# CHUYÊN ĐỀ: LÝ THUYẾT ĐỒ THỊ

Đặng Phúc An Khang*

Ngày 18 tháng 7 năm 2025

**Tóm tắt nội dung**

**Code:**

- C/C++: https://github.com/GrootTheDeveloper/OLP-ICPC/tree/master/2025/C%2B%2B.
- Python:

**Tài khoản trên các Online Judge:**

- Codeforces: https://codeforces.com/profile/vuivethoima.
- VNOI: oj.vnoi.info/user/Groot.
- IUHCoder: oj.iuhcoder.com/user/ankhang2111.
- MarisaOJ: https://marisaoj.com/user/grootsiuvip/submissions.
- CSES: https://cses.fi/user/212174.
- UMTOJ: sot.umtoj.edu.vn/user/grootsiuvip.
- SPOJ: www.spoj.com/users/grootsiuvip/.
- POJ: http://poj.org/userstatus?user_id=vuivethoima.
- ATCoder: https://atcoder.jp/users/grootsiuvip
- OnlineJudge.org: vuivethoima
- updating...

## Mục lục



*E-mail: ankhangluonvuituoi@gmail.com. Tây Ninh, Việt Nam.

# 1 Preliminaries – Kiến thức chuẩn bị

**Resources – Tài nguyên.**

1. [**CP10**]. *CP10. Competitive Programming* https://drive.google.com/drive/folders/1MTEVHT-7nBnMJ7C9LgyAR_pEVSE3FlKz?fbclid=IwAR3TovIj2rKCRe1a4oZxW-LQCoEoVkipVAvCzwrr0nJ6GzcAd47P6LO1Rwc

2. [**cp-algorithms**]. *Algorithms for Competitive Programming* https://cp-algorithms.com

3. [**VNOI-WIKI**]. *Thư viện VNOI* https://wiki.vnoi.info

# 2 Kiến thức

## 2.1 Giới thiệu về đồ thị và thuật toán DFS

### 2.1.1 Lý thuyết đồ thị là gì?

**Định nghĩa 1.** *Lý thuyết đồ thị là một nhánh của toán học, cụ thể thuộc toán rời rạc. Lý thuyết đồ thị chuyên nghiên cứu các bài toán liên quan đến việc biểu diễn và phân tích các sự vật, hiện tượng hoặc trạng thái có mối quan hệ lẫn nhau thông qua mô hình đồ thị.*

**Ví dụ 1.** *Mạng lưới giao thông, cây phả hệ (cây gia phả), mạng máy tính, sơ đồ tổ chức, v.v.*

### 2.1.2 Một số khái niệm căn bản trong lý thuyết đồ thị

1. **Đỉnh:** Được biểu diễn nhằm mục đích thể hiện sự vật, sự việc hay một trạng thái.

2. **Cạnh:** Biểu diễn cho mối quan hệ giữa 2 đỉnh với nhau. **Lưu ý:** Giữa 2 đỉnh trong đồ thị có thể có cạnh, không có, hoặc có thể có nhiều cạnh với nhau. Cạnh được chia thành 2 dạng:

   (a) **Cạnh vô hướng:** Nếu một cạnh vô hướng nối 2 đỉnh $u$ và $v$, thì $u$ có thể đến $v$ trực tiếp và ngược lại.

   (b) **Cạnh có hướng:** Nếu một cạnh có hướng nối từ đỉnh $u$ đến đỉnh $v$, thì ta có thể đi trực tiếp từ $u$ đến $v$, nhưng không thể đi ngược lại từ $v$ đến $u$ trừ khi có một cạnh khác từ $v$ đến $u$.

3. **Đường đi:** Một đường đi là một danh sách các đỉnh $x_1, x_2, x_3, x_4, \ldots, x_k$. Trong đó 2 đỉnh $x_i$ và $x_{i+1}$ thì có một đường nối trực tiếp để đi từ $x_i \to x_{i+1}$.

4. **Trọng số:** Là một giá trị trên cạnh (hoặc trên đỉnh) nhằm thể hiện một thông số nào đó với bài toán ta đang xét.

### 2.1.3 Danh sách kề

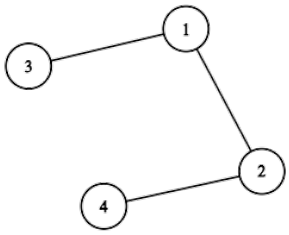
Một trong những cách phổ biến để biểu diễn đồ thị là sử dụng **danh sách kề (adjacency list)**. Cách biểu diễn này đặc biệt hiệu quả đối với đồ thị thưa (sparse graph).
Cụ thể, ta sử dụng cấu trúc dữ liệu `vector<int> adj[u]` trong C++, trong đó:

- Mỗi phần tử `adj[u]` là một vector chứa các đỉnh kề với đỉnh $u$.
- Nghĩa là nếu có cạnh nối từ đỉnh $u$ đến đỉnh $v$ thì $v$ sẽ xuất hiện trong `adj[u]`.

- Đối với đồ thị vô hướng, nếu có cạnh giữa $u$ và $v$ thì cả $v \in \texttt{adj}[u]$ và $u \in \texttt{adj}[v]$.

**Ví dụ:**
Giả sử đồ thị vô hướng có các cạnh: $(1,2)$, $(1,3)$, $(2,4)$ thì danh sách kề sẽ là:

$$\texttt{adj[1]} = \{2,3\}$$
$$\texttt{adj[2]} = \{1,4\}$$
$$\texttt{adj[3]} = \{1\}$$
$$\texttt{adj[4]} = \{2\}$$




Hình 1: Minh họa danh sách kề của đồ thị vô hướng vừa mô tả

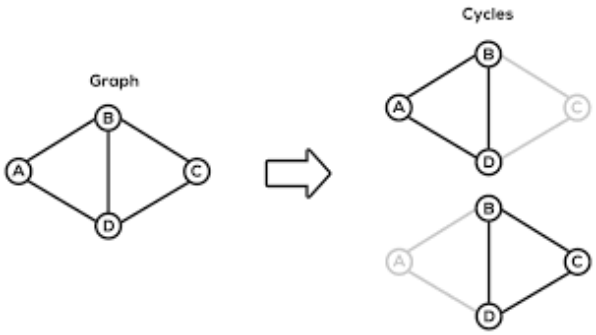
Cách biểu diễn này có độ phức tạp về bộ nhớ là $\mathcal{O}(n+m)$, với $n$ là số đỉnh và $m$ là số cạnh.

### 2.1.4 Chu trình (Cycle)

**Định nghĩa 2.** *Một **chu trình** là một đường đi bắt đầu và kết thúc tại cùng một đỉnh, trong đó không có đỉnh nào khác (ngoại trừ đỉnh đầu/cuối) được lặp lại.*

- Trong đồ thị vô hướng: chu trình là dãy đỉnh $v_1 \to v_2 \to \cdots \to v_k \to v_1$ với $k \geq 3$.
- Trong đồ thị có hướng: các cung phải có hướng phù hợp với trình tự chu trình.

Một chu trình được gọi là *chu trình đơn giản* nếu không có cạnh hoặc đỉnh nào bị lặp lại (trừ đỉnh đầu/cuối).

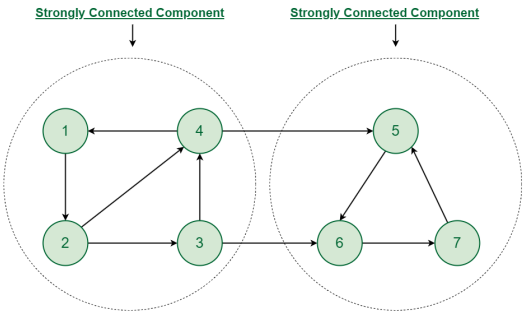



Hình 2: Minh họa chu trình của đồ thị vô hướng

### 2.1.5 Thành phần liên thông (Connected Component)

**Định nghĩa 3.** *Một **thành phần liên thông** là một tập con các đỉnh sao cho giữa mọi cặp đỉnh trong đó đều tồn tại một đường đi.*

- Với đồ thị vô hướng: liên thông nếu có đường đi giữa mọi cặp đỉnh.
- Với đồ thị có hướng:
  - *Liên thông mạnh* nếu tồn tại đường đi theo chiều từ mọi đỉnh đến mọi đỉnh khác.
  - *Liên thông yếu* nếu bỏ hướng trên các cạnh thì đồ thị trở nên liên thông.




