

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Thuật toán ứng dụng

Nguyễn Khánh Phương

**Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn**

Nội dung khóa học

Chương 1. Các cấu trúc dữ liệu và thư viện

Chương 2. Kỹ thuật đệ quy và nhánh cận

Chương 3. Chia để trị

Chương 4. Quy hoạch động

Chương 5. Các thuật toán trên đồ thị và ứng dụng

Chương 6. Các thuật toán xử lý xâu và ứng dụng

Chương 7. Lớp bài toán NP-đầy đủ

Các ứng dụng thực tế của đồ thị

- Có tiềm năng ứng dụng trong nhiều lĩnh vực:
 - Mạng máy tính
 - Mạng giao thông
 - Mạng điện
 - Mạng cung cấp nước
 - Lập lịch
 - Tối ưu hóa luồng, thiết kế mạch
 - Phân tích gen DNA
 - Trò chơi máy tính
 - Thiết kế hướng đối tượng
 -

Nội dung

1. Đồ thị và cách biểu diễn đồ thị
2. Duyệt đồ thị
3. Một số bài toán cơ bản

Nội dung

1. Đồ thị và cách biểu diễn đồ thị
2. Duyệt đồ thị
3. Một số bài toán cơ bản

1. Đồ thị và cách biểu diễn đồ thị

1.1. Đồ thị vô hướng và có hướng

1.2. Một số khái niệm cơ bản trên đồ thị

1.3. Một số dạng đồ thị đặc biệt

1.4. Biểu diễn đồ thị

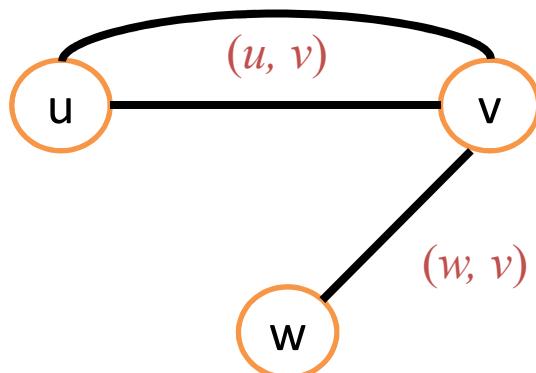


Đồ thị vô hướng (Undirected Graphs)

Định nghĩa. Đơn (đa) đồ thị vô hướng $G = (V, E)$ là cặp gồm:

- Tập đỉnh V là tập hữu hạn phần tử, các phần tử gọi là các **đỉnh**
- Tập cạnh E là tập (họ) các bộ không có thứ tự dạng

$$(u, v), \text{ với } u, v \in V, u \neq v$$



Đa đồ thị vô hướng

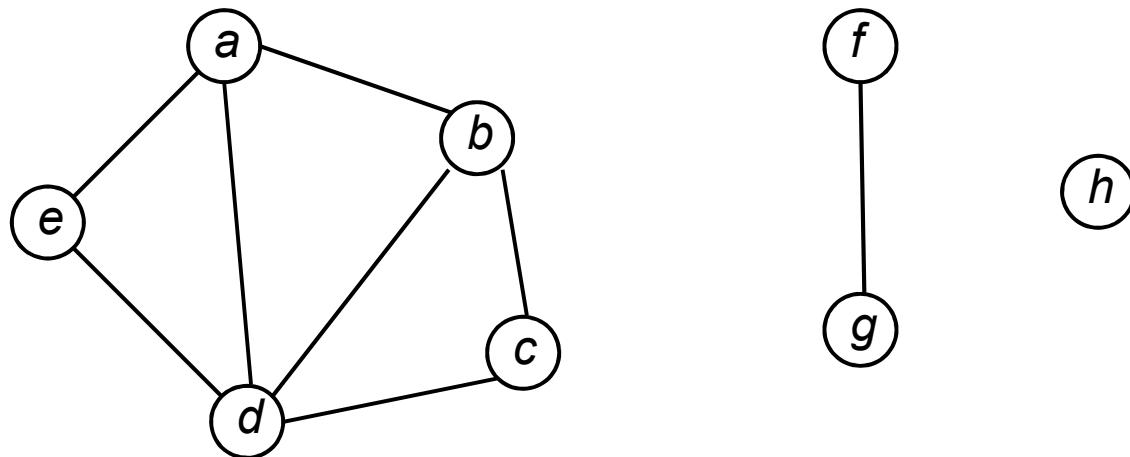
Đơn đồ thị vô hướng

Đơn đồ thị vô hướng (Simple Graph)

