**Grafos Implícitos**

Imagine o seguinte "labirinto":

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | @ | @ | @ | 6 | 7 |
| 2 | 8 | 9 | A | B | @ |
| 3 | @ | @ | @ | @ | @ |
| 4 | C | D | E | F | G |

É possível ir do "1" até o "A" sem passar/bater nas paredes (simbolizadas pelos @) e apenas andando nos sentidos básicos (cima, baixo, direita e esquerda)?

Resposta: SIM!

Um dos possíveis caminhos seria:

1 -> 2 -> 3 -> 4 -> 6 -> B -> A

Outro caminho seria:

1 -> 2 -> 3 -> 4 -> 5 -> 7 -> 6 -> B -> A

E temos vários outros possíveis caminhos, mas, por exemplo, não é possível ir do "1" para o "C"!

Para resolvermos esse tipo de problema, poderíamos criar um grafo, e dizer que os seus vizinhos são aqueles que estão do seu lado, desconsiderando as paredes!

Por exemplo:

Os vizinhos do 6 são:

4 (em cima)

7 (direita)

B (embaixo).

Os vizinhos do 5 são:

4(esquerda)

7(embaixo).

E então usar um DFS e descobrir se esses "vértices" pertencem ou não ao mesmo Componente Conexo.

Mas acredite, isso será desnecessário!

Não precisamos criar um grafo para isso pois o grafo já está implicitamente nessa matriz, nesse GRID!

Já sabemos quem são os vizinhos simplesmente pelas suas posições!

Por exemplo:

O número 6 está na posição: (linha 1, coluna 3), ou seja:

Seus vizinhos, SEMPRE serão:

Linha 1, coluna 2 (esquerda)

Linha 1, coluna 4 (direita)

Linha 0, coluna 3 (em cima)

Linha 2, coluna 3 (embaixo)

A única coisa que temos que SEMPRE verificar é se algum desses vizinhos não é uma parede, ou se é uma posição fora do grid!

Por exemplo:

O número 1 não tem vizinho nem para cima nem para esquerda!

O número 3 não tem vizinho para cima

A letra F não tem vizinho para baixo

E etc...

Para facilitar nossas vidas, para não enchermos o código com um bando de "if", o que podemos fazer é: envolta do nosso grid, vamos colocar um monte de parede, simbolizando bordas!

Aquele nosso grid ficará algo assim:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | @ | @ | @ | @ | @ | @ | @ |
| 1 | @ | 1 | 2 | 3 | 4 | 5 | @ |
| 2 | @ | @ | @ | @ | 6 | 7 | @ |
| 3 | @ | 8 | 9 | A | B | @ | @ |
| 4 | @ | @ | @ | @ | @ | @ | @ |
| 5 | @ | C | D | E | F | G | @ |
| 6 | @ | @ | @ | @ | @ | @ | @ |

Porque desta forma, nossa verificação será sempre apenas: "vizinho é diferente de @ ?"

Desta forma, o número 1 tem vizinho tanto para esquerda, quanto para a direita, quanto para cima, quanto para baixo, independentemente se algum deles são paredes!

E obviamente, os índices agora mudam!

Nosso grid sempre começará na linha 1 e na coluna 1!

Isso quer dizer que o nosso número 6 não está mais na posição (linha 1, coluna 3) e sim na posição (linha 2, coluna 4)

E seus vizinhos:

Linha 2, coluna 3 (esquerda)

Linha 2, coluna 5 (direita)

Linha 1, coluna 4 (em cima)

Linha 3, coluna 4 (embaixo)

E agora sim, usamos o DFS e verificamos se ambos pertencem ao mesmo Componente Conexo!

Vamos aos códigos:

Problema 1907 do URI Online Judge, da qual na prática pergunta quantos componentes conexos existem naquele grid, aonde “o” (letra “o” minúscula) simbolizam as paredes:

#include <bits/stdc++.h>

#define MAXLINHAS 1030 // SEMPRE um pouco maior que o valor indicado no enunciado

#define MAXCOLUNAS 1030 // SEMPRE um pouco maior que o valor indicado no enunciado

using namespace std; // Básico de C++

using ii = pair<int,int>; // Apenas para evitar escrever "pair <int,int>" um monte de vezes no código

char grid[MAXLINHAS][MAXCOLUNAS]; // GRID = Grafo Implícito

bool visited[MAXLINHAS][MAXCOLUNAS]; // Básico de Grafos

vector <ii> movimentos = {

{0, 1}, // BAIXO

{1, 0}, // DIREITA

{0, -1}, // CIMA

{-1, 0} // ESQUERDA

};

int qtdCmpCx; // Quantidade de Ilhas

void dfs(int linha, int coluna, int qtdCmpCx){

visited[linha][coluna] = true;

for (auto &movimento : movimentos){

int dLinha = movimento.first, dColuna = movimento.second;

if (!visited[linha + dLinha][coluna + dColuna] && grid[linha + dLinha][coluna + dColuna] != 'o'){

dfs(linha + dLinha, coluna + dColuna, qtdCmpCx);

}

}

}

void inic(){

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

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

grid[i][j] = 'o'; // Todo grid será uma “parede” no início

visited[i][j] = false; // Ninguém foi visitado ainda

}

}

}

int main(){

int N, M; // N = quantidade de linhas

cin >> N >> M; // M = quantidade de colunas

// Zerando GRID e Visited:

inic();

// Populando GRID:

for (int i = 1; i <= N; i++){ // Sempre começaremos no 1

for (int j = 1; j <= M; j++){ // Sempre começaremos no 1,

cin >> grid[i][j];

}

}

// DFS:

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

for (int coluna = 1; coluna <= M; coluna++){

if (!visited[linha][coluna] && grid[linha][coluna] != 'o'){

dfs(linha,coluna,qtdCmpCx);

qtdCmpCx++;

}

}

}

cout << qtdCmpCx << endl;

return 0;

}