Hình 3: Minh họa thành phần liên thông mạnh của đồ thị có hướng

### 2.1.6 Bậc của đỉnh (Degree)

- Trong đồ thị vô hướng, bậc của một đỉnh là số cạnh nối với nó.
- Trong đồ thị có hướng:
  - *Bậc vào* (in-degree): số cung đi vào đỉnh.
  - *Bậc ra* (out-degree): số cung đi ra từ đỉnh.

### 2.1.7 Đường đi đơn giản (Simple Path)

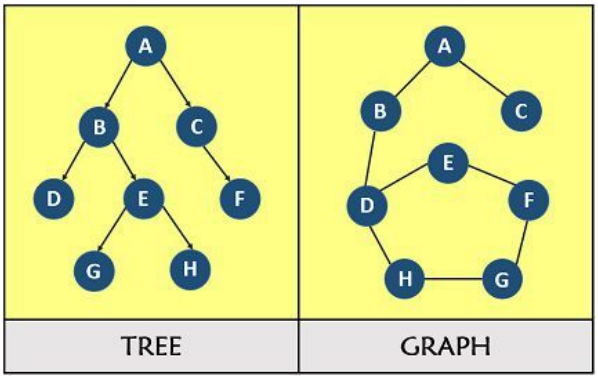
Một **đường đi đơn giản** là đường đi không đi qua một đỉnh nào hai lần (trừ khi là chu trình).

### 2.1.8 Cây (Tree)

**Định nghĩa 4.** *Một **cây** là một đồ thị vô hướng liên thông và không có chu trình.*

Tính chất quan trọng của cây:

- Với $n$ đỉnh, cây có đúng $n - 1$ cạnh.
- Có duy nhất một đường đi giữa hai đỉnh bất kỳ.

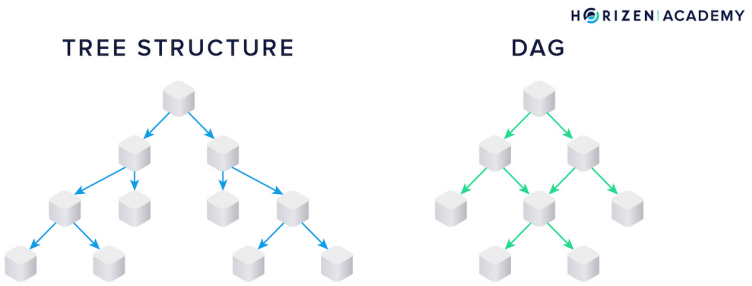



Hình 4: Phân biệt Cây và Đồ thị (không phải cây)

### 2.1.9 Đồ thị vô chu trình (Acyclic Graph).

Đồ thị gọi là **vô chu trình** nếu không tồn tại chu trình nào trong nó.

- Với đồ thị **vô hướng**, một đồ thị được xem là **Acyclic Graph** nếu không tồn tại dãy các đỉnh $v_1 \to v_2 \to \cdots \to v_k \to v_1$ với $k \geq 3$ và các cạnh liên tiếp nối các đỉnh đó.
- Một đồ thị vô hướng đơn giản gồm hai đỉnh được nối với nhau bằng một cạnh cũng là một **Acyclic Graph**, vì không tồn tại chu trình nào (phải có ít nhất 3 đỉnh để hình thành chu trình trong đồ thị vô hướng).
- Đồ thị có hướng vô chu trình gọi là **DAG** (Directed Acyclic Graph). Nghĩa là một đồ thị có hướng không chứa bất kỳ chu trình nào tuân theo chiều các cung.
- Cây là một DAG, nhưng không phải tất cả DAG đều là cây.



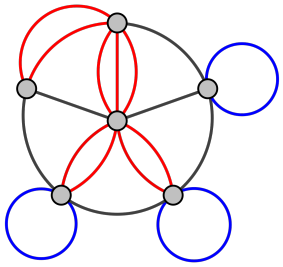

Hình 5: Phân biệt Cây và DAG

#### 2.1.10 Đồ thị đơn (Simple Graph).

Đồ thị đơn là đồ thị không có *cạnh lặp* giữa cùng một cặp đỉnh và không có *khuyên* (loop – cạnh nối đỉnh với chính nó).

#### 2.1.11 Đa đồ thị (Multigraph).

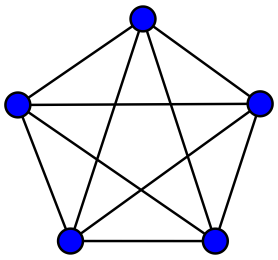
Đa đồ thị cho phép tồn tại *nhiều cạnh song song* giữa hai đỉnh và/hoặc khuyên. Thường dùng để mô hình hoá các mạng có nhiều kênh kết nối.



Hình 6: Minh họa Đa đồ thị

#### 2.1.12 Đồ thị đầy đủ (Complete Graph).

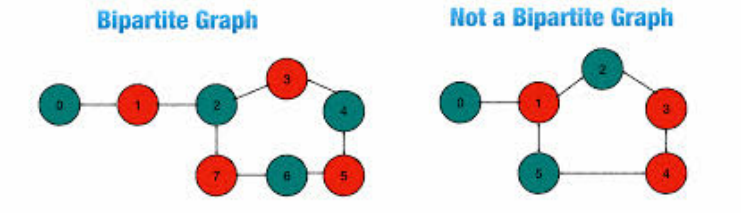
Đồ thị vô hướng $K_n$ có $n$ đỉnh, trong đó mọi cặp đỉnh phân biệt đều được nối bởi một cạnh.
Số cạnh là $\dfrac{n(n-1)}{2}$.
Đồ thị có hướng đầy đủ có $n(n-1)$ cung.



Hình 7: Minh họa đồ thị đầy đủ

#### 2.1.13 Đồ thị hai phía (Bipartite Graph).

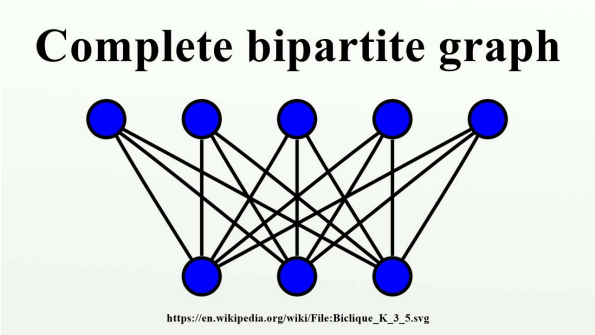
Một **đồ thị hai phía** là đồ thị mà tập đỉnh có thể chia thành hai tập rời $U$ và $V$ sao cho mọi cạnh đều nối một đỉnh từ $U$ đến một đỉnh từ $V$.




Hình 8: Phân biệt đồ thị hai phía

#### 2.1.14 Đồ thị hai phía đầy đủ (Complete Bipartite Graph).

Cho hai tập đỉnh rời $U$ và $V$ với $|U| = m$, $|V| = n$. Đồ thị $K_{m,n}$ chứa mọi cạnh nối một đỉnh của $U$ với một đỉnh của $V$ và *không có cạnh nội bộ* trong $U$ hay $V$. Tổng số cạnh: $m\,n$.



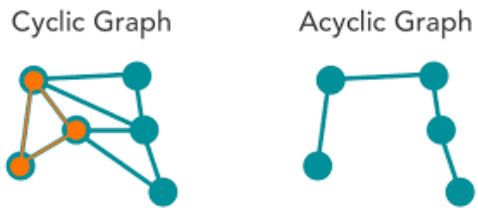

Hình 9: Minh họa đồ thị hai phía đầy đủ

#### 2.1.15 Đồ thị vòng (Cycle Graph)

Đồ thị vòng ký hiệu $C_n$ ($n \geq 3$) là đồ thị vô hướng gồm $n$ đỉnh $\{v_1, v_2, \ldots, v_n\}$ và $n$ cạnh

$$E = \big\{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\big\}.$$

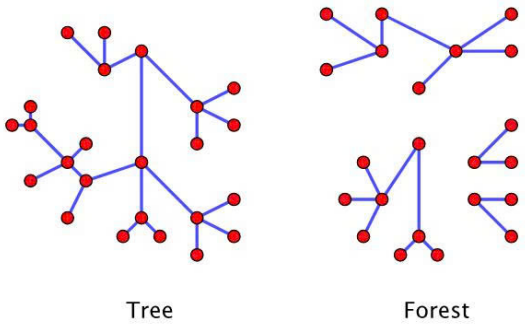
- Mỗi đỉnh có bậc 2 – $C_n$ là đồ thị 2-chính quy.
- $C_n$ chứa đúng một chu trình đơn giản độ dài $n$.
- $C_n$ là **bipartite** khi và chỉ khi $n$ chẵn.
- Số cạnh $m = n$; số đỉnh $n$; đường kính (diameter) $\operatorname{diam}(C_n) = \lfloor n/2 \rfloor$.
- Tồn tại cây khung nhỏ nhất với trọng số tổng bằng $n - 1$ nếu mọi cạnh có trọng số 1.




