#include <cstdio>

#include <iostream>

#include <string>

#include <vector>

#include <set>

#include <deque>

#include <algorithm>

#include <map>

#include <cstring>

using namespace std;

#define FOR(i,a,b) for (int i = (a); i < (b); i++)

#define FORIT(i,c) for (\_\_typeof\_\_((c).begin()) i = (c).begin(); i != (c).end(); i++)

#define FORITR(i,c) for (\_\_typeof\_\_((c).begin()) i = ((c).end() - 1); i >= (c).begin(); i--)

#define MAX 20

//normal

vector<int> adj[MAX][2];

int n, m;

// Topsort

vector<int> ts\_list;

int ts\_state[MAX];

int ts\_dist[MAX];

bool topsort\_loop;

void topsort\_dfs(int current){

    if (ts\_state[current] == 1) topsort\_loop = true;

    if (ts\_state[current]) return;

    ts\_state[current] = 1;

    FORIT(i,adj[current][0]) topsort\_dfs(\*i);

    ts\_state[current] = 2;

    ts\_list.push\_back(current);

}

void topsort(){

    topsort\_loop = false;

    ts\_list.clear();

    memset(ts\_state, 0, sizeof(ts\_state));

    FOR(i,0,n) topsort\_dfs(i);

    reverse(ts\_list.begin(), ts\_list.end());

    //Print out Result

    cout << "topsort: ";

    FORIT(i,ts\_list) cout << " " << \*i + 1;

    cout << endl;

    cout << (topsort\_loop ? "has loop":"no loop") << endl;

}

// DAG - Algorithm

void dag\_short\_paths(int b, int e){

    // find the shorted path between [b]egin and [e]nd

    memset(ts\_state, 0, sizeof(ts\_state));

    memset(ts\_dist, 0, sizeof(ts\_dist));

    // fill points with orders

    int ts\_order[MAX];

    int c = 0;

    FORIT(i,ts\_list) ts\_order[\*i] = c++;

    // find the starting point

    // In this example, all the path length is 1

    ts\_state[ts\_order[b]] = 1; // mark starting point valid.

    cout << "s:" << ts\_order[b] << "\tt:" << ts\_order[e] << endl;

    FOR(i,ts\_order[b],ts\_order[e]) FORIT(j,adj[ts\_list[i]][0]) if (ts\_state[ts\_order[\*j]]){

        int p = 1;

        if (ts\_dist[ts\_order[\*j]] > ts\_dist[i] + p) ts\_dist[ts\_order[\*j]] = ts\_dist[i] + p;

    }else{

        ts\_state[ts\_order[\*j]] = 1;

        int p = 1;

        ts\_dist[ts\_order[\*j]] = ts\_dist[i] + p;

    }

    //print result

    cout << "Has path:" << ts\_state[ts\_order[e]] << endl;

    cout << "length  :" << ts\_dist[ts\_order[e]] << endl;

    cout << "DAG-SP: ";

    FOR(i,0,n) cout << " " << ts\_dist[i];

}

// scc id from 0..N

int main(){

    cin >> n >> m;

    FOR(i,0,n){

        adj[i][0].clear();

        adj[i][1].clear();

    }

    FOR(i,0,m){

        int f,t;

        cin >> f >> t;

        f--;t--;

        adj[f][0].push\_back(t);

        adj[t][1].push\_back(f);

    }

    FOR(i,0,n){

        cout << i + 1 << " to:";

        FORIT(j,adj[i][0]) cout << \*j + 1<< "\t";

        cout << "\tfrom:";

        FORIT(j,adj[i][1]) cout << \*j + 1<< "\t";

        cout << endl;

    }

    //Topsot

    topsort();

    dag\_short\_paths(\_\_\_,\_\_\_);

    return 0;

}

//normal

vector<int> adj[MAX][2];

int n, m;

