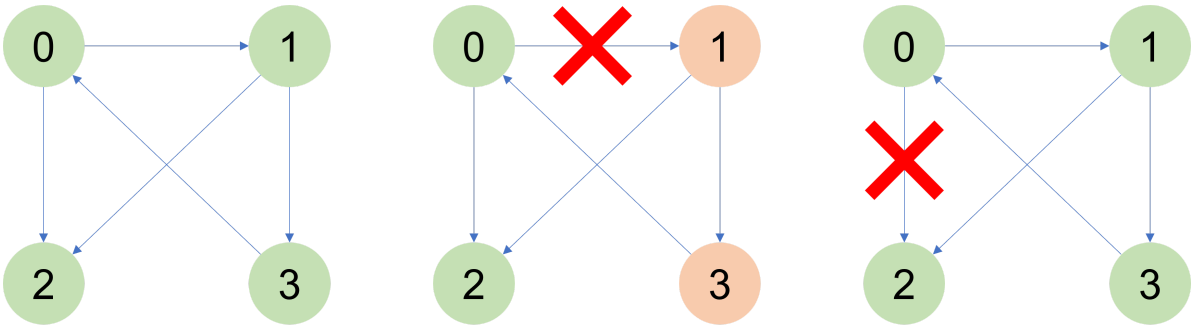
OJ5 信息传递

问题描述

某组织在执行任务时需要进行信息传递，为保证信息传递的安全性，每个节点只能向特定的某些节点发送消息，已知有N个节点，初始信息由0号节点开始发送，现在系统的设计师需要确认所设计的连接方式能否使得所有节点最终都收到0号节点发送的消息。为了保证系统的鲁棒性，系统设计师还需要考虑若有信道损坏是否会影响信息的传递。具体来说，给定M条可能毁坏的信道，需要依次考虑只有一条信道被毁坏时是否所有节点仍能收到0号节点发送的信息。下面是一幅与输入输出样例对应的示意图，绿色节点能收到0号节点发送的信息，橙色节点则无法收到。



输入格式

输入共N+M+1行

第一行为两个正整数，第一个正整数N表示节点总数，第二个正整数M表示可能被破坏的信道数

接下来的N行依次给出从0号节点到N-1号节点能发送消息的节点；每行第一个数n表示该节点能传递消息的节点个数，后面n个数表示能传递消息的节点编号，同一行的数之间用空格分隔。

接下来的M行对应M个可能毁坏的信道，每行两个数i j，用空格分隔，表示从节点i到节点j的信道毁坏。

输出格式

输出为M+1行，每行一个正整数，第一行对应原始情况，后面M行依次对应M条信道毁坏后的情况。若对应情况下所有节点都能收到0号节点发送的消息，则输出1，反之输出0。

输入样例

```
4 2
2 1 2
2 2 3
0
1 0
0 1
0 2
```

输出样例

1
0
1

提示

1. 测试样例情况：

   1-4测试样例：N<2^{10}，M=0

   5-7测试样例：N<2^{16}，M=0

   8-10测试样例：N<2^{16}，M<2^{16}

   对于所有测试样例，边数E<2^{20}

2. 图中无自环、重边

解题思路

本题实质上是考察判断一个图是否连通，删掉一条边之后图是否仍然连通。

1. 读入的数据天然构成一个邻接表，因此考虑使用二维向量`std::vector<std::vector<int>> graph`存储边，读取每一行时建立一个临时向量`std::vector<int> temp`，再`push_back`到`graph`。

2. 对于原始的图是否连通，使用深度优先遍历算法（广度优先亦可，总之要遍历一遍图），使用计数器`sum`判断是否访问过图中所有`N`个节点。最后图连通的标志是`sum == N`，也即所有节点都能收到0号节点发送的消息。

   深度优先遍历`int DFS_1(std::vector<std::vector<int>> &graph, int N)`的思路为：

   使用`std::vector<int> stk`实现栈功能，`stk.push_back(node)`相当于压栈，`stk.pop_back()`相当于出栈，不需要再使用`std::stack<int>`，可以节省一个头文件`#include<stack>`的内存。

   使用`std::vector<bool> visited`，记录某节点是否被访问过，初始时均值为`false`，被访问过置为`true`。

   接下来开始遍历。0首先入栈，然后进行一个循环：当前栈不为空时，弹出栈顶，并把栈顶所有未被访问的出度按序压入栈，并标记他们为已访问过。每访问过一个元素，计数器`sum`自增，当`sum == N`时即可退出循环，输出1并返回1，若最终`sum < N`，则输出0并返回0