Hình 10: Minh họa CG và AG

#### 2.1.16 Rừng (Forest).

Một **rừng** là đồ thị vô hướng *không chứa chu trình* nhưng *không nhất thiết liên thông*. Mỗi thành phần liên thông của rừng là một **cây**. Nếu rừng có $c$ cây và $n$ đỉnh, nó có đúng $n - c$ cạnh.




Hình 11: Tree vs Forest

## 2.2 Thuật toán DFS

Thuật toán DFS (Depth-First Search – Duyệt theo chiều sâu) là một trong những thuật toán cơ bản để duyệt hoặc tìm kiếm trên đồ thị. Ý tưởng chính là xuất phát từ một đỉnh ban đầu, đi sâu theo từng nhánh con của đồ thị cho đến khi không còn đỉnh nào có thể đi tiếp, sau đó quay lui để khám phá các nhánh khác.
DFS có thể được cài đặt đệ quy hoặc sử dụng ngăn xếp. Nó thường được dùng để:

- Kiểm tra tính liên thông của đồ thị
- Tìm thành phần liên thông
- Phát hiện chu trình
- Tìm đường đi trong mê cung hoặc đồ thị

### 2.2.1 Ý tưởng cài đặt thuật toán DFS

DFS thường được cài đặt bằng đệ quy hoặc sử dụng ngăn xếp. Trong cài đặt đệ quy, ta cần một mảng đánh dấu để theo dõi các đỉnh đã được thăm nhằm tránh lặp vô hạn trong trường hợp đồ thị có chu trình.
**Các bước cơ bản trong cài đặt DFS đệ quy:**

1. Khởi tạo một mảng `visited[]` để đánh dấu các đỉnh đã được duyệt, với ý nghĩa: `visited[u] = true / false` nếu đỉnh $u$ đã thăm / chưa thăm.
2. Gọi hàm `DFS(u)` tại đỉnh bắt đầu $u$.
3. Trong mỗi lần gọi:
   - Đánh dấu `visited[u] = true`.
   - Duyệt qua tất cả các đỉnh kề $v$ của $u$:
     - Nếu $v$ chưa được thăm (`visited[v] == false`), đệ quy gọi `DFS(v)`.

### 2.2.2 Cài đặt DFS sử dụng đệ quy trong C++

Listing 1: Thuật toán DFS sử dụng đệ quy

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100005; // So dinh toi da
vector<int> adj[MAXN];   // Danh sach ke
bool visited[MAXN];      // Mang danh dau

void DFS(int u) {
    visited[u] = true;
    cout << "Tham dinh: " << u << endl;
    for (int v : adj[u]) {
        if (!visited[v]) {
            DFS(v);
        }
    }
}
int main() {
    int n, m; // so dinh va so canh
```

```
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u); // Neu la do thi vo huong
    }
    for (int i = 1; i <= n; i++) {
        visited[i] = false;
    }
    // Goi DFS tu dinh 1 (hoac 1 dinh bat ky)
    DFS(1);

    return 0;
}
```

**Độ phức tạp:** $\mathcal{O}(V + E)$ với $V$ là số đỉnh, $E$ là số cạnh.

### 2.2.3 Bài tập

**Bài tập 1.** *MAKEMAZE*

**Đề bài**
Một mê cung hợp lệ là một mê cung có chính xác 1 lối vào và 1 lối ra và phải tồn tại ít nhất một đường đi thỏa mãn từ lối vào đến lối ra. Cho một mê cung, hãy chỉ ra rằng mê cung có hợp lệ hay không. Nếu có, in "valid", ngược lại in "invalid"
**Input**
Dòng đầu chứa một số nguyên $t$ ($1 \leq t \leq 10^4$) là số lượng test cases. Sau đó với mỗi test case, dòng đầu chứa 2 số nguyên $m$ ($1 \leq m \leq 20$) và $n$ ($1 \leq n \leq 20$), lần lượt là số lượng hàng và cột trong mê cung. Sau đó, là mê cung $M$ với kích thước $m$ x $n$. $M[i][j] = '\#'$ đại diện cho bức tường, $M[i][j] = '.'$ đại diện cho ô trống có thể đi vào được.
**Output**
Với mỗi test case, tìm xem mê cung tương ứng là "invalid" hay "valid"
**Example**

Listing 2: Input

```
1
4 4
####
#...
#.##
#.##
```

Listing 3: Output

```
valid
```

**Phân tích bài toán**
Vì mê cung chỉ có chính xác 1 lối vào và 1 lối ra. Trước hết ta cần kiểm tra biên ngoài của mê cung, nếu có chính xác 2 ô '.' thì có thể đó là một mê cung hợp lệ. Ngược lại (có ít hơn hoặc nhiều hơn 1 ô '.' ở biên) ta có thể khẳng định rằng đó không phải là một mê cung hợp lệ.

Khi ta đã có giả thuyết rằng mê cung là hợp lệ, ta cần 2 biến *start* và *target* lần lượt lưu lại tọa độ $(x, y)$ của 2 ô ngoài biên. Áp dụng thuật toán DFS tại ô *start*, sau khi DFS nếu *visited[target.first][target.second] = true* thì khẳng định rằng mê chung hợp lệ. Ngược lại, mê cung khônxg hợp lệ.

Listing 4: Cài đặt C++ bài MAKEMAZE

```
#include <bits/stdc++.h>
#define int long long
#define endl "\n"
using namespace std;

const int MAXN = 21;
bool visited[MAXN][MAXN];

int dx[4] = {1, -1, 0, 0};
int dy[4] = {0, 0, 1, -1};

void dfs(pair<int, int> start, const vector<vector<char>> &a, int m, int n) {
    auto [x, y] = start;
    visited[x][y] = true;
    for (int i = 0; i < 4; i++) {
        int new_x = dx[i] + x;
        int new_y = dy[i] + y;
        if (new_x >= 1 && new_x <= m && new_y >= 1 && new_y <= n &&
```

```
            !visited[new_x][new_y] && a[new_x][new_y] == '.') {
            dfs({new_x, new_y}, a, m, n);
        }
    }
}

signed main() {
    int t; cin >> t;
    while (t--) {
        int m, n; cin >> m >> n;
        vector<vector<char>> a(m + 1, vector<char>(n + 1));
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                cin >> a[i][j];
                visited[i][j] = false;
            }
        }

        pair<int, int> start = {-1, -1}, target = {-1, -1};
        int cnt = 0;

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if ((i == 1 || i == m || j == 1 || j == n) && a[i][j] == '.') {
                    if (start == make_pair(-1LL, -1LL)) start = {i, j};
                    else target = {i, j};
                    cnt++;
                }
            }
        }

