//کمر گراف

O(V∗(V+E))

مسئله را می‌توانیم به چند حالت تقسیم کنیم:

* گراف‌های جهت‌دار: در این‌جا پیدا کردن دور کمی متفاوت است و هر دوری حتما در یک [مؤلفه قویا همبند](http://opedia.ir/%D8%A2%D9%85%D9%88%D8%B2%D8%B4/%D8%A7%D9%84%DA%AF%D9%88%D8%B1%DB%8C%D8%AA%D9%85/%D9%BE%DB%8C%D8%AF%D8%A7_%DA%A9%D8%B1%D8%AF%D9%86_%D9%85%D8%A4%D9%84%D9%81%D9%87%E2%80%8C%E2%80%8C%D9%87%D8%A7%DB%8C_%D9%87%D9%85%D8%A8%D9%86%D8%AF) است. پس ابتدا باید مؤلفه‌های قویا همبند را پیدا کرد و سپس برای هر مؤلفه کمر را پیدا کرد.
* گراف‌های وزن‌دار: در این حالت تعریف طول دور تعداد یال‌ها نیست بلکه وزن یال‌های آن دور است؛ اگر گراف یال با وزن منفی داشته باشد، چه بسا ممکن است دوری با طول منفی پیدا کنیم و هرچه روی آن حرکت کنیم طول آن کمتر شود! در این حالت [روشی برای پیدا کردن دور منفی](http://opedia.ir/%D8%A2%D9%85%D9%88%D8%B2%D8%B4/%D8%A7%D9%84%DA%AF%D9%88%D8%B1%DB%8C%D8%AA%D9%85%E2%80%8C%D9%87%D8%A7%DB%8C_%D8%AA%DA%A9%D9%85%DB%8C%D9%84%DB%8C/%D9%85%D9%81%D8%A7%D9%87%DB%8C%D9%85_%DA%A9%D9%88%D8%AA%D8%A7%D9%87%D8%AA%D8%B1%DB%8C%D9%86_%DA%AF%D8%B4%D8%AA_%D9%88_%D8%AF%D9%88%D8%B1_%D9%85%D9%86%D9%81%DB%8C) وجود دارد. اگر دور منفی نداشته باشیم نیز باید از الگوریتم هایی که روی گراف‌های وزن‌دار اجرا می‌شوند استفاده کرد مانند [فلوید](http://opedia.ir/%D8%A2%D9%85%D9%88%D8%B2%D8%B4/%D8%A7%D9%84%DA%AF%D9%88%D8%B1%DB%8C%D8%AA%D9%85/%D8%A7%D9%84%DA%AF%D9%88%D8%B1%DB%8C%D8%AA%D9%85_%D9%81%D9%84%D9%88%DB%8C%D8%AF-%D9%88%D8%A7%D8%B1%D8%B4%D8%A7%D9%84) که همه کوتاه‌ترین مسیر‌ها (از جمله کوتاه‌ترین مسیر هر رأس به خودش) را پیدا می‌کند.
* گراف‌های بدون‌جهت و بی‌وزن: در اینجا به بررسی این حالت می‌پردازیم و با استفاده از [جست‌و‌جوی سطح‌اول](http://opedia.ir/%D8%A2%D9%85%D9%88%D8%B2%D8%B4/%D8%A7%D9%84%DA%AF%D9%88%D8%B1%DB%8C%D8%AA%D9%85/%D8%AC%D8%B3%D8%AA%E2%80%8C%D9%88%D8%AC%D9%88%DB%8C_%D8%B3%D8%B7%D8%AD%E2%80%8C%D8%A7%D9%88%D9%84) مسئله را حل می‌کنیم.

توجه کنید که اگر در مرحله‌ای طول دور کمینه پیدا شده برابر dd باشد، آنگاه اگر در پیمایشمان به ارتفاع بیش از d/2d/2 برسیم پس حتما دوری که پیدا می‌شود بیشتر از dd می‌شود بنابراین کمر از این رأس نمی‌گذرد و پیمایش را متوقف کرده و از رأس بعدی آغاز می‌کنیم.