- **Ví dụ:** Đơn đồ thị $G_1 = (V_1, E_1)$, trong đó

$$V_1 = \{a, b, c, d, e, f, g, h\},$$

$$E_1 = \{(a,b), (b,c), (c,d), (a,d), (d,e), (a,e), (d,b), (f,g)\}.$$



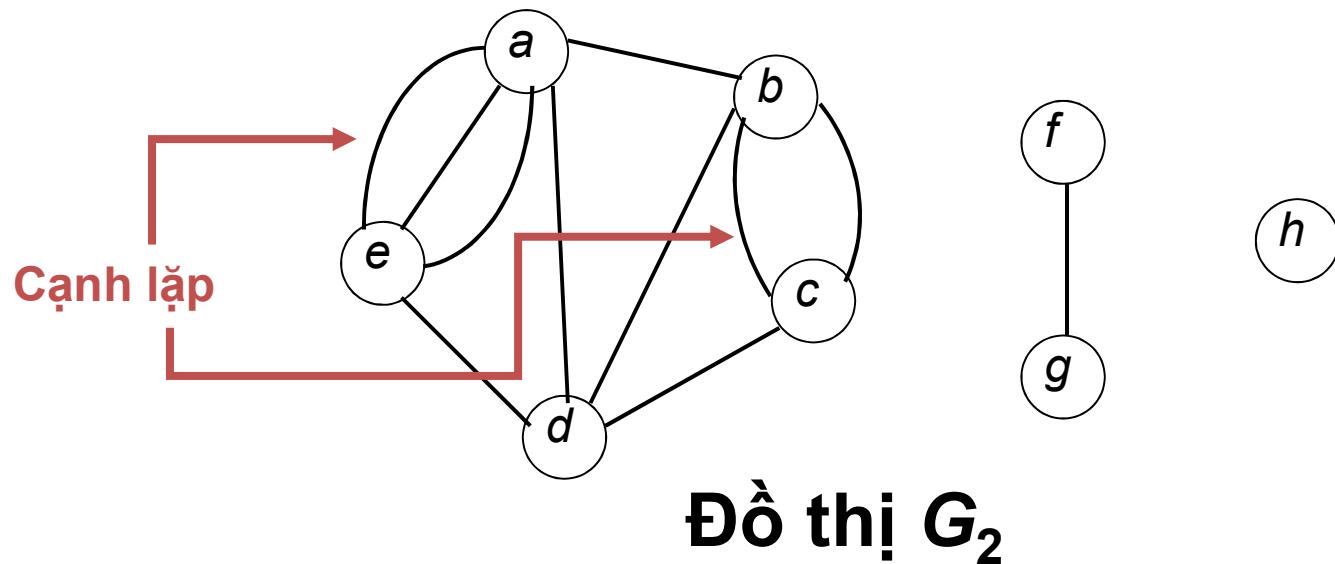
Đồ thị G_1

Đa đồ thị vô hướng (Multi Graphs)

- **Ví dụ:** Đa đồ thị $G_2 = (V_2, E_2)$, trong đó

$$V_2 = \{a, b, c, d, e, f, g, h\},$$

$$E_2 = \{(a,b), (b,c), (b,c), (c,d), (a,d), (d,e), (a,e), (a,e), (a, e), (d,b), (f,g)\}.$$

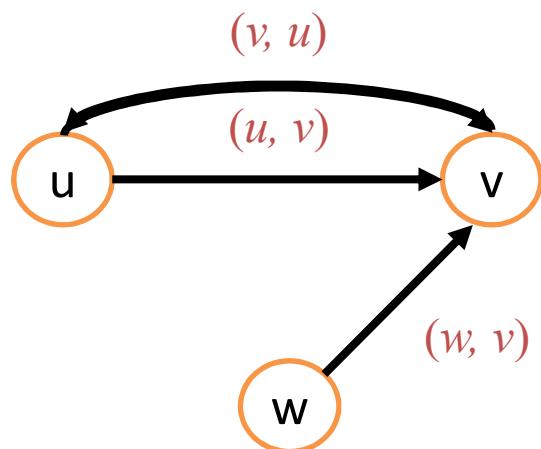


Đồ thị có hướng (Directed Graph)

Định nghĩa. Đơn (đa) đồ thị có hướng $G = (V, E)$ là cặp gồm:

