# M. Цивилизация

Two structures, node and edge, are defined in the code, which are used to construct the adjacency list representation of the graph.

The input part reads the size of the map, the coordinates of the start and end points, and initializes some data structures.

The init\_edges function builds the adjacency list of the graph based on the feasible paths in the map.

The start\_find\_dij function starts to execute Dijkstra's algorithm.

First, initialize the distance from the starting point to each node to infinity, set the distance from the starting point to 0, and add the starting point to the priority queue.

After that, it enters the loop, and each time the node with the smallest distance is taken out of the priority queue.

If the node has been visited, skip the current loop, otherwise mark it as visited.

Traverse all the neighbor nodes of this node, update the shortest distance from the starting point to the neighbor nodes, and add the neighbor nodes to the priority queue.

Loop until the end node is visited or the priority queue is empty.

Finally, according to the shortest distance array and path auxiliary array, output the shortest path length and path.

Code：

#include <iostream>

#include <queue>

#include <vector>

#include <stack>

#include <algorithm>

using namespace std;

// n - 行，m - 列

int n, m, sx, sy, ex, ey;

vector<int> distances;

vector<bool> visited;

vector<int> path\_help;

int infinity = 2000001;

int index\_node = 0;

vector<vector<int>> maps;

vector<struct node \*> link\_maps;

//此处的xy是matrix里的

struct node

{

int index;

struct edge \*next;

struct edge \*end;

};

struct edge

{

int length;

struct node \*next;

struct edge \*neiber;

};

vector<string> split(string s)

{

vector<string> result;

while (s.find(" ") != string::npos)

{

int p = s.find(" ");

result.emplace\_back(s.substr(0, p));

s = s.substr(p + 1, s.size());

}

result.emplace\_back(s);

return result;

}

void add\_end(struct node \*n, struct edge \*e)

{

if (n->next == NULL)

{

n->next = e;

n->end = e;

return;

}

struct edge \*f = n->end;

f->neiber = e;

n->end = e;

}

//输入map的ij

int get\_index(int x, int y)

{

return x \* m + y;

}

// i1 j1-link\_maps, i j-目标

void set\_edges(int i1, int j1, int i, int j)

{

string direc = "";

if (maps[i][j] == infinity)

{

return;

}

struct edge \*e = new struct edge;

e->length = maps[i][j];

e->neiber = NULL;

e->next = link\_maps[get\_index(i, j)];

add\_end(link\_maps[get\_index(i1, j1)], e);

}

void init\_edges()

{

int k = 0;

int i = 0;

do {

int j = 0;

do {

if (maps[i][j] != infinity) {

if (i - 1 >= 0 && maps[i - 1][j] != infinity) {

set\_edges(i, j, i - 1, j);

}

if (i + 1 <= n - 1 && maps[i + 1][j] != infinity) {

set\_edges(i, j, i + 1, j);

}

if (j + 1 <= m - 1 && maps[i][j + 1] != infinity) {

set\_edges(i, j, i, j + 1);

}

if (j - 1 >= 0 && maps[i][j - 1] != infinity) {

set\_edges(i, j, i, j - 1);

}

}

j++;

} while (j < m);

i++;

} while (i < n);

}

// position and length

struct cmp

{

bool operator()(pair<int, int> a, pair<int, int> b)

{

return a.second >= b.second;

}

};

void print\_path(int i){

if (i == (sx - 1) \* m + sy - 1) {

return;

}

string path = "";

int tmp = (ex - 1) \* m + ey - 1;

do {

if (tmp - i == 1)

path.append("E");

if (tmp - i == -1)

path.append("W");

if (tmp - i > 1)

path.append("S");

if (tmp - i < -1)

path.append("N");

tmp = i;

} while ((i = path\_help[i]) != -1);

reverse(path.begin(), path.end());

cout << path <<endl;

}

void start\_find\_dij()

{

path\_help.resize(n\*m+1);

int r\_index = get\_index(sx - 1, sy - 1);

int l\_index = get\_index(ex - 1, ey - 1);

distances[r\_index] = 0;

path\_help[r\_index] = -1;

priority\_queue<pair<int, int>, vector<pair<int, int>>, cmp> que;

que.push(pair<int, int>(r\_index, 0));

while (!que.empty())