//kosaraju

// scc id from 0..N

int scc\_n, scc\_list[MAX];

vector<int> scc\_stack;

set<int> scc\_group[MAX];

void kosaraju\_first\_dfs(int node){

    if (scc\_list[node] != -1) return;

    scc\_list[node] = 0;

    FORIT(i,adj[node][0]) kosaraju\_first\_dfs(\*i);

    scc\_stack.push\_back(node);

}

void kosaraju\_second\_dfs(int node, int scc\_id){

    if (scc\_list[node] != -1) return;

    scc\_list[node] = scc\_id;

    FORIT(i,adj[node][1]) kosaraju\_second\_dfs(\*i, scc\_id);

}

void kosaraju(){

    memset(scc\_list, -1, sizeof(scc\_list));

    scc\_stack.clear();

    FOR(i,0,n) kosaraju\_first\_dfs(i);

    scc\_n = 0;

    memset(scc\_list, -1, sizeof(scc\_list));

    reverse(scc\_stack.begin(),scc\_stack.end());

    FORIT(i,scc\_stack) if (scc\_list[\*i] == -1) kosaraju\_second\_dfs(\*i,scc\_n ++);

    // TEST PRINT OUT

    //FOR(i,0,n) cout << i+1 << " : SCC\_ID" << scc\_list[i] << endl;

}

//Dynamic Programming

int dp\_length[MAX];

int dp\_result[MAX];

void dp\_fill\_distance\_dfs(int node, set<int> next, int distance){

    if (distance > dp\_length[node]) dp\_length[node] = distance;

    next.erase(node);

    FORIT(j,adj[node][0]) if (next.count(\*j)) dp\_fill\_distance\_dfs(\*j,next,distance+1);

}

int main(){

    cin >> n >> m;

    FOR(i,0,n){

        adj[i][0].clear();

        adj[i][1].clear();

    }

    FOR(i,0,m){

        int f,t;

        cin >> f >> t;

        f--;t--;

        adj[f][0].push\_back(t);

        adj[t][1].push\_back(f);

    }

    /\*

     FOR(i,0,n){

     cout << i + 1 << " to:";

     FORIT(j,adj[i][0]) cout << \*j + 1<< "\t";

     cout << "\tfrom:";

     FORIT(j,adj[i][1]) cout << \*j + 1<< "\t";

     cout << endl;

     }

     \*/

    // Finding Strongly Connected Components

    kosaraju();

    // Clean up a little bit scc

    // scc\_group[0] stores all the isolated nodes.

    FOR(i,0,scc\_n){

        scc\_group[i].clear();

    }

    FOR(i,0,n){

        scc\_group[scc\_list[i]].insert(i);

    }

    //Print out scc group

    /\*

     FOR(i,0,scc\_n){

     cout << "group id:" << i << "\t";

     FORIT(j,scc\_group[i]) cout << \*j + 1 << "\t";

     cout << endl;

     }

     \*/

    // ------------- Dynamic Programming -------------

    memset(dp\_result, 0 ,sizeof(dp\_result));

    // Finding Result "longest path"

    int max\_length = 0;

    FOR(i, 0, scc\_n){

        FORIT(p,scc\_group[i]){

            //loose

            FORIT(j,scc\_group[i]) dp\_length[\*j] = -1;

            //DFS from the node \*p

            dp\_fill\_distance\_dfs(\*p,scc\_group[i],1);

            int length = 0;

            //From outside of the SCC

            FORIT(j,adj[\*p][1]) if((i != scc\_list[\*j]) && (dp\_result[\*j] > length)) {

                length = dp\_result[\*j];

            }

            FORIT(j,scc\_group[i]) if (dp\_result[\*j] < (length + dp\_length[\*j])) {

                dp\_result[\*j] = (length + dp\_length[\*j]);

                if (dp\_result[\*j] > max\_length) {

                    max\_length = dp\_result[\*j];

                }

            }

        }

    }

    cout << max\_length << endl;

    return 0;

}