- Tập đỉnh V là tập hữu hạn phần tử, các phần tử gọi là các **đỉnh**
- Tập cung E là tập (họ) các bộ có thứ tự dạng

$$(u, v), \text{ với } u, v \in V, u \neq v$$

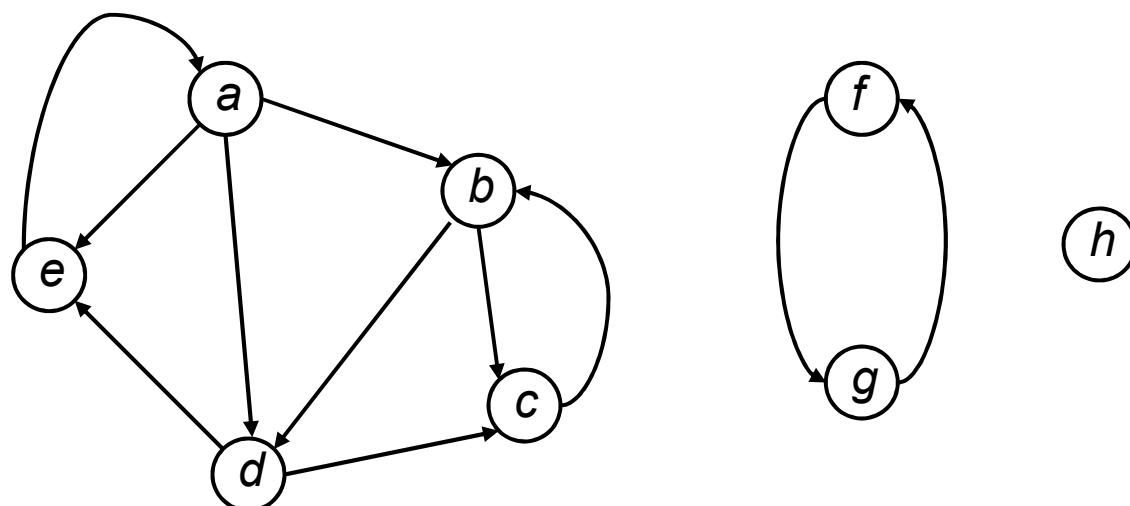


Đa đồ thị có hướng

Đơn đồ thị có hướng

Đơn đồ thị có hướng (Simple digraph)

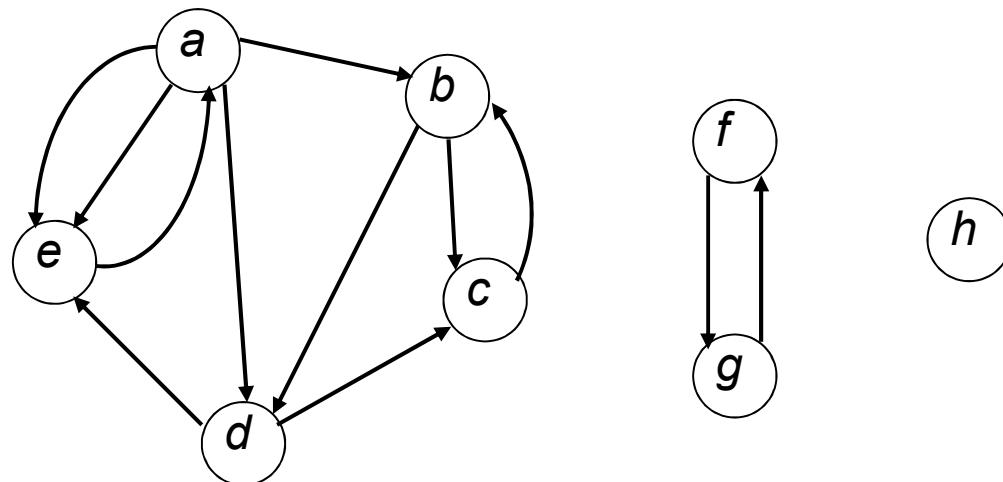
- **Ví dụ:** Đơn đồ thị có hướng $G_3 = (V_3, E_3)$, trong đó
 $V_3 = \{a, b, c, d, e, f, g, h\}$,
 $E_3 = \{(a,b), (b,c), (c,b), (d,c), (a,d), (b, d), (a,e), (d,e), (e,a), (f,g), (g,f)\}$



Đồ thị G_3

Đa đồ thị có hướng (Multi Graphs)

- **Ví dụ:** Đa đồ thị có hướng $G_4 = (V_4, E_4)$, trong đó
 $V_4 = \{a, b, c, d, e, f, g, h\}$,
 $E_4 = \{(a,b), (b,c), (c,b), (d,c), (a,d), (b, d), (a,e), (a,e), (d,e), (e,a), (f,g), (g,f)\}$