3. 有损毁信道的情况下（即删掉一条边）：

   ①原本图不连通，有损毁时更不可能连通，输出0

   ②原本图连通，假设损毁信道为`start --> end`，只需判断`end`是否仍然能够收到`0`的信息，也即是否存在路径`0 --> ... --> end`，也即我们在使用深度优先算法时，此时节点的某一出度是否为`end`，同时需要额外判断此节点应该不为`start`，因为此边不复存在。

   此处的深度优先算法需要做一些调整，函数形参需要额外的损毁信道的起终点：`int start, int end`，故调整为`DFS_2(std::vector<std::vector<int>> &graph, int N, int start, int end)`：

   `start`首先入栈，将`start`的除了end以外所有的出度压栈，再次进行上面`DFS_1`的循环，在循环中进行如下判断：若当前起点不为`start`且终点为`end`，说明存在其他路径可以到达`end`，返回1并在主函数中输出1；

```
while (stk.size() != 0)
    {
        int temp = stk[stk.size() - 1]; // 取栈顶元素的值
```

```
        stk.pop_back();
        for (int i = 1; i <= graph[temp][0]; i++)
        {
            if (visited[graph[temp][i]] == false)
            {
                if (end != graph[temp][i])
                {
                    stk.push_back(graph[temp][i]);
                    visited[graph[temp][i]] = true;
                }
                else if (start != temp && end == graph[temp][i])
                {
                    return 1;
                }
            }
        }
    }
```

代码（C++）

```cpp
#include <cstdio>
#include <vector>

int DFS_1(std::vector<std::vector<int>> &graph, int N)
{
    std::vector<int> stk;
    std::vector<bool> visited(N, false);

    stk.push_back(0); // 栈实现深度优先遍历
    visited[0] = true;
    int sum = 1;

    while (stk.size() != 0)
    {
        int temp = stk[stk.size() - 1];
        stk.pop_back();
        for (int i = 1; i <= graph[temp][0]; i++)
        {
            if (visited[graph[temp][i]] == false)
            {
                stk.push_back(graph[temp][i]);
                visited[graph[temp][i]] = true;
                sum++;

                if (sum == N)
                {
                    printf("%d\n", 1);
                    return 1;
                }
            }
        }
    }
    if (sum == N)
    {
        printf("%d\n", 1);
```

```
        return 1;
    }
    else
    {
        printf("%d\n", 0);
        return 0;
    }
}

int DFS_2(std::vector<std::vector<int>> &graph, int N, int start, int end)
{
    std::vector<int> stk;
    std::vector<bool> visited(N, false);

    stk.push_back(start); // 栈实现深度优先遍历
    visited[start] = true;
    // 首先把start的除了end所有的出度压栈
    for (int i = 0; i <= graph[start][0]; i++)
    {
        if (graph[start][i] != end)
        {
            stk.push_back(graph[start][i]);
            visited[graph[start][i]] = true;
        }
    }

    while (stk.size() != 0)
    {
        int temp = stk[stk.size() - 1]; // 取栈顶元素的值
        stk.pop_back();
        for (int i = 1; i <= graph[temp][0]; i++)
        {
            if (visited[graph[temp][i]] == false)
            {
                if (end != graph[temp][i])
                {
                    stk.push_back(graph[temp][i]);
                    visited[graph[temp][i]] = true;
                }
                else if (start != temp && end == graph[temp][i])
                {
                    return 1;
                }
            }
        }
    }
    return 0;
}

int main()
{
    int N; // 节点总数
    int M; // 可能破坏的信道数
    scanf("%d%d", &N, &M);
    std::vector<std::vector<int>> graph; // 邻接表实现

    for (int i = 0; i < N; i++)
    {
```

```
        int num; // 该节点能发送的消息数
        scanf("%d", &num);
        std::vector<int> temp(num + 1);
        temp[0] = num;
        for (int j = 1; j <= num; j++)
        {
            scanf("%d", &temp[j]); // 信道i至temp[j]
        }
        graph.push_back(temp);
    }

    int flag = DFS_1(graph, N); // 函数体内已经打印过1/0

    if (flag == 0)
    {
        // 原本就不能所有节点都受到，现在损坏后也不能
        for (int i = 0; i < M; i++)
        {
            printf("%d\n", 0);
        }
    }
    else
    {
        // 原本可以收到所有节点，先只需考虑损坏信道的终点能不能收到
        for (int i = 0; i < M; i++)
        {
            int start, end; // 损坏信道的起点和终点
            scanf("%d%d", &start, &end);
            if (DFS_2(graph, N, start, end) == 1)
            {
                printf("%d\n", 1);
            }
            else
            {
                printf("%d\n", 0);
            }
        }
    }
    return 0;
}
```