        if (cnt != 2) {
            cout << "invalid" << endl;
            continue;
        } else {
            dfs(start, a, m, n);
            if (!visited[target.first][target.second]) cout << "invalid";
            else cout << "valid";
            cout << endl;
        }
    }
    return 0;
}
```

## 2.3 Sắp xếp tô-pô (Topological Sorting)

Nội dung bài chủ yếu tham khảo/copy từ [VNOI WIKI] : https://wiki.vnoi.info/algo/graph-theory/topological-sort.md

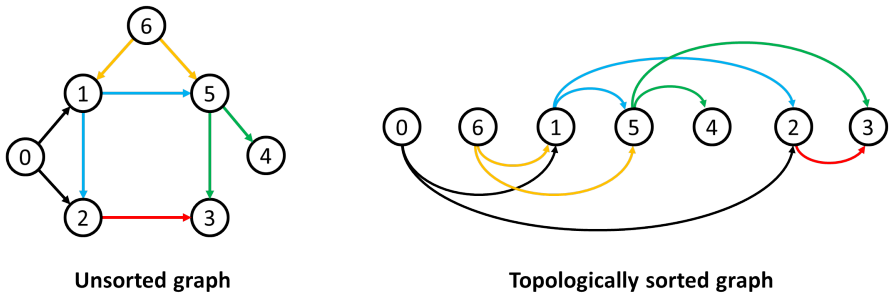
### 2.3.1 Định nghĩa

**Định nghĩa 5.** *Cho một đồ thị có hướng $G = (V, E)$, sắp xếp tô-pô (topological sorting) là một ánh xạ từ tập đỉnh $V$ vào tập các số nguyên $\{1, 2, \ldots, |V|\}$, sao cho với mọi cung $(u, v) \in E$, ta có:*

$$\text{order}(u) < \text{order}(v).$$

**Định nghĩa 6.** *Sắp xếp tô-pô là cách sắp xếp các đỉnh của một đồ thị có hướng sao cho nếu có một mũi tên từ đỉnh $u$ đến đỉnh $v$, thì $u$ phải đứng trước $v$ trong thứ tự đó.*

Nói cách khác, thứ tự tô-pô là một hoán vị của các đỉnh sao cho mọi cung đều đi từ đỉnh đứng trước đến đỉnh đứng sau trong thứ tự này. Hay nếu một công việc $u$ cần hoàn thành trước công việc $v$, thì $u$ phải xuất hiện trước $v$ trong danh sách kết quả.




Hình 12: Minh họa sắp xếp Tô-pô

### 2.3.2 Cảm hứng và Động cơ ứng dụng

Sắp xếp tô-pô là một công cụ quan trọng trong việc mô hình hóa và giải quyết các bài toán liên quan đến **phụ thuộc thứ tự** giữa các đối tượng. Về bản chất, nó cho phép ta xác định một trình tự thực hiện hợp lệ sao cho mọi điều kiện tiên quyết đều được thỏa mãn trước khi thực hiện bước tiếp theo.
Một ứng dụng thực tế điển hình là trong **lập kế hoạch công việc**. Khi một tập hợp các công việc có quan hệ phụ thuộc lẫn nhau, ta cần xác định thứ tự thực hiện sao cho mỗi công việc chỉ bắt đầu sau khi tất cả các công việc phụ thuộc của nó đã hoàn thành.
**Ví dụ minh họa:** Trong chương trình đào tạo đại học, sinh viên cần hoàn thành nhiều học phần để tốt nghiệp. Một số học phần là điều kiện tiên quyết cho các học phần khác. Chẳng hạn:

- Để học được môn "Giới thiệu về thuật toán", sinh viên phải hoàn thành các môn: "Nhập môn lập trình", "Cấu trúc dữ liệu", "Nhập môn thuật toán", v.v.

Ta có thể xây dựng một đồ thị có hướng, trong đó:

- Mỗi đỉnh tương ứng với một học phần;
- Có một cung từ đỉnh $u$ đến đỉnh $v$ nếu học phần $u$ là điều kiện tiên quyết của học phần $v$.

Khi đó, việc tìm một sắp xếp tô-pô của đồ thị này sẽ cho ta một thứ tự học hợp lệ. Nếu không tồn tại sắp xếp tô-pô (tức đồ thị có chu trình), điều đó phản ánh sự xung đột hoặc vòng lặp trong điều kiện tiên quyết giữa các môn học - một cấu trúc bất hợp lệ trong thiết kế chương trình đào tạo.

**Ghi chú 1.** *Sắp xếp tô-pô không xử lý các xung đột tài nguyên như trùng lịch học, mà chỉ đảm bảo mối quan hệ thứ tự phụ thuộc.*

**Ghi chú 2.** *Chỉ tồn tại sắp xếp tô-pô nếu và chỉ nếu đồ thị là DAG (Directed Acyclic Graph).*

### 2.3.3 Chứng minh điều kiện tồn tại thứ tự Tô-pô

**Giả thuyết 1.** *Một đồ thị có hướng tồn tại thứ tự Tô-pô khi và chỉ khi nó là một **DAG**.*

**Giả thuyết 2.** *Đồng nghĩa, mọi DAG đều tồn tại ít nhất một thứ tự Tô-pô.*

**Giả thuyết 3.** *Có thể tìm được một thứ tự Tô-pô bằng thuật toán trong thời gian tuyến tính $\mathcal{O}(V + E)$.*

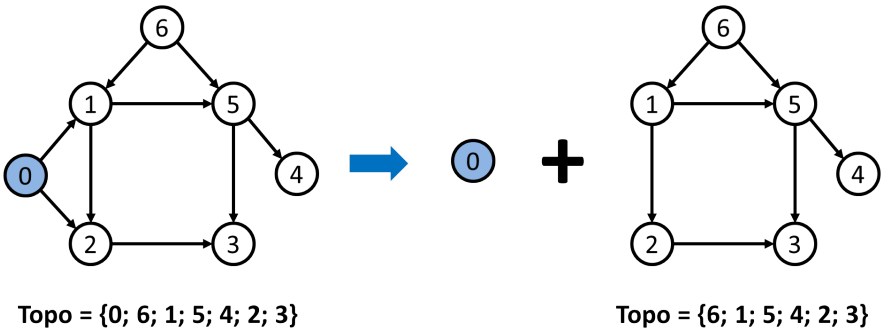
**Chứng minh 1.** *Ta sẽ chứng minh hai chiều của giả thuyết chính.*
***Chiều thuận:** Nếu đồ thị $G$ có chu trình, thì không thể tồn tại thứ tự tô-pô.*
*Giả sử tồn tại chu trình $v_1 \to v_2 \to \cdots \to v_n \to v_1$. Khi đó, theo định nghĩa của thứ tự tô-pô, ta có:*

$$\text{order}(v_1) < \text{order}(v_2) < \cdots < \text{order}(v_n) < \text{order}(v_1),$$

*tức là* $\text{order}(v_1) < \text{order}(v_1)$*, mâu thuẫn. Vậy, nếu có chu trình, không tồn tại thứ tự tô-pô.*

***Chiều nghịch:** Nếu $G$ là một DAG, thì tồn tại ít nhất một thứ tự tô-pô.*

1. *Vì $G$ không có chu trình, nên tồn tại ít nhất một đỉnh không có cung đi vào (bậc vào bằng 0). Nếu mọi đỉnh đều có bậc vào $\geq 1$, thì bắt đầu từ một đỉnh bất kỳ, ta luôn đi được sang đỉnh khác (vì có cung đi vào), và cuối cùng sẽ đi thành một chu trình, mâu thuẫn với giả thiết $G$ là DAG.*
2. *Gọi đỉnh đó là $u$. Đặt $u$ là đỉnh đầu tiên trong thứ tự tô-pô.*
3. *Loại bỏ $u$ khỏi đồ thị cùng tất cả các cung đi ra từ $u$. Đồ thị còn lại vẫn là DAG (vì việc xóa đỉnh không thể tạo ra chu trình mới).*
4. *Áp dụng lại quá trình trên với đồ thị còn lại: luôn tồn tại đỉnh có bậc vào bằng 0, đưa nó vào tiếp theo trong thứ tự.*
5. *Lặp lại cho đến khi tất cả các đỉnh được đưa vào thứ tự.*
6. *Cuối cùng, ta thu được một thứ tự thỏa mãn định nghĩa sắp xếp tô-pô.*




Hình 13: Minh họa chứng minh chiều nghịch

***Kết luận:*** *Với mỗi DAG, luôn tồn tại ít nhất một thứ tự tô-pô.*

### 2.3.4 Bài tập

**Bài tập 2.** *TOPOSORT - Sắp xếp TOPO*

**Đề bài** Cho đồ thị có hướng không chu trình $G(V, E)$. Hãy đánh số lại các đỉnh của $G$ sao cho chỉ có cung nối từ đỉnh có chỉ số nhỏ đến đỉnh có chỉ số lớn hơn.
**Input**

- Dòng đầu chứa hai số nguyên $n$ $(1 \leq n \leq 100)$ và $m$ $(0 \leq m \leq \frac{n(n-1)}{2})$
- $m$ dòng tiếp theo, mỗi dòng chứa một cặp số $u, v$ cho biết một cung nối từ $u \to v$ trong $G$.

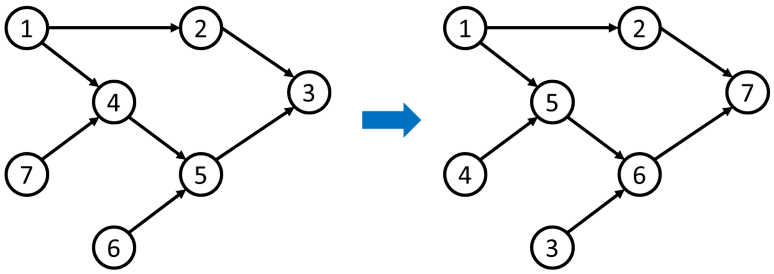
**Output**
Ghi ra $n$ số nguyên dương, số thứ $i$ là chỉ số của đỉnh $i$ sau khi đánh số lại. Hai số trên cùng một dòng được ghi cách nhau một dấu cách (space).
**Ví dụ**

Listing 5: Input