Đồ thị G_4

Các loại đồ thị: Tóm tắt

| Loại | Kiểu cạnh | Có cạnh lặp? |
|---------------------|-----------|--------------|
| Đơn đồ thị vô hướng | Vô hướng | Không |
| Đa đồ thị vô hướng | Vô hướng | Có |
| Đơn đồ thị có hướng | Có hướng | Không |
| Đa đồ thị có hướng | Có hướng | Có |

- Chú ý:
 - Một dạng đồ thị ít sử dụng hơn, đó là giả đồ thị.
Giả đồ thị là đa đồ thị mà trong đó có các **khuyên** (cạnh nối 1 đỉnh với chính nó).

Khuyên (loop)



1. Đồ thị và cách biểu diễn đồ thị

1.1. Đồ thị vô hướng và có hướng

1.2. Một số khái niệm cơ bản trên đồ thị

1.3. Một số dạng đồ thị đặc biệt

1.4. Biểu diễn đồ thị



1.2. Một số khái niệm cơ bản trên đồ thị

1.2.1. Kề

1.2.2. Bậc của đỉnh

1.2.3. Loại bỏ đỉnh

1.2.4. Hợp của hai đồ thị

1.2.5. Đồ thị con

1.2.6. Đồ thị con bao trùm

1.2.7. Đường đi và chu trình

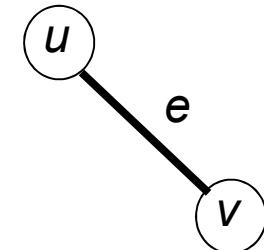
1.2.8. Tính liên thông

1.2.1. Kề (Adjacency)

Cho G là đồ thị vô hướng với tập cạnh E . Giả sử $e = (u, v)$ là cạnh của G .

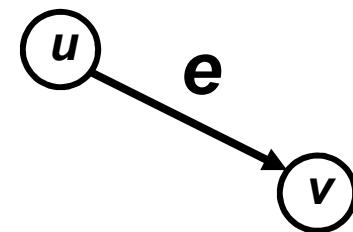
Khi đó ta nói:

- u, v là *kề nhau/lân cận/nối với nhau* (*adjacent / neighbors / connected*).
- Cạnh e là *liên thuộc* với hai đỉnh u và v .
- Cạnh e *nối (connect)* u và v .
- Các đỉnh u và v là các *đầu mút (endpoints)* của cạnh e .



Cho G là đồ thị có hướng (có thể là đơn hoặc đa). Giả sử $e = (u, v)$ là cạnh của G . Ta nói:

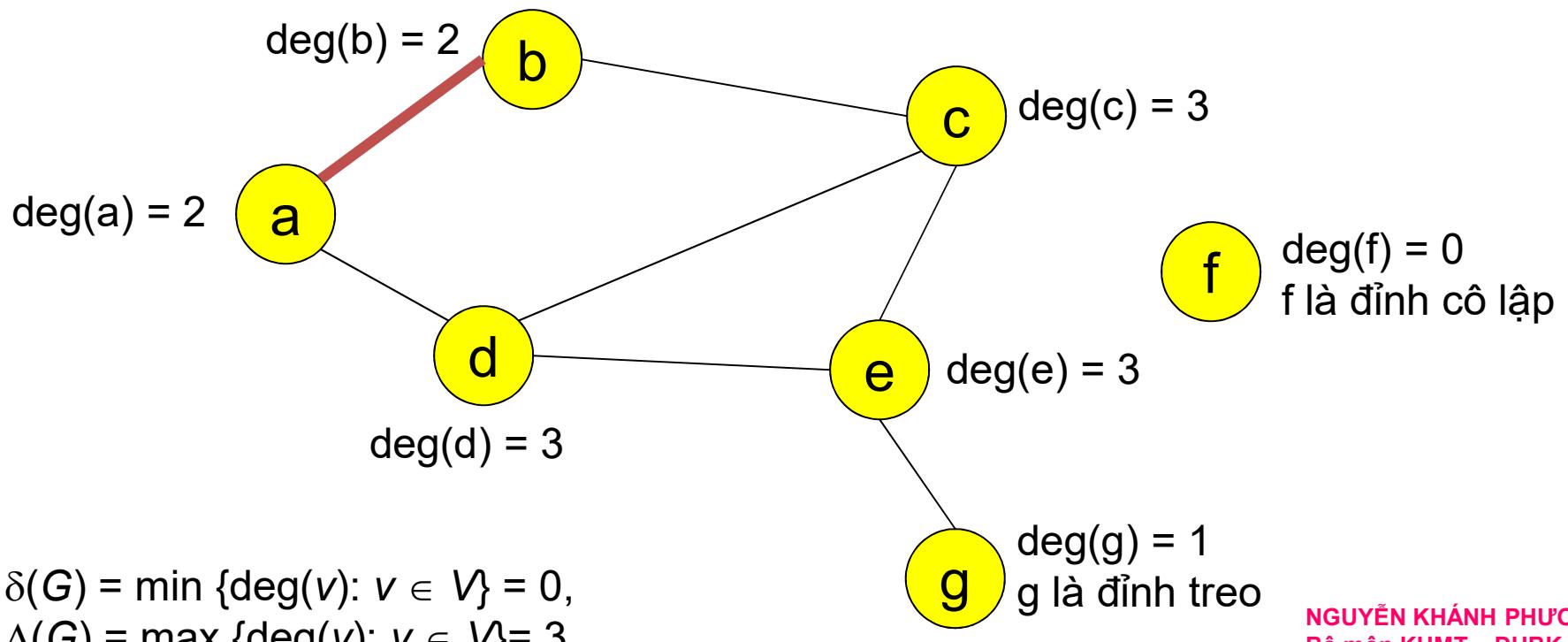
- u và v là *kề nhau*, u là *kề tới* v , v là *kề từ* u
- e *đi ra khỏi* u , e *đi vào* v .
- e *nối* u *với* v , e *đi từ* u *tới* v
- Đỉnh đầu (*initial vertex*) của e là u
- Đỉnh cuối (*terminal vertex*) của e là v