{

int index = que.top().first;

que.pop();

if (visited[l\_index])

{

break;

}

if (visited[index])

{

continue;

}

else

{

visited[index] = true;

}

struct edge \*e = link\_maps[index]->next;

while (e != NULL)

{

int next\_node = e->next->index;

if (!visited[next\_node] && distances[index] + e->length < distances[next\_node])

{

distances[next\_node] = distances[index] + e->length;

que.push(pair<int, int>(next\_node, distances[next\_node]));

path\_help[next\_node] = index;

}

e = e->neiber;

}

}

if (visited[l\_index])

{

cout << distances[l\_index] << endl;

print\_path(l\_index);

}

else

{

cout << -1 << endl;

}

}

void init\_vectors()

{

vector<int> distances\_init(n \* m, infinity);

vector<bool> visited\_init(n \* m, false);

distances = distances\_init;

visited = visited\_init;

}

int main()

{

string nums;

getline(cin, nums);

vector<string> all\_nums = split(nums);

n = stoi(all\_nums[0]);

m = stoi(all\_nums[1]);

sx = stoi(all\_nums[2]);

sy = stoi(all\_nums[3]);

ex = stoi(all\_nums[4]);

ey = stoi(all\_nums[5]);

init\_vectors();

// struct node\* path = new struct node[n\*m];

string line;

int i = 0;

do {

vector<int> v;

getline(cin, line);

int j = 0;

do {

switch (line[j]) {

case '.':

v.emplace\_back(1);

break;

case 'W':

v.emplace\_back(2);

break;

case '#':

v.emplace\_back(infinity);

int index = get\_index(i, j);

visited[index] = true;

break;

}

struct node\* dot = new struct node;

dot->index = index\_node;

index\_node++;

dot->next = NULL;

dot->end = NULL;

link\_maps.emplace\_back(dot);

j++;

} while (j < m);

maps.emplace\_back(v);

i++;

} while (i < n);

if ((n == 1 && m == 1) || (ex == sx && ey == sy))

{

cout << 0 << endl;

return 0;

}

init\_edges();

if (maps[ex - 1][ey - 1] == infinity || maps[sx - 1][sy - 1] == infinity)

{

cout << -1 << endl;

}

else

{

maps.clear();

start\_find\_dij();

}

return 0;

}

# N. Свинки-копилки

An integer array with a size of 101 is defined in the code, which is used to store and retrieve the node information.

The input part reads the integer n, indicating the number of nodes.

In the initialization phase, the parent node of each node is initialized to itself.

Iterate over the input, read the connection relationship of each node, and merge them into the same set. If the parent nodes of the two nodes are different, point the parent node of one node to the parent node of the other node to realize the merge operation.

Count the final number of connected components. Traversing each node, if the parent node of the node points to itself, it means that the node is a representative of the set, that is, the root node of the connected component, and the statistical value is increased by one.

Finally, the number of connected components is output.

Code：

#include <bits/stdc++.h>

using namespace std;

int disjoint\_set[101], n, answer = 0;

int check(int x)

{

if (disjoint\_set[x] != x) {

disjoint\_set[x] = check(disjoint\_set[x]);

}

return disjoint\_set[x];

}

int main()

{

int n;

ios::sync\_with\_stdio(0);

cin.tie(0);

cout.tie(0);

std::cin >> n;

int i = 1;

do {

disjoint\_set[i] = i;

i++;

} while (i <= n);

i = 1;

do {

int x;

std::cin >> x;

int head = check(i);

int end = check(x);

if (head != end)

{

disjoint\_set[end] = head;

}

i++;

} while (i <= n);

i = 1;

do {

if (disjoint\_set[i] == i)

{

++answer;

}

i++;

} while (i <= n);

cout << answer;

return 0;

}

# O. Долой списывание!

Create an adjacency list lnk to store the edge relationship of the graph.

Create a part vector to record the color of each node, the initial value is 0 (WHITE).

Create a visited vector to mark whether the node has been visited, the initial value is false.

Traverse the input edges and add the edge relationship to the adjacency list lnk.

Initialize a variable i to 0, indicating the current starting node for DFS.

Loop until all nodes are traversed:

If node i has not been visited, and the DFS starting from node i returns false (indicating that it is not a bipartite graph), then output "NO" and end the program.