```
7 7
1 2
1 4
2 3
4 5
6 5
5 3
7 4
```

Listing 6: Output

```
1 2 7 5 6 3 4
```



Hình 14: Minh họa ví dụ

**Phân tích bài toán**
Với một đỉnh $u$ bất kỳ, khi DFS thăm đến một đỉnh u, sau khi đã đệ quy thăm hết tất cả các đỉnh con của nó, ta đẩy u vào ngăn xếp (Stack). Lúc này ngăn xếp sẽ chứa các đỉnh theo thứ tự "postorder" đảo ngược (reverse postorder) – postorder là thời điểm kết thúc của DFS tại đỉnh đó.

Để gán nhãn, ta lấy lần lượt từng đỉnh trên cùng của ngăn xếp (tương đương là đỉnh có postorder muộn nhất), gán nhãn lần lượt $1, 2, \ldots, n$ đến khi ngăn xếp rỗng (đã xử lý hết đỉnh). Ta đảm bảo được rằng, nếu $u$ được đẩy trước $v$ (tức là hoàn thành DFS sớm hơn), thì $u$ sẽ được gán nhãn lớn hơn.

Listing 7: Cài đặt Sắp xếp Tô-pô bằng DFS

```cpp
    #include <bits/stdc++.h>
#define int long long
#define endl "\n"
using namespace std;

int n, m;
vector<int> adj[101];
stack<int> st;
vector<bool> visited(101, false);

void dfs(int u) {
    visited[u] = true;
    for (auto v : adj[u]) {
        if (visited[v] == false) {
            dfs(v);
        }
    }
    st.push(u);
}

signed main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
    }
    for (int i = 1; i <= n; i++) {
        if (visited[i] == false) {
            dfs(i);
        }
    }
    vector<int> ans(n + 1, 0);
    int cnt = 1;
    while (st.empty() == false) {
        ans[st.top()] = cnt++;
        st.pop();
    }
    for (int i = 1; i <= n; i++) {
        cout << ans[i] << "␣";
    }
    return 0;
}
```

**Bài tập 3.** *Course Schedule*

**Đề bài**

Cho $n$ khóa học, có $m$ yêu cầu có dạng "khóa học $a$ phải được hoàn thành mới đủ điều kiện học khóa học $b$". Nhiệm vụ của bạn là tìm thứ tự học sao cho hoàn thành toàn bộ khóa học.

**Input**

- Dòng đầu tiên chứa 2 số nguyên $n$ ($1 \leq n \leq 10^5$) và $m$ ($1 \leq m \leq 2.10^5$)
- $m$ dòng tiếp theo mô tả các yêu cầu. Mỗi dòng chứa hai số nguyên $a$ và $b$ ($1 \leq a, b \leq n$): khóa học $a$ phải được hoàn thành trước khóa học $b$.

**Output**

In ra thứ tự học để hoàn thành các khóa học. Có thể in bất kỳ thứ tự nào thỏa mãn.
Nếu không tìm được thứ tự thỏa mãn, in ra "IMPOSSIBLE".

**Ví dụ**

Listing 8: Input

```
5 3
1 2
3 1
4 5
```

Listing 9: Output

```
4 5 3 1 2
```

**Phân tích bài toán**
Để tìm được thứ tự thỏa mãn, ta phải đảm bảo rằng đồ thị biểu diễn là một DAG.
Để kiểm tra đồ thị có phải là DAG, ta kiểm tra như sau:

- Gọi mảng kiểm tra trạng thái duyệt của đỉnh $i$ bất kỳ là visited[i] = 0, 1, 2 với ý nghĩa lần lượt là: chưa thăm, đang thăm, đã thăm xong.
- Khi duyệt đỉnh $u$, đặt trạng thái visited[u] = 1.
- Thăm các con $v_i$ của $u$, nếu tồn tại $v_i$ đang có trạng thái visited$[v_i] = 1$, nghĩa là đồ thị có chu trình → Không phải là DAG, ta in ra "IMPOSSIBLE".
- Ngược lại, nếu visited$[v_i] = 0$, ta thăm $v_i$.
- Sau khi thăm xong, ta đặt trạng thái visited[u] = 2.

Phần tìm thứ tự là một bài toán Sắp xếp Tô-pô, đã được mô tả thuật toán ở bài **TOPOSORT - Sắp xếp TOPO** phía trên.

Listing 10: Cài đặt

```cpp
#include <bits/stdc++.h>
#define int long long
#define endl "\n"
using namespace std;

vector<int> adj[100005];
int n, m;
vector<int> visited(100005, 0);
stack<int> st;
void dfs(int u) {
    visited[u] = 1;
    for (auto v : adj[u]) {
        if (visited[v] == 1) {
            cout << "IMPOSSIBLE";
            exit(0);
        }
        if (visited[v] == 0) dfs(v);
    }
    visited[u] = 2;
    st.push(u);
}
signed main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
    }
    for (int i = 1; i <= n; i++) {
        if (visited[i] == 0) {
            dfs(i);
        }
    }
    while (st.empty() == false) {
        cout << st.top() << "␣";
        st.pop();
    }
    return 0;
}
```

**Bài tập 4.** *Longest Path*

**Đề bài**
Cho đồ thị $G$ với $N$ đỉnh và $M$ cạnh. $G$ không tồn tại chu trình có hướng.

Hãy tìm độ dài của đường đi có hướng dài nhất trong đồ thị $G$. Độ dài đường đi có hướng dài nhất là tổng số cạnh có trong đường đi đó.
**Input**

- Dòng đầu tiên chứa 2 số nguyên $N$ $(1 \le n \le 10^5)$ và $M$ $(1 \le m \le 10^5)$
- $M$ dòng tiếp theo mô tả các yêu cầu. Mỗi dòng chứa hai số nguyên $x$ và $y$ $(1 \le x, y \le n)$ với ý nghĩa: tồn tại cạnh có hướng từ đỉnh $x \to y$.

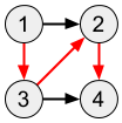
**Output**
In ra độ dài đường đi có hướng dài nhất trong đồ thị $G$.
**Ví dụ**

Listing 11: Input

```
4 5
1 2
1 3
3 2
2 4
3 4
```

Listing 12: Output

```
3
```




Hình 15: Mô tả ví dụ

**Phân tích bài toán**
Gọi $dp[u]$ là đường đi có hướng dài nhất kết thúc tại đỉnh $u$. Với $dp[i] = 0, \forall i \in [1..N]$.
Giả sử đã biết được giá trị $dp[u]$, với mỗi cạnh có hướng từ $u \to v$, ta cập nhật được:

$$dp[v] = \max(dp[v], dp[u] + 1)$$

Với ý nghĩa là nếu ta đi từ $u \to v$, thì đường đi có hướng dài nhất kết thúc ở $v$ có thể thu được bằng đường đi có hướng dài nhất đến $u$ cộng thêm 1 bước.
Để đảm bảo mỗi khi cập nhật dp[v] thì dp[u] đã được tính xong, ta duyệt các đỉnh theo thứ tự tô-pô của đồ thị.

Như vậy, sau khi lặp qua hết các cạnh theo thứ tự tô-pô, giá trị $\max_{1 \le i \le N} dp[i]$ chính là độ dài đường đi có hướng dài nhất trong toàn đồ thị.

**Câu hỏi phụ:** Tại sao với $dp[v]$ cần lấy $\max(dp[v], dp[u] + 1)$?
**Trả lời luôn:** Vì trước khi thăm $v$ từ đỉnh $u$, có thể tồn tại đường đi dài nhất kết thúc tại $v$ mà không thông qua đỉnh $u$. Vì vậy ta cần lấy max của 2 trường hợp: tồn tại đường đi dài nhất kết thúc tại $v$ mà: không qua $u$ & qua $u$.

**Ngoài lề:** Vì tác giả lười xử lý/thao tác trên ngăn xếp (stack) nên sau này với các bài toán sắp xếp tô-pô, tác giả sẽ thao tác trên vector và reverse vector để lấy thứ tự tô-pô (Vector is the best data structure in C++/the world).

Listing 13: Cài đặt

```cpp
#include <bits/stdc++.h>
#define int long long
#define endl "\n"
using namespace std;

vector<int> adj[100005];
int n, m;
vector<int> visited(100005, 0);
vector<int> dp(100005, 0);
vector<int> st;

void dfs(int u) {
    visited[u] = true;
    for (auto v : adj[u]) {
        if (!visited[v]) dfs(v);
    }
    st.push_back(u);
}

signed main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
    }

    for (int i = 1; i <= n; i++) {
        if (visited[i] == false) {
            dfs(i);
        }
```

