**★★★★☆**

**題組：Problem Set Archive** **with Online Judge**

**題號：11770:** **Lighting Away**

**解題者：羅巧雯**

**解題日期：2025年5月1日**

**題意：**

夜間警衛要點亮草地上的燈。某些燈會自動觸發其他燈亮起。他希望手動點亮最少數量的燈，就能讓所有燈都亮。

**題意範例：**輸入

 5 4

 1 2

 1 3

 3 4

 5 3

 表示有5盞燈，4個關係，1會觸發2,3，3會觸發4，5會觸發3。因此最少要手動點亮 2 盞燈（燈 1 和 5），才能讓全部 5 盞燈都亮。

**解法：**

我們可以用 Tarjan 演算法 來解這題。首先，我們要找出圖中的所有強連通分量（SCC）。我們透過深度優先搜尋（DFS）來完成這個步驟，並且記錄每個節點的訪問順序以及它最早能回溯到的順序。當我們發現某個節點的 low 值等於它自己的 discovery 值時，就代表找到了一個完整的 SCC。接下來，我們將每個 SCC 縮成一個超節點，建立一張新的圖，稱為 SCC DAG。如果原圖中有一條邊從節點 a 指向節點 b，且 a 和 b 分屬不同的 SCC，那麼在新圖中會建立一條從 SCC\_a 指向 SCC\_b 的邊。由於強連通分量的特性，這張圖一定是沒有環的。接著我們計算這張 圖中每個 SCC 節點的入度，也就是有多少其他 SCC 有邊連向它。入度為 0 的 SCC 表示沒有其他 SCC 可以影響它，因此這些 SCC 至少要各自手動啟動一次，才能開始點亮整個圖。最後，統計入度為 0 的 SCC 數量，就是我們要找的答案。

**解法範例：**

以前面的範例作為例子的話，有四個關係 1 ⮕2、1 ⮕3、3 ⮕4、5 ⮕3。使用 Tarjan 演算法尋找SCC，從節點 1 開始做DFS。節點 1 先後訪問節點 2 和節點 3：節點 2 無出邊，形成一個單獨的 SCC。節點 3 繼續訪問節點 4，節點 4 也無出邊，形成一個 SCC。回到節點 3，它自身也構成一個 SCC。接著從節點 5 開始 DFS，訪問節點 3，不過節點3已訪問過，不影響低鏈值，最終節點 5 也形成一個單獨的 SCC。最後統計入度為0的SCC，分別是SCC0跟SCC4。



**討論：**

我原本使用貪婪演算法，每次重新選擇起點，看哪個起點可以點亮最多的燈，重複步驟這樣時間負責度過高，會time limit。再來我使用Kosaraju 演算法找出SCC:首先建立正向圖和反向圖，用DFS1建立拓撲排序，再用DFS2在反向圖上找SCC，接著建立SCC之間的圖，最後計算入度為0的數量，這樣的方法只需要兩次DFS，每個節點和邊只會被訪問一次。在今天報告後，根據老師說可以不使用反向圖，因此我又重新思考過後使用Tarjan 演算法來作題，這樣只需要一次 DFS而且不需要反向圖，時間複雜度依然是O(V+E)。

**程式：**

#include <bits/stdc++.h>

using namespace std;

const int MAXN = 10001;

vector<int> graph[MAXN];

vector<int> disc; // discovery time

vector<int> low; // lowest discovery time

vector<bool> inStack; // 檢查節點是不是在stack中

stack<int> st; // 儲存目前SCC節點

vector<vector<int>> allSCC; // 所有SCC

int discovery\_time;

void tarjan(int u) {

 disc[u] = low[u] = ++discovery\_time;

 st.push(u);

 inStack[u] = true;

 for(int v : graph[u]) {

 if(disc[v] == 0) { // 如果v還沒走過

 tarjan(v);

 low[u] = min(low[u], low[v]);

 }

 else if(inStack[v]) { // 如果v在stack中

 low[u] = min(low[u], disc[v]);

 }

 }

 // u是scc的根

 if(low[u] == disc[u]) {

 vector<int> scc;

 while(true) {

 int v = st.top();

 st.pop();

 inStack[v] = false;

 scc.push\_back(v);

 if(v == u) break;

 }

 allSCC.push\_back(scc);

 }

}

int main() {

 int T;

 cin >> T;

 for(int Case = 1; Case <= T; Case++) {

 int N, M;

 cin >> N >> M;

 //初始化

 for(int i = 1; i <= N; i++) {

 graph[i].clear();

 }

 //讀取邊

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

 int a, b;

 cin >> a >> b;

 graph[a].push\_back(b);

 }

 //初始化

 disc.assign(N + 1, 0);

 low.assign(N + 1, 0);

 inStack.assign(N + 1, false);

 allSCC.clear();

 discovery\_time = 0;

 for(int i = 1; i <= N; i++) {

 if(disc[i] == 0) {

 tarjan(i);

 }

 }

 // 建立SCC DAG

 vector<vector<int>> scc\_graph(allSCC.size());

 vector<int> node\_to\_scc(N + 1);

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

 for(int node : allSCC[i]) {

 node\_to\_scc[node] = i;

 }

 }

 //建立邊

 for(int i = 1; i <= N; i++) {

 for(int next : graph[i]) {

 if(node\_to\_scc[i] != node\_to\_scc[next]) {

 scc\_graph[node\_to\_scc[i]].push\_back(node\_to\_scc[next]);

 }

 }

 }

 // 數入度為0

 vector<int> in\_degree(allSCC.size(), 0);

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

 for(int next : scc\_graph[i]) {

 in\_degree[next]++;

 }

 }

 int result = 0;

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

 if(in\_degree[i] == 0) {

 result++;

 }

 }

 cout << "Case " << Case << ": " << result << endl;

 if(Case < T) {

 string empty;

 getline(cin, empty);

 getline(cin, empty);

 }

 }

 return 0;

}