Otherwise, increase i by 1 and continue to judge the next node.

If all nodes meet the conditions of the bipartite graph, the output "YES" indicates that it is a bipartite graph.

In this algorithm, the nodes of the graph are traversed through DFS, and a color (1 or 2) is assigned to each node, and the nodes adjacent to the current node are marked with a different color from the current node. If it is found that the adjacent nodes of a certain node have the same color as it in the DFS process, it means that the graph is not a bipartite graph.

Code：

#include <iostream>

#include <vector>

#include <algorithm>

#include <cassert>

using namespace std;

const int WHITE = 0;

const int GRAY = 1;

const int BLACK = 2;

int n, m;

vector<vector<int>> lnk;

vector<int> part;

vector<bool> visited;

bool dfs(int v, int color)

{

visited[v] = true;

part[v] = color;

for (int next : lnk[v])

{

if (part[next] == color)

return false;

}

for (int next : lnk[v])

{

if (!visited[next] && !dfs(next, 3 - color))

return false;

}

return true;

}

int main()

{

cin >> n >> m;

lnk.resize(n);

part.resize(n, WHITE);

visited.resize(n, false);

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

{

int u, v;

cin >> u >> v;

u--;

v--;

lnk[u].push\_back(v);

lnk[v].push\_back(u);

}

int i = 0;

do {

if (!visited[i] && !dfs(i, GRAY))

{

cout << "NO" << endl;

return 0;

}

i++;

} while (i < n);

cout << "YES" << endl;

return 0;

}

# Авиаперелёты

Through binary search and depth-first search, the scope of connection distance is continuously narrowed, and finally the minimum connection distance that meets the conditions is found.

Read the input city number n, and initialize some data structures, such as vector and bool type arrays.

Use nested loops to read the connections between cities and store them in the corresponding data structures.

Perform a binary search, set the minimum value mini to 0, and the maximum value maxi to 2147483647 (the maximum value of the int type).

In each cycle of binary search, the current intermediate value mid is calculated as (mini+maxi)/2.

Update the connection status array g according to the median value mid, set the connection distance between cities less than or equal to mid to true, and set it to false if the distance between cities is greater than mid.

Use the depth-first search (DFS) algorithm to traverse the connection array g starting from city 0, and mark the visited cities as visited.

Check whether all cities have been visited, if so, it means that the connectivity of all cities can be achieved under the current median value mid, and update maxi to mid.

If not all cities have been visited, it means that the connectivity of all cities cannot be achieved under the current median value mid, and mini is updated to mid+1.

Output the final mini value, which is the minimum connection distance that satisfies the condition.

#include <iostream>

#include <vector>

using namespace std;

int n;

vector<vector<int>> cities;

vector<bool> visited;

vector<vector<bool>> g;

int mini = 0;

int maxi = 2147483647;

void dfs(int d,int i){

visited[d] = true;

int j = 0;

do {

bool enter = false;

if (i == 1) {

enter = g[j][d] && !visited[j];

} else {

enter = g[d][j] && !visited[j];

}

if (enter) {

dfs(j, i);

}

j++;

} while (j < n);

}

bool all\_visited(){

int i = 0;

do {

if (!visited[i]) {

return false;

}

i++;

} while (i < n);

return true;

}

int main()

{

cin >> n;

cities.resize(n);

visited.resize(n);

g.resize(n);

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

visited[i] = false;

}

int i = 0;

do {

vector<int> v(n);

vector<bool> b(n);

int j = 0;

do {

int k;

cin >> k;

v[j] = k;

b[j] = false;

j++;

} while (j < n);

g[i] = b;

cities[i] = v;

i++;

} while (i < n);

while(mini<maxi){

int mid = (mini+maxi)/2;

int i = 0;

do {

int j = 0;

do {

g[i][j] = cities[i][j] <= mid;

j++;

} while (j < n);

i++;

} while (i < n);

fill(visited.begin(),visited.end(),false);

dfs(0,0);

bool b = false;

if(all\_visited()){

fill(visited.begin(),visited.end(),false);

dfs(0,1);

if(!all\_visited()){

b = true;

}

}else{

b =true;

}

if(!b){

maxi = mid;

}else{

mini = mid + 1;

}

}

cout<<mini<<endl;

return 0;

}