```
    }
    reverse(st.begin(), st.end());

    for (auto u : st) {
        for (auto v : adj[u]) {
            dp[v] = dp[u] + 1;
        }
    }
    int ans = 0;
    for (int i = 1; i <= n; i++) ans = max(ans, dp[i]);
    cout << ans;
    return 0;
}
```

## 2.4 Cây DFS (Depth-First Search Tree) và ứng dụng

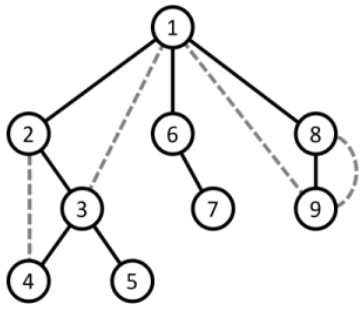
Nội dung bài chủ yếu tham khảo/copy từ [VNOI WIKI]: https://wiki.vnoi.info/algo/graph-theory/Depth-First-Search-Tree.md

### 2.4.1 Cây duyệt chiều sâu DFS (cây DFS)

Trong quá trình $DFS$, với mỗi đỉnh $u$ ta có $par[u]$ là số hiệu của đỉnh mà từ đỉnh đó thủ tục $DFS$ gọi để quy đến $u$. Xây dựng đồ thị con với các cạnh là $(par[u], u)$, ta có được một cây. Cây này được gọi là **cây** $DFS$.
Các cạnh thuộc cây $DFS$ được gọi là các "cạnh nét liền".
Các cạnh còn lại không thuộc cây $DFS$ được gọi là các "cạnh nét đứt".

Nói cách khác, khi ta thực hiện $DFS$, tưởng tượng như sau:

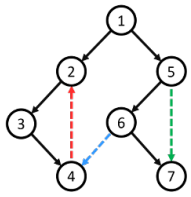
1. Bắt đầu từ một đỉnh gốc:
   - Ta gọi DFS tại đó, coi nó là "gốc" của cây.
2. Mỗi lần đi từ u xuống v lần đầu tiên
   - Nếu v chưa được thăm, ta đánh dấu par[v] = u (vì u "gọi" v), và gọi tiếp DFS(v).
   - Cạnh (u, v) đó chính là một cạnh cây (nét liền), vì nó nằm trên hành trình ta thực sự đi.
3. Khi gặp một cạnh nối u với một đỉnh v đã thăm rồi
   - Ta không đi tiếp, vì v đã vào cây.
   - Cạnh đó được gọi là cạnh không phải cây (nét đứt). Nó chỉ là "đường tắt" giữa hai đỉnh đã có trong cây.




Hình 16: Minh họa cây DFS

Trong đồ thị có hướng, xét các cung được thăm và không được thăm bởi $DFS$, ta có 4 loại cung sau:

- Cung của cây $DFS$ **(Tree egde):** là các cung thuộc cây $DFS$ được định hướng theo chiều từ cha đến con. (ví dụ cạnh $(u, v)$ thuộc cây $DFS$ mà $u$ được thăm trước $v$ hay $u$ là cha của $v$ thì ta có cung $u \to v$ là cung của cây $DFS$). < Các cung của cây $DFS$ được đánh dấu là các cạnh màu đen trong hình bên dưới >
- Cung xuôi **(Forward edge):** là các cung không thuộc cây $DFS$ và có dạng $u \to v$ trong đó $u$ là tổ tiên của $v$ trong cây $DFS$. < Các cung xuôi được đánh dấu là các cạnh màu xanh lá trong hình bên dưới >
- Cung ngược **(Back edge):** là các cung không thuộc cây $DFS$ và có dạng $v \to u$ trong đó $u$ là tổ tiên của $v$ trong cây $DFS$. < Các cung ngược được đánh dấu là các cạnh màu đỏ trong hình bên dưới >
- Cung chéo **(Cross edge):** là các cung không thuộc cây $DFS$ và có dạng $u \to v$ trong đó $u$ và $v$ thuộc hai nhánh khác nhau của cùng một cây $DFS$. < Các cung chéo được đánh dấu là các cạnh màu xanh dương trong hình bên dưới >




Hình 17: Mô tả các loại cung trong cây

Trong đồ thị vô hướng:

- Không tồn tại cung chéo. Vì khi đỉnh $u$ được duyệt trong hàm $DFS$ ta sẽ duyệt tất cả các đỉnh $v$ kề $u$ mà $v$ chưa được thăm. Như vậy nếu tồn tại một cung chéo $(u, v)$ chứng tỏ khi duyệt đến đỉnh $u$ hoặc đỉnh $v$ ta đã không duyệt cạnh $(u, v)$.
- Vì các cạnh trên đồ thị vô hướng không được định chiều nên không thể định nghĩa 2 loại cung xuôi và cung ngược như ở đồ thị có hướng. Do đó, ở đồ thị vô hướng, cung xuôi và cung ngược sẽ được định nghĩa như sau:
  - Cung xuôi **(Forward edge):** là các cung thuộc cây $DFS$. Hay còn có cách gọi khác là "cạnh nét liền" hoặc "cung của cây $DFS$".
  - Cung ngược (Back edge): là các cung không thuộc cây $DFS$. Hay còn có cách gọi khác là "cạnh nét đứt".
- Như vậy trên đồ thị vô hướng lúc này chỉ còn loại cung là cung ngược và cung xuôi (cung của cây $DFS$).

**Một số mảng quan trọng trong cây DFS:**

- Mảng **num[]**: cho biết thứ tự duyệt DFS của các đỉnh (thứ tự mà mỗi đỉnh bắt đầu duyệt).
- Mảng **low[]**: Với mỗi đỉnh $u$, $low[u]$ cho biết thứ tự (giá trị $num$) nhỏ nhất có thể đi đến được từ $u$ bằng cách đi xuôi xuống theo các cạnh nét liền (các cung trên cây DFS) và kết thúc đi ngược lên không quá 1 lần theo cạnh nét đứt. Ngoài ra ta cũng có thể hiểu ý nghĩa của $low[u]$ là thứ tự thăm của đỉnh có thứ tự thăm sớm nhất nằm trong cây con gốc $u$ hoặc kề cạnh với 1 đỉnh bất kì nằm trong cây con gốc $u$.
- Mảng **tail[]**: cho biết thời điểm kết thúc duyệt DFS của mỗi đỉnh cũng là thời điểm duyệt xong của đỉnh đó.

**Nhận xét:** Các đỉnh có thứ tự thăm nằm trong khoảng từ $num[u]$ đến $tail[u]$ chính là các đỉnh nằm trong cây con gốc $u$ trong cây DFS.

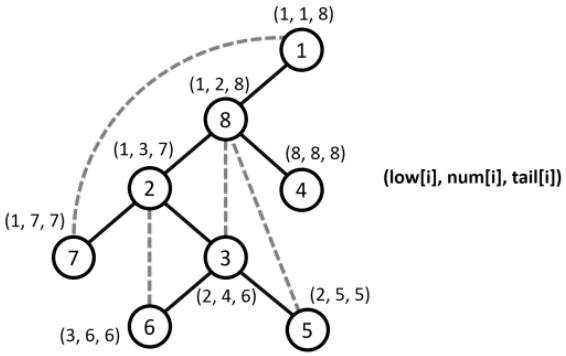
**Cách tính mảng low[], num[], tail[]:**

- **Ý tưởng chính:** Mảng $num[]$, $tail[]$ ta có thể tính dễ dàng bằng cách DFS xác định thời điểm duyệt tới và thời điểm duyệt xong của các đỉnh. Với mảng $low[]$ ta có:
  - Trước hết, với 1 đỉnh $u$ bất kì có thể tự đi tới chính nó nên ta gán $low[u] = num[u]$.
  - Từ $u$ có thể đến các đỉnh $v$ kề u bằng 1 cạnh nét đứt nên ta có $low[u] = \min(low[u], num[v])$ với $(u, v)$ là một cạnh nét đứt.
  - Ngược lại, nếu $(u, v)$ là một cạnh nét liền và v không phải cha của u ta có $low[u] = \min(low[u], low[v])$ do từ $u$ ta có thể đi xuống $v$ sau đó đi theo con đường đã xác định ở đỉnh $v$ để tới đỉnh có thứ tự thăm là $low[v]$.
- **Chú ý:** Giá trị thực sự của $num[], low[]$ được xác định bằng giá trị thực sự của $low[u], tail[u]$ chỉ được xác định khi đã duyệt xong đỉnh $u$. Thời điểm duyệt tới của một đỉnh $u$ luôn diễn ra trước thời điểm duyệt tới của các đỉnh trong cây con gốc $u$ của cây DFS, thời điểm duyệt xong của đỉnh $u$ luôn diễn ra sau thời điểm duyệt xong của các đỉnh trong cây con gốc $u$.
- **Cách thực hiện:**
  - Đầu tiên ta sẽ bắt đầu duyệt $DFS$ từ đỉnh gốc. Khi duyệt tới đỉnh $u$ ta sẽ cập nhật thời điểm duyệt tới. Lúc này $low[u] = num[u] =$ *thứ tự duyệt DFS*. Ta sẽ duyệt tất cả các con $v$ trong gốc $u$.
  - **Trường hợp 1:** Nếu đỉnh $v$ chưa được thăm thì sau khi hoàn thành $DFS$ của $v$ thì ta sẽ cập nhật lại giá trị của $low[u]$: $low[u] = \min(low[u], low[v])$;
  - **Trường hợp 2:** Nếu đỉnh $v$ đã được thăm, thì ta sẽ cập nhật lại giá trị cho $low[u]$: $low[u] = \min(low[u], num[v])$;
  - Ở trường hợp này ta không thể cập nhật $low[u] = \min(low[u], low[v])$ được. Vì khi ta thăm đến đỉnh $u$ mà đỉnh $v$ đã được thăm thì tức là $(u, v)$ là một cạnh nét đứt, do đó khi đi từ $u$ ta đã sử dụng 1 cạnh nét đứt nên không thể tiếp tục di chuyển nữa (theo định nghĩa của mảng $low[]$) suy ra ta chỉ cập nhật $low[u] = \min(low[u], num[v])$.
- **Chú ý:** *Nếu v là cha trực tiếp của u thì ta bỏ qua không xét đến.*
- Khi đã duyệt xong đỉnh $u$ và các nút trong cây con $DFS$ gốc $u$ ta sẽ tiến hành cập nhật giá trị $tail[u] =$ *thời gian duyệt DFS hiện tại.*

Listing 14: Cài đặt

