b) \

for (int i = int(a); i <= int(b); i++)

char line[150], grid[150][150];

int TC, R, C, row, col;

int dr[] = {1,1,0,-1,-1,-1, 0, 1}; // S,SE,E,NE,N,NW,W,SW

int dc[] = {0,1,1, 1, 0,-1,-1,-1}; // neighbors

int floodfill(int r, int c, char c1, char c2) {

if (r<0 || r>=R || c<0 || c>=C) return 0; // outside

if (grid[r][c] != c1) return 0; // we want only c1

grid[r][c] = c2; // important step to avoid cycling!

int ans = 1; // coloring c1 -> c2, add 1 to answer

REP (d, 0, 7) // recurse to neighbors

ans += floodfill(r + dr[d], c + dc[d], c1, c2);

return ans;

}

// inside the int main() of the solution for UVa 469 - Wetlands of Florida

int main() {

// read the implicit graph as global 2D array 'grid'/R/C and (row, col) query coordinate

sscanf(gets(line), "%d", &TC);

gets(line); // remove dummy line

while (TC--) {

R = 0;

while (1) {

gets(grid[R]);

if (grid[R][0] != 'L' && grid[R][0] != 'W') // start of query

break;

R++;

}

C = (int)strlen(grid[0]);

strcpy(line, grid[R]);

while (1) {

sscanf(line, "%d %d", &row, &col); row--; col--; // index starts from 0!

printf("%d\n", floodfill(row, col, 'W', '.')); // change water 'W' to '.'; count size of this lake

floodfill(row, col, '.', 'W'); // restore for next query

gets(line);

if (strcmp(line, "") == 0 || feof(stdin)) // next test case or last test case

break;

}

if (TC)

printf("\n");

}

return 0;

}

**06.TOPOLOGY SORT (DAG)**

vi topoSort; // global vector to store the toposort in reverse order

void dfs2(int u) { // change function name to differentiate with original dfs

dfs\_num[u] = DFS\_BLACK;

for (int j = 0; j < (int)AdjList[u].size(); j++) {

ii v = AdjList[u][j];

if (dfs\_num[v.first] == DFS\_WHITE)

dfs2(v.first);

}

topoSort.push\_back(u); }

printThis("Topological Sort (the input graph must be DAG)");

topoSort.clear();

dfs\_num.assign(V, DFS\_WHITE);

for (int i = 0; i < V; i++) // this part is the same as finding CCs

if (dfs\_num[i] == DFS\_WHITE)

dfs2(i);

reverse(topoSort.begin(), topoSort.end()); // reverse topoSort

for (int i = 0; i < (int)topoSort.size(); i++) // or you can simply read

printf(" %d", topoSort[i]); // the content of `topoSort' backwards

printf("\n");

**07. BIPARTITE GRAPH CHECK**

queue<int> q; q.push(s); // start from source

map<int, int> dist; dist[s] = 0; // distance to source is 0 (default)

Bool isBipartite = true; // addition of one boolean flag, initially true

while (!q.empty()) {

int u = q.front(); q.pop(); // queue: layer by layer!

if (dist[u] != layer) printf("\nLayer %d:", dist[u]);

layer = dist[u];

printf(", visit %d", reverseMapper[u]); // reverseMapper maps index to actual vertex label

for (int j = 0; j < (int)AdjList[u].size(); j++) {

ii v = AdjList[u][j]; // for each neighbors of u

if (!dist.count(v.first)) { // dist.find(v.first) == dist.end() also ok

dist[v.first] = dist[u] + 1; // v unvisited + reachable

p[v.first] = u; // addition: the parent of vertex v->first is u

q.push(v.first); // enqueue v for next step

}

else if ((dist[v.first] % 2) == (dist[u] % 2)) // same parity

isBipartite = false;

} }

**08. GRAPH EDGE PROPERTY CHECK**

printThis("Graph Edges Property Check");

dfs\_num.assign(V, DFS\_WHITE); dfs\_parent.assign(V, -1);

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

if (dfs\_num[i] == DFS\_WHITE)

printf("Component %d:\n", ++numCC), graphCheck(i); // 2 lines in one

void graphCheck(int u) { // DFS for checking graph edge properties

dfs\_num[u] = DFS\_GRAY; // color this as DFS\_GRAY (temp) instead of DFS\_BLACK

for (int j = 0; j < (int)AdjList[u].size(); j++) {

ii v = AdjList[u][j];

if (dfs\_num[v.first] == DFS\_WHITE) { // Tree Edge, DFS\_GRAY to DFS\_WHITE

dfs\_parent[v.first] = u; // parent of this children is me

graphCheck(v.first);