1.2.2. Độ của đỉnh (Degree of a Vertex)

Giả sử G là đồ thị vô hướng, $v \in V$ là một đỉnh nào đó của G :

- Độ của đỉnh v , $\deg(v)$, là số cạnh kề với nó.
- Đỉnh bậc 0 được gọi là *đỉnh cô lập* (*isolated*).
- Đỉnh bậc 1 được gọi là *đỉnh treo* (*pendant*).
- Các ký hiệu thường dùng: $\delta(G) = \min \{\deg(v): v \in V\}$,
 $\Delta(G) = \max \{\deg(v): v \in V\}$.



Định lý về các cái bắt tay (Handshaking Theorem)

- **Định lý.** Giả sử G là đồ thị vô hướng (đơn hoặc đa) với tập đỉnh V và tập cạnh E . Khi đó

$$\sum_{v \in V} \deg(v) = 2|E|$$

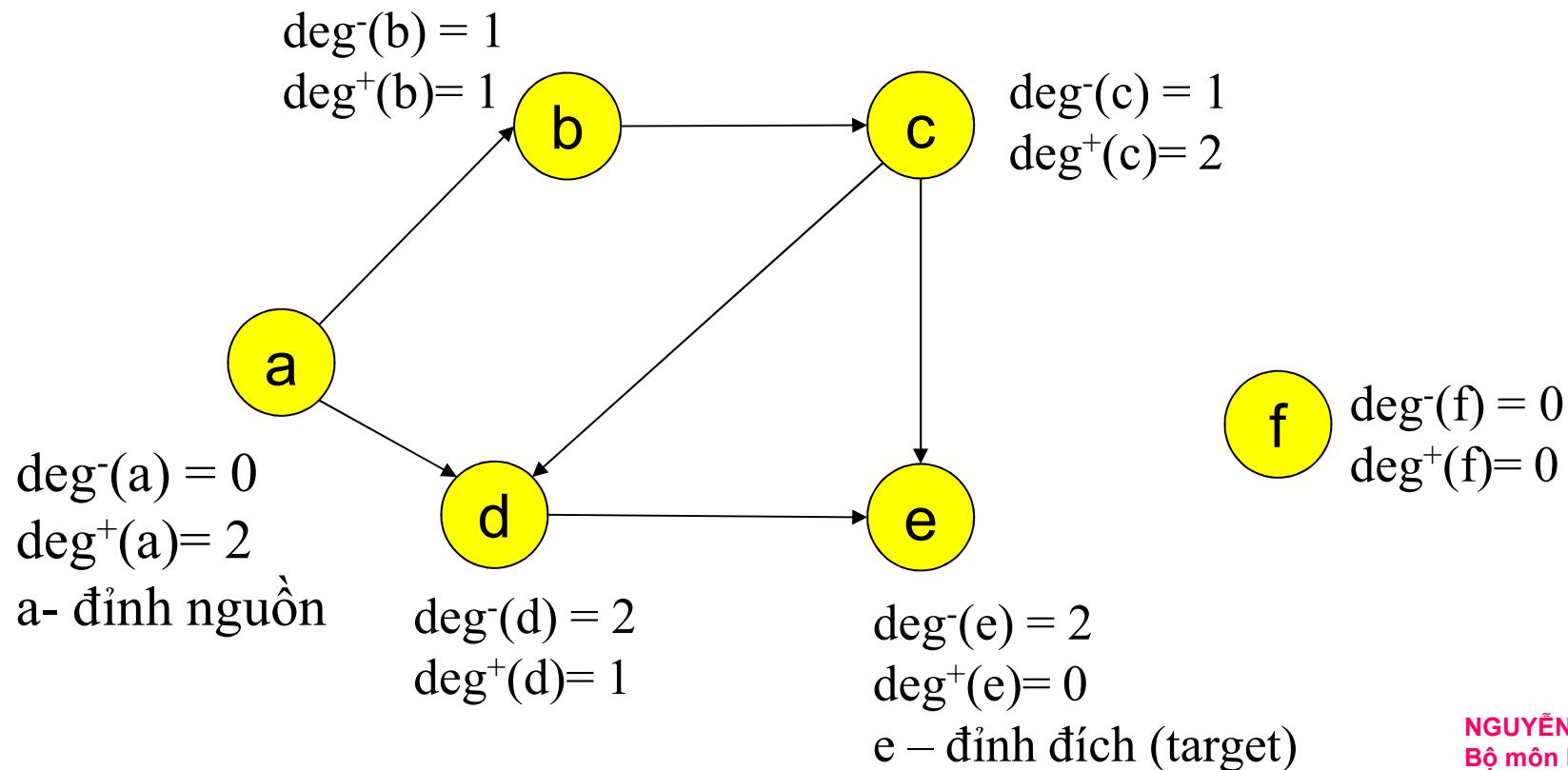
Chứng minh: Trong tổng ở trên mỗi cạnh $e=(u,v) \in E$ được tính hai lần: trong $\deg(u)$ và $\deg(v)$.

- **Hệ quả:** Trong một đồ thị vô hướng bất kỳ, số lượng đỉnh bậc lẻ (đỉnh có bậc là số lẻ) bao giờ cũng là số chẵn.

Bậc của đỉnh của đồ thị có hướng

Cho G là đồ thị có hướng, v là đỉnh của G :