```
int timeDfs = 0;

void dfs(int u, int pre) {
    num[u] = low[u] = ++timeDfs;
    for (int v : g[u]){
        if (v == pre) continue;
        if (!num[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
        }
        else low[u] = min(low[u], num[v]);
    }
    tail[u] = timeDfs;
}
```



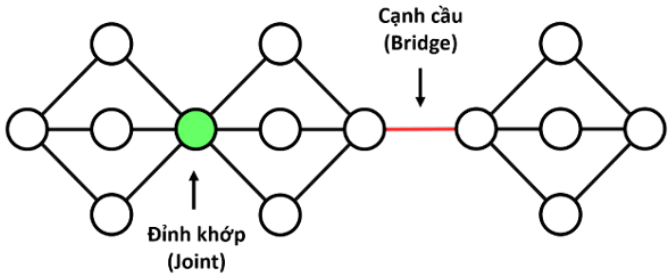

Hình 18: Ví dụ minh họa

## 2.5 Khớp và Cầu (Joints and Brides)

**Định nghĩa 7.**

*Trong đồ thị vô hướng, một đỉnh được gọi là đỉnh khớp nếu như loại bỏ đỉnh này và các cạnh liên thuộc với nó ra khỏi đồ thị thì số thành phần liên thông của đồ thị tăng lên.*

*Trong đồ thị vô hướng, một cạnh được gọi là cạnh cầu nếu như loại bỏ cạnh này ra khỏi đồ thị thì số thành phần liên thông của đồ thị tăng lên.*




Hình 19: Minh họa khớp và cầu

**Bài tập 5.** *GRAPH_ - Tìm khớp và cầu (Cơ bản)*

Xét đơn đồ thị vô hướng $G = (V, E)$ có $N$ $(1 \leq N \leq 10000)$ đỉnh và $M$ $(1 \leq M \leq 50000)$ cạnh. Người ta định nghĩa một đỉnh gọi là khớp nếu như xóa đỉnh đó sẽ làm tăng số thành phần liên thông của đồ thị. Tương tự như vậy, một cạnh được gọi là cầu nếu xóa cạnh đó sẽ làm tăng số thành phần liên thông của đồ thị.
Vấn đề đặt ra là cần đếm tất cả các khớp và cầu của đồ thị $G$.
**Input**

- Dòng đầu: chứa hai số tự nhiên $N$, $M$.
- $M$ dòng sau, mỗi dòng chứa một cặp số $(u, v)$ $(u \neq v, 1 \leq u \leq N, 1 \leq v \leq N)$ mô tả một cạnh của $G$.

**Output**

- Gồm một dòng duy nhất ghi hai số, số thứ nhất là số khớp, số thứ hai là số cầu của $G$.

**Example**

Listing 15: Input

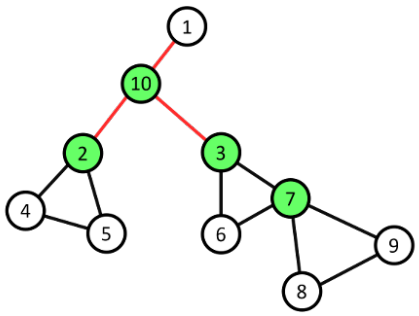
```
10 12
1 10
10 2
10 3
2 4
4 5
5 2
3 6
6 7
7 3
7 8
8 9
9 7
```

Listing 16: Output

