## 1390. Car race

In city N, a stage of the World Championship for Formula-0 class cars will take place in the near future. Since the organizers did not manage to build a special racetrack for this competition, it was decided to hold the race on the city streets.

City N has *n* intersections, some of which are connected by roads with two-way traffic. Moreover, any two intersections are connected by no more than one road, and it is possible to travel between any pair of intersections via the existing roads.

The race track must be circular, meaning it must start and end at the same intersection, and no intersection should be visited more than once along the way.

During the initial preparation stage, the organizing committee compiled a list of all the roads in the city, and now it's time to use it. The first question that needs to be answered is whether a suitable circular track exists in the city. Of course, if the answer is negative, the organizers will have to urgently build a few more roads. However, there is a problem: the organizers suspect that some roads in the list might be listed more than once, as the list was not compiled very carefully.

Write a program that, given the list of roads, determines whether a circular track can be organized in the city.

**Input.** The first line contains two integers: the number of intersections *n* (1 ≤ *n* ≤ 1000) in the city N and the number of roads *m* (0 ≤ *m* ≤ 105) in the list.

The next *m* lines describe the roads. Each road is defined by two numbers *u* and *v* (1 ≤ *u*, *v* ≤ *n*, *u* ≠ *v*) – the intersection numbers it connects. Since the roads are bidirectional, the pairs of numbers (*u*, *v*) and (*v*, *u*) describe the same road.

**Output.** Print “YES” if a circular race track can be organized in the city, and “NO” otherwise.

|  |  |
| --- | --- |
| **Sample input 1** | **Sample output 1** |
| 3 4  1 2  2 3  3 1  3 2 | YES |
|  |  |
| **Sample input 2** | **Sample output 2** |
| 2 3  1 2  2 1  2 1 | NO |

## SOLUTION

**graphs – cycles**

# Algorithm analysis

The problem provides an undirected connected graph. It is necessary to check whether it contains a cycle (which can be turned into a Formula-0 track).

An undirected graph contains a cycle if there is a back edge, i.e., an edge leading to a vertex that is already visited.

# Example

The graphs given in the example are as follows:



The first graph contains a cycle, while the second does not.

# Algorithm implementation

The graph is stored in the adjacency matrix *g*. The array *used* is used to mark the visited vertices.

#define MAX 1010

int g[MAX][MAX], used[MAX];

The ***dfs*** function implements depth-first search from vertex *v*. It is necessary to exclude the case when we move from *v* to its ancestor: the ancestor is already visited, but there is no cycle. To handle this, we will introduce a second parameter *p* in the ***dfs*** function, representing the ancestor of vertex *v*.

void dfs(int v, int p = -1)

{

Mark vertex *v* as visited.

used[v] = 1;

Iterate over the vertices *i* that can be reached from *v*.

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

{

Consider all edges except for the one that leads to the ancestor *p*.

if (i == p) continue;

If there is an edge from *v* to *i*, and *i* is already visited (used[*i*] = 1), then the graph contains a cycle. Set *flag* = 1.

if (g[v][i] == 1)

{

if (used[i] == 1) flag = 1;

Otherwise, continue the depth-first search from vertex *i*.

else dfs(i, v);

}

}

}

The main part of the program. Read the input data.

scanf("%d %d",&n,&m);

memset(g,0,sizeof(g));

memset(used,0,sizeof(used));

Read the input undirected graph. The graph is stored in an adjacency matrix, so repeated roads will be counted only once.

for (i = 0; i < m; i++)

{

scanf("%d %d",&u,&v);

g[u][v] = g[v][u] = 1;

}

Set *flag* = 0, which indicates the absence of a cycle in the graph. If a cycle is found, the value of *flag* will change to 1.

flag = 0;

Initiate depth-first search from vertex 1 (according to the problem statement, the graph is connected).

dfs(1);

Based on the value of *flag*, print the result.

if (flag) printf("YES\n");

else printf("NO\n");

# Java implementation

**import** java.util.\*;

**public** **class** Main

{

**static** **int** *g*[][], *used*[];

**static** **int** *n*, *m*, *flag*;

**static** **void** dfs(**int** v, **int** prev)

{

*used*[v] = 1;

**for**(**int** i = 1; i <= *n*; i++)

**if** ((i != prev) && *g*[v][i] == 1)

**if** (*used*[i] == 1) *flag* = 1; **else** *dfs*(i,v);

}

**public** **static** **void** main(String[] args)

{

Scanner con = **new** Scanner(System.***in***);

*n* = con.nextInt();

*m* = con.nextInt();

*g* = **new** **int**[*n*+1][*n*+1];

*used* = **new** **int**[*n*+1];

**for**(**int** i = 0; i < *m*; i++)

{

**int** u = con.nextInt();

**int** v = con.nextInt();

*g*[u][v] = *g*[v][u] = 1;

}

*flag* == 0;

*dfs*(1,-1);

**if** (*flag* == 1) System.***out***.println("YES");

**else** System.***out***.println("NO");

}

}

# Python implementation

The ***dfs*** function implements depth-first search from vertex *v*. It is necessary to exclude the case when we move from *v* to its ancestor: the ancestor is already visited, but there is no cycle. To handle this, we will introduce a second parameter *p* in the ***dfs*** function, representing the ancestor of vertex *v*.

**def** dfs(v, p = -1):

**global** flag

Mark vertex *v* as visited.

used[v] = 1

Iterate over the vertices *i* that can be reached from *v*.

**for** i **in** range(1, n + 1):

Consider all edges except for the one that leads to the ancestor *p*.

**if** i == p:

**continue**

If there is an edge from *v* to *i*, and *i* is already visited (used[*i*] = 1), then the graph contains a cycle. Set *flag* = 1.

**if** g[v][i] == 1:

**if** used[i] == 1:

flag = 1

Otherwise, continue the depth-first search from vertex *i*.

**else**:

dfs(i, v)

The main part of the program. Read the input data.

n, m = map(int, input().split())

g = [[0] \* (n + 1) **for** \_ **in** range(n + 1)]

used = [0] \* (n + 1)

Read the input undirected graph. The graph is stored in an adjacency matrix, so repeated roads will be counted only once.

**for** \_ **in** range(m):

a, b = map(int, input().split())

g[a][b] = g[b][a] = 1

Set *flag* = 0, which indicates the absence of a cycle in the graph. If a cycle is found, the value of *flag* will change to 1.

flag = 0

Initiate depth-first search from vertex 1 (according to the problem statement, the graph is connected).

dfs(1)

Based on the value of *flag*, print the result.

**if** flag == 1:

print(**"YES"**)

**else**:

print(**"NO"**)