- *Bán bậc vào (in-degree)* của v , $\deg^-(v)$, là số cạnh đi vào v .
- *Bán bậc ra (out-degree)* của v , $\deg^+(v)$, là số cạnh đi ra khỏi v .
- *Bậc* của v , $\deg(v) = \deg^-(v) + \deg^+(v)$, là tổng của bán bậc vào và bán bậc ra của v .



Định lý về các cái bắt tay có hướng Directed Handshaking Theorem

- **Định lý.** Giả sử G là đồ thị có hướng (có thể là đơn hoặc đa) với tập đỉnh V và tập cạnh E . Khi đó:

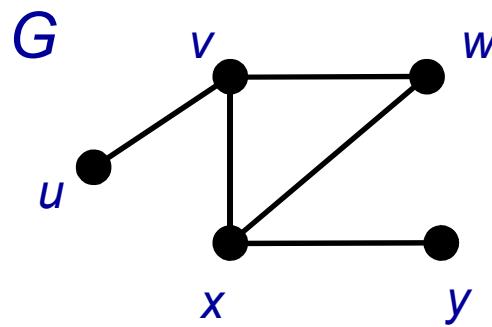
$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = \frac{1}{2} \sum_{v \in V} \deg(v) = |E|$$

- Chú ý là khái niệm bậc của đỉnh là không thay đổi cho dù ta xét đồ thị vô hướng hay có hướng.

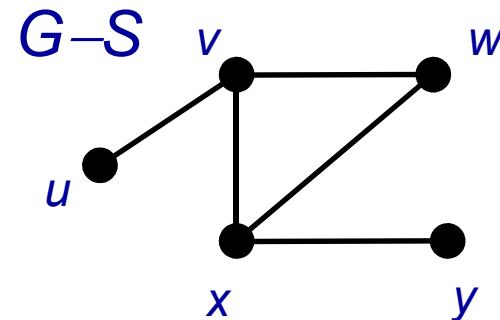
1.2.3. Loại bỏ đỉnh (The deletion of vertices)

Định nghĩa. Cho $G = (V, E)$ là đồ thị vô hướng. Giả sử $S \subseteq V$. Ta gọi việc **loại bỏ tập đỉnh S** khỏi đồ thị là việc loại bỏ tất cả các đỉnh trong S cùng các cạnh kề với chúng.