```
4 3
```

**Note**

- Các cạnh màu đỏ là cạnh cầu.
- Các đỉnh màu xanh lá là đỉnh khớp.

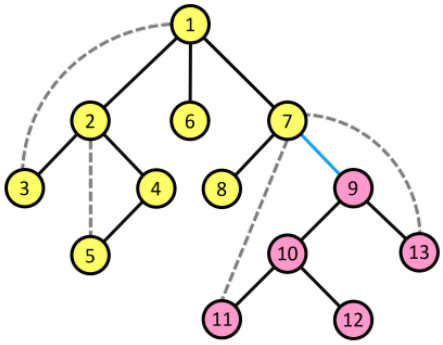



Hình 20: Minh họa ví dụ

**Phân tích**

**Tìm cạnh cầu**

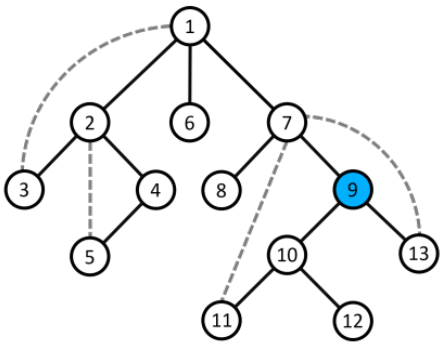
- Dễ thấy rằng cạnh cầu của đồ thị không thể là cạnh nét đứt vì việc bỏ đi cạnh nét đứt sẽ không ảnh hưởng đến tính liên thông giữa các đỉnh của đồ thị. Do vậy, cạnh cầu chỉ có thể là cạnh nét liền.
- Ta sẽ xét riêng từng thành phần liên thông của đồ thị. Xét vùng liên thông $G$ như sau:
  - Xét cây con gốc $v$ trong cây $DFS$ của $G$ có $u$ là cha trực tiếp của $v$. Gọi tập hợp các đỉnh thuộc cây con gốc $v$ là $A$, tập hợp các đỉnh không thuộc cây con gốc $v$ là $B$. Khi xóa đi cạnh $(u, v)$ thì giữa 2 đỉnh bất kì thuộc cùng 1 tập hợp vẫn có thể đến với nhau bằng các cạnh nét liền. Một đỉnh thuộc $A$ với một đỉnh thuộc $B$ muốn đi đến với nhau bằng các **cạnh nét liền** thì đều phải thông qua cạnh $(u, v)$.
  - **Ví dụ minh họa:** Xét cạnh nét liền $(7, 9)$ với đỉnh 9 là con trực tiếp của đỉnh 7 trên cây $DFS$. Tập đỉnh $A$ là các đỉnh được đánh dấu màu hồng. Tập đỉnh $B$ là các đỉnh được đánh dấu màu vàng. Đỉnh 11 thuộc tập $A$ muốn đi đến đỉnh 6 thuộc tập $B$ bằng các cạnh nét liền thì đều phải thông qua cạnh $(7, 9)$.
- Giả sử không có cạnh nét đứt nào nối giữa 1 đỉnh thuộc $A$ với 1 đỉnh thuộc $B$ thì khi xóa cạnh $(u, v)$, $G$ sẽ tách ra thành 2 vùng liên thông $A$ và $B$. Ngược lại nếu tồn tại cạnh nét đứt nối giữa 1 đỉnh thuộc $A$ và 1 đỉnh thuộc $B$ đồ thị vẫn liên thông. Do đó ta chỉ cần xét xem có tồn tại cạnh nét đứt nối giữa $A$ và $B$ hay không để kết luận $(u, v)$ có phải cầu không?
- Ta có từ $v$ có thể đi đến một đỉnh $p$ nào đó có $num[p] = low[v]$ bằng cách đi theo các cung của cây $DFS$ và đi qua không quá 1 cạnh nét đứt và $p$ có thứ tự thăm sớm nhất khi $DFS$. Nếu $p$ nằm trong $B$ thì $p$ phải là tổ tiên của $v$ cũng đồng nghĩa với việc $num[p] < num[v]$ hay $low[v] < num[v]$ (**vì đồ thị không có cung chéo**), nghĩa là tồn tại 1 cạnh nét đứt nối giữa 1 đỉnh thuộc $A$ với 1 đỉnh thuộc $B$ (vì nếu chỉ đi bằng các cung của cây $DFS$ thì $v$ không thể tới một tổ tiên của nó).



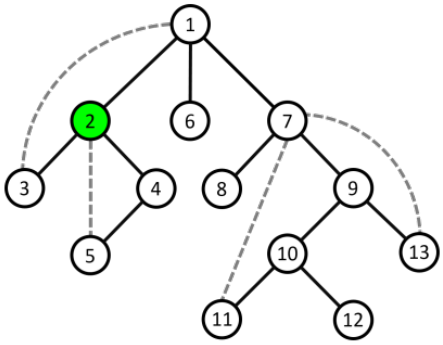

- Do đó nếu $low[v] \geq num[v]$ chắc chắn đỉnh $p$ thuộc cây con gốc $v$ hay $p$ thuộc tập hợp $A$ khi đó không tồn tại cạnh nét đứt nối giữa 1 đỉnh thuộc $A$ với 1 đỉnh thuộc $B$. Tuy nhiên, ta dễ dàng nhận thấy $low[v] \leq num[v]$ vì đỉnh $v$ luôn tới được chính nó.

**Kết luận:** Nếu $low[v] = num[v]$ thì $(u, v)$ là một cạnh cầu trong đồ thị.
**Tìm đỉnh khớp**

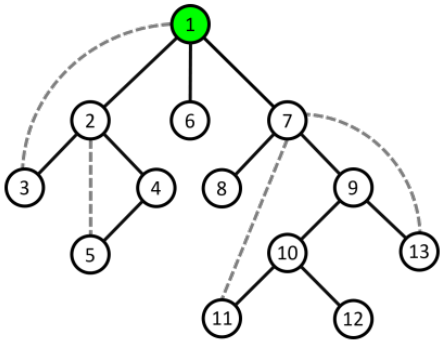
- Ta sẽ xét riêng từng thành phần liên thông của đồ thị. Xét vùng liên thông $G$ như sau:
  - Xét cây con gốc $u$ trong cây $DFS$ của $G$, nếu mọi nhánh con của $u$ đều có cung ngược lên tới tổ tiên của $u$ ($low[v] < num[u]$, với $v$ là tất cả các con trực tiếp của $u$ trên cây $DFS$) thì đỉnh $u$ không thể là đỉnh khớp. Bởi trong đồ thị ban đầu, nếu ta loại bỏ đỉnh $u$ đi thì từ mỗi đỉnh bất kỳ thuộc nhánh con vẫn có thể đi lên một tổ tiên của $u$, rồi đi sang nhánh con khác hoặc đi sang tất cả những đỉnh còn lại của cây nên số thành phần liên thông của đồ thị không thay đổi.
  - **Ví dụ minh họa:** Xét đỉnh 9 không phải là đỉnh khớp vì cả 2 nhánh con của nó là cây con gốc 10 và cây con gốc 13 trong cây $DFS$ đều có cung ngược lên tới đỉnh 7 là tổ tiên của đỉnh 9.




- Nếu $u$ không phải là đỉnh gốc của cây $DFS$, và tồn tại ít nhất một nhánh con trong cây con gốc $u$ không có cung ngược lên một tổ tiên của $u$ ($low[v] \geq num[u]$, với $v$ là một con trực tiếp bất kì của $u$ trên cây $DFS$) thì đỉnh $u$ là đỉnh khớp. Bởi khi đó, tất cả những cung xuất phát từ nhánh con đó chỉ có thể đi tới những đỉnh thuộc cây con gốc $u$ mà thôi, trên đồ thị ban đầu, không tồn tại cạnh nối từ những đỉnh thuộc nhánh con đó tới một tổ tiên của $u$. Vậy nên từ một đỉnh bất kì thuộc nhánh con đó muốn đi lên một tổ tiên của $u$ thì bắt buộc phải đi qua $u$ nên việc loại bỏ đỉnh $u$ ra khỏi đồ thị sẽ làm tăng số thành phần liên thông của đồ thị.
- **Ví dụ minh họa:** Xét đỉnh 2 là đỉnh khớp vì tồn tại 1 nhánh con của nó là cây con gốc 4 không có cung ngược lên tới tổ tiên của đỉnh 2.



- Nếu $u$ là đỉnh gốc của cây $DFS$, thì $u$ là đỉnh khớp khi và chỉ khi $u$ có ít nhất 2 nhánh con. Vì đồ thị không có cung chéo nên khi $u$ có 2 nhánh con thì đường đi giữa hai đỉnh thuộc hai nhánh con đó bắt buộc phải đi qua $u$. Việc loại bỏ đỉnh $u$ ra khỏi đồ thị sẽ làm tăng số thành phần liên thông của đồ thị.
- **Ví dụ minh họa:** Xét đỉnh 1 là đỉnh khớp vì đỉnh 1 là đỉnh gốc của cây $DFS$ và có tới 3 nhánh con.




**Kết luận:** Đỉnh $u$ là đỉnh khớp khi:

- Đỉnh $u$ không phải là gốc của cây $DFS$ và $low[v] \geq num[u]$ (với $v$ là một con trực tiếp bất kì của $u$ trong cây $DFS$).
- **Hoặc**
- Đỉnh $u$ là gốc của cây $DFS$ và có ít nhất 2 con trực tiếp trong cây $DFS$.

### Cài đặt

**Cấu trúc dữ liệu:**

- Hằng số `maxN = 10010`
- Biến `timeDfs` – Thứ tự $DFS$
- Biến `bridge` – Số lượng cạnh cầu
- Mảng `low[]`, `num[]`
- Mảng `joint[]` – Đánh dấu đỉnh khớp
- Vector `g[]` – Danh sách cạnh kề của mỗi đỉnh

```cpp
#include <bits/stdc++.h>

using namespace std;

const int maxN = 10010;

int n, m;
bool joint[maxN];
int timeDfs = 0, bridge = 0;
int low[maxN], num[maxN];
vector <int> g[maxN];

void dfs(int u, int pre) {
    int child = 0; // So luong con truc tiep cua dinh u trong cay DFS
    num[u] = low[u] = ++timeDfs;
    for (int v : g[u]) {
        if (v == pre) continue;
        if (!num[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] == num[v]) bridge++;
            child++;
            if (u == pre) { // Neu u la dinh goc cua cay DFS
                if (child > 1) joint[u] = true;
            }
            else if (low[v] >= num[u]) joint[u] = true;
        }
        else low[u] = min(low[u], num[v]);
    }
}

```

```cpp
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    for (int i = 1; i <= n; i++)
        if (!num[i]) dfs(i, i);

    int cntJoint = 0;
    for (int i = 1; i <= n; i++) cntJoint += joint[i];

    cout << cntJoint << '␣' << bridge;
}
```

### 2.6 Thành phần liên thông mạnh (Strongly Connected Components)

### 2.7 Thuật toán BFS

### 2.8 Thuật toán Dijkstra + Heap

### 2.9 Disjoint Set Union (DSU)

### 2.10 Maximum Flow and Maximum Matching

### 2.11 Minimum Cut

### 2.12 Euler Tour

### 2.13 Lowest Common Ancestor

### 2.14 Heavy Light Decomposition

### 2.15 Centroid Decomposition

### 2.16 2-SAT

## 3 Miscellaneous

### 3.1 Contributors