}

else if (dfs\_num[v.first] == DFS\_GRAY) { // DFS\_GRAY to DFS\_GRAY

if (v.first == dfs\_parent[u]) // to differentiate these two cases

printf(" Bidirectional (%d, %d) - (%d, %d)\n", u, v.first, v.first, u);

else // the most frequent application: check if the given graph is cyclic

printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);

}

else if (dfs\_num[v.first] == DFS\_BLACK) // DFS\_GRAY to DFS\_BLACK

printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);

}

dfs\_num[u] = DFS\_BLACK; // after recursion, color this as DFS\_BLACK (DONE)

}

**09. TÌM ĐIỂM KHỚP HOẶC CẦU (ARTICULATION/BRIDGE)**

vi dfs\_low; // additional information for articulation points/bridges/SCCs

vi articulation\_vertex;

int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {

dfs\_low[u] = dfs\_num[u] = dfsNumberCounter++; // dfs\_low[u] <= dfs\_num[u]

for (int j = 0; j < (int)AdjList[u].size(); j++) {

ii v = AdjList[u][j];

if (dfs\_num[v.first] == DFS\_WHITE) { // a tree edge

dfs\_parent[v.first] = u;

if (u == dfsRoot) rootChildren++; // special case, count children of root

articulationPointAndBridge(v.first);

if (dfs\_low[v.first] >= dfs\_num[u]) // for articulation point

articulation\_vertex[u] = true; // store this information first

if (dfs\_low[v.first] > dfs\_num[u]) // for bridge

printf(" Edge (%d, %d) is a bridge\n", u, v.first);

dfs\_low[u] = min(dfs\_low[u], dfs\_low[v.first]); // update dfs\_low[u]

}

else if (v.first != dfs\_parent[u]) // a back edge and not direct cycle

dfs\_low[u] = min(dfs\_low[u], dfs\_num[v.first]); // update dfs\_low[u]

} }

printThis("Articulation Points & Bridges (the input graph must be UNDIRECTED)");

dfsNumberCounter = 0; dfs\_num.assign(V, DFS\_WHITE); dfs\_low.assign(V, 0);

dfs\_parent.assign(V, -1); articulation\_vertex.assign(V, 0);

printf("Bridges:\n");

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

if (dfs\_num[i] == DFS\_WHITE) {

dfsRoot = i; rootChildren = 0;

articulationPointAndBridge(i);

articulation\_vertex[dfsRoot] = (rootChildren > 1); } // special case

printf("Articulation Points:\n");

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

if (articulation\_vertex[i])

printf(" Vertex %d\n", i);

**10. TÌM THÀNH PHẦN LIÊN THÔNG MẠNH (STRONGLY CONNECTED COMPONENTS)**

vi S, visited; // additional global variables

int numSCC;

void tarjanSCC(int u) {

dfs\_low[u] = dfs\_num[u] = dfsNumberCounter++; // dfs\_low[u] <= dfs\_num[u]

S.push\_back(u); // stores u in a vector based on order of visitation

visited[u] = 1;

for (int j = 0; j < (int)AdjList[u].size(); j++) {

ii v = AdjList[u][j];

if (dfs\_num[v.first] == DFS\_WHITE)

tarjanSCC(v.first);

if (visited[v.first]) // condition for update

dfs\_low[u] = min(dfs\_low[u], dfs\_low[v.first]);

}

if (dfs\_low[u] == dfs\_num[u]) { // if this is a root (start) of an SCC

printf("SCC %d:", ++numSCC); // this part is done after recursion

while (1) {

int v = S.back(); S.pop\_back(); visited[v] = 0;

printf(" %d", v);

if (u == v) break;

}

printf("\n");

} }

printThis("Strongly Connected Components (the input graph must be DIRECTED)");

dfs\_num.assign(V, DFS\_WHITE); dfs\_low.assign(V, 0); visited.assign(V, 0);

dfsNumberCounter = numSCC = 0;

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

if (dfs\_num[i] == DFS\_WHITE)

tarjanSCC(i);