Như vậy nếu ký hiệu đồ thị thu được là $G-S$, ta có $G-S = \langle V-S \rangle$.

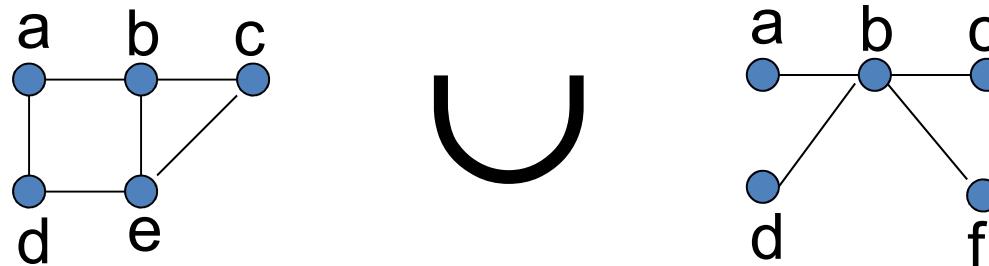


Giả sử $S=\{x, u\} \Rightarrow$



1.2.4. Hợp của hai đồ thị

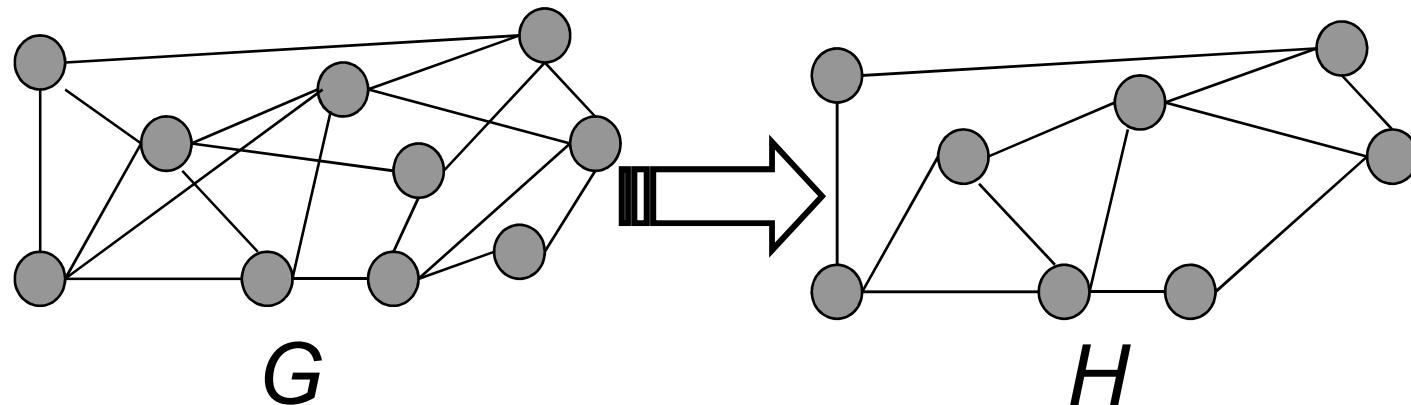
- Hợp $G_1 \cup G_2$ của hai đơn đồ thị $G_1 = (V_1, E_1)$ và $G_2 = (V_2, E_2)$ là đơn đồ thị $(V_1 \cup V_2, E_1 \cup E_2)$.



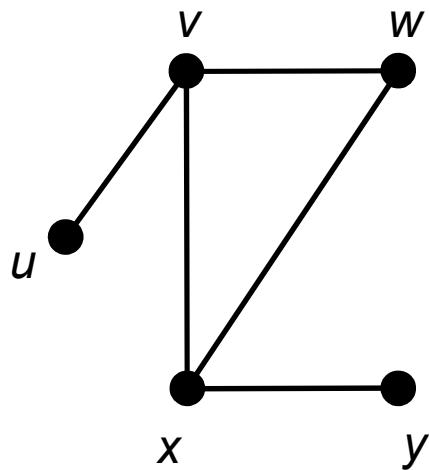
1.2.5. Đồ thị con (Subgraphs)

Định nghĩa. Đồ thị $H=(W,F)$ được gọi là đồ thị con của đồ thị $G=(V,E)$ nếu $W \subseteq V$ và $F \subseteq E$.