به ازای هر رأس مانند vv [جست‌و‌جوی سطح‌اول](http://opedia.ir/%D8%A2%D9%85%D9%88%D8%B2%D8%B4/%D8%A7%D9%84%DA%AF%D9%88%D8%B1%DB%8C%D8%AA%D9%85/%D8%AC%D8%B3%D8%AA%E2%80%8C%D9%88%D8%AC%D9%88%DB%8C_%D8%B3%D8%B7%D8%AD%E2%80%8C%D8%A7%D9%88%D9%84) می‌زنیم.   
در هر پیمایش، رأس‌های دیده شده را علامت‌گذاری می‌کنیم. اولین باری که به رأسی علامت‌خورده رسیدیم، یعنی قبلا با یک مسیر به آن رسیده بودیم و الان با یک مسیر دیگر رسیده ایم، پس حتما یک دور پیدا شده است. اندازه این دور برابر جمع طول این دو مسیر است. از آنجایی که طول مسیر اول و دوم برابر ارتفاع (سطح) این رأس و یا یک واحد بیشتر است، پس این طول به راحتی قابل محاسبه است. این مقدار را ذخیره و پیمایش بعدی را آغاز می‌کنیم.   
بین همه دور‌های پیدا شده، کمترین را به عنوان کمر گزارش می‌کنیم.

#include <queue>

#include <vector>

#include <cstring>

#include <iostream>

using namespace std;

const int MAXN = 100 \* 1000 + 10;

const int INF = 1<<30; // بینهایت

bool mark[MAXN];

vector<int> adj[MAXN]; // لیست مجاورت رأس‌ها

int depth[MAXN]; // ارتفاع هر رأس

int parent[MAXN]; // پدر هر رأس

int n; // تعداد رأس ها توجه کنید شماره رأس‌ها از ۰ شروع می‌شود

int m; // تعداد یال‌ها

int bfs(int v, int max\_depth) {

queue<int> q; // صفی که رأس‌ها در آن قرار می‌گیریند

memset(depth, -1, sizeof(depth));

memset(mark, 0, sizeof(mark));

mark[v] = 1;

depth[v] = 0;

q.push(v);

while(q.size()) {

v = q.front();

q.pop();

if (depth[v] == max\_depth) // دوری که در آینده پیدا می‌شود نمی‌تواند کمینه باشد

return INF;

for(unsigned int i = 0; i < adj[v].size(); i++) {

int u = adj[v][i];

if (mark[u] == 1 && u != parent[v]) // پس دور پیدا شده است

return (depth[v]+1) + depth[u]; // ارتفاع قبلی پیدا شده برای این رأس + ارتفاع الان پیدا

mark[u] = 1;

q.push(u);

parent[u] = v;

depth[u] = depth[v] + 1; // ارتفاع همسایه یک واحد بیشتر از ارتفاع این رأس است

}

}

return INF;

}

int find\_min\_cycle()

{

int dist = INF; // تقریبا بینهایت برای اینکه در کمینه تاثیری نگذارد و دور را نگه می‌دارد

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

dist = min(dist, bfs(i, dist/2));

return dist;

}

void input()

{

cin >> n >> m;

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

int v, u;

cin >> v >> u;

adj[--v].push\_back(--u);

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

}

}

int main()

{

input();

cout << find\_min\_cycle() << endl;

}

------------------------------------------------------------------------------------------------------------------

# **Detect Cycle in a Directed Graph**

bool Graph::isCyclicUtil(int v, bool visited[], bool \*recStack)

{

    if(visited[v] == false)

    {

       visited[v] = true;

        recStack[v] = true;

        // Recur for all the vertices adjacent to this vertex

        list<int>::iterator i;

        for(i = adj[v].begin(); i != adj[v].end(); ++i)

        {

            if ( !visited[\*i] && isCyclicUtil(\*i, visited, recStack) )

                return true;

            else if (recStack[\*i])

                return true;

        }

    }

    recStack[v] = false;  // remove the vertex from recursion stack

    return false;

}

 bool Graph::isCyclic()

{

    // Mark all the vertices as not visited and not part of recursion

    // stack

    bool \*visited = new bool[V];

    bool \*recStack = new bool[V];

    for(int i = 0; i < V; i++)

    {

        visited[i] = false;

        recStack[i] = false;

    }

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

        if (isCyclicUtil(i, visited, recStack))

            return true;

    return false;

}

----------------------------------------------------------------------------------------------------

# **Detect cycle in an undirected graph**

bool Graph::isCyclicUtil(int v, bool visited[], int parent)

{

    visited[v] = true;

    list<int>::iterator i;

    for (i = adj[v].begin(); i != adj[v].end(); ++i)

    {

        // If an adjacent is not visited, then recur for that adjacent

        if (!visited[\*i])

        {

           if (isCyclicUtil(\*i, visited, v))

              return true;

        }

        else if (\*i != parent)

           return true;

    }

    return false;

}

bool Graph::isCyclic()

{

    bool \*visited = new bool[V];

    for (int i = 0; i < V; i++)

        visited[i] = false;

    for (int u = 0; u < V; u++)

        if (!visited[u]) // Don't recur for u if it is already visited

          if (isCyclicUtil(u, visited, -1))

             return true;

    return false;

}