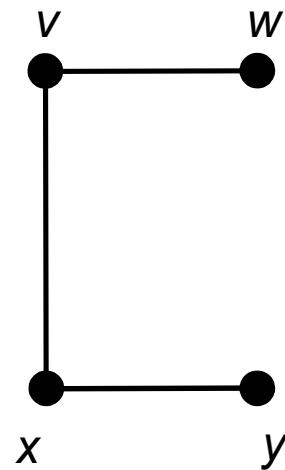
Ký hiệu: $H \subseteq G$.



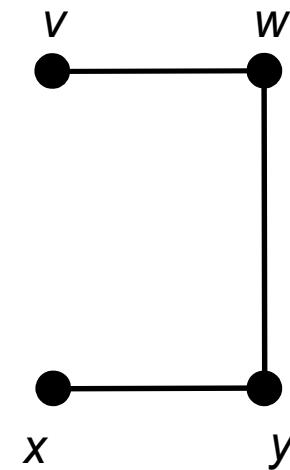
Ví dụ



G



$H \subseteq G$



$F \not\subseteq G$

1.2.6. Đồ thị con bao trùm (Spanning Subgraph)

Định nghĩa.

Đồ thị con $H \subseteq G$ được gọi là đồ thị con bao trùm của G nếu tập đỉnh của H là tập đỉnh của G : $V(H) = V(G)$.

1.2.7. Đường đi (Path)

- **Định nghĩa.** *Đường đi P độ dài n* từ đỉnh u đến đỉnh v, trong đó n là số nguyên dương, trên đồ thị $G=(V,E)$ là dãy

$$P: \quad x_0, x_1, \dots, x_{n-1}, x_n$$

trong đó $u = x_0$, $v = x_n$, $(x_i, x_{i+1}) \in E$, $i = 0, 1, 2, \dots, n-1$.

Đường đi nói trên còn có thể biểu diễn dưới dạng dãy các cạnh:

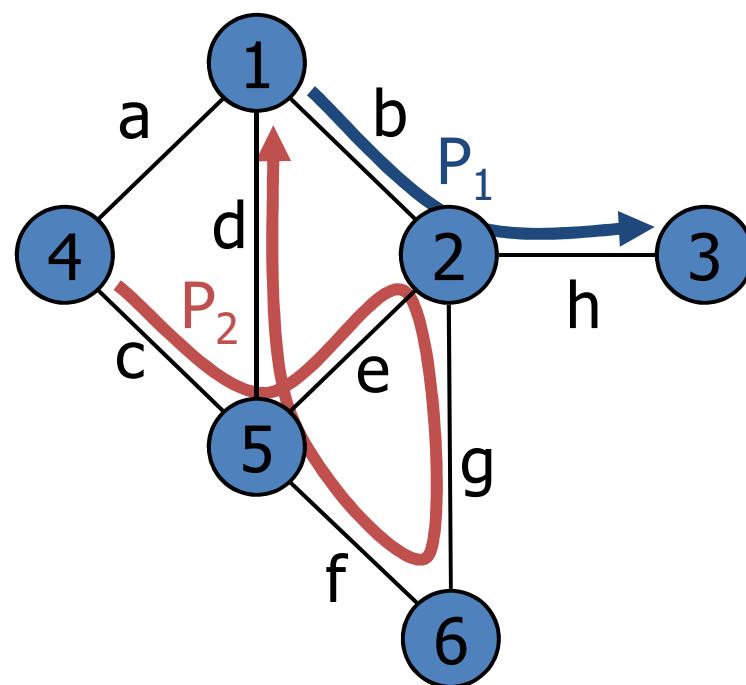
$$(x_0, x_1), (x_1, x_2), \dots, (x_{n-1}, x_n).$$

Đỉnh u được gọi là **đỉnh đầu**, còn đỉnh v được gọi là **đỉnh cuối** của đường đi P.

- Đường đi gọi là **đường đi đơn (simple path)** nếu không có đỉnh nào bị lặp lại trên nó.
- Đường đi gọi là **đường đi cơ bản (elementary path)** nếu không có cạnh nào bị lặp lại trên nó.

Đường đi (Path)

- $P_1 = (1, b, 2, h, 3)$ là đường đi đơn
- $P_2 = (4, c, \textcolor{red}{5}, e, 2, g, 6, f, \textcolor{red}{5}, d, 1)$ là đường đi nhưng không là đường đi đơn



1.2.7. Chu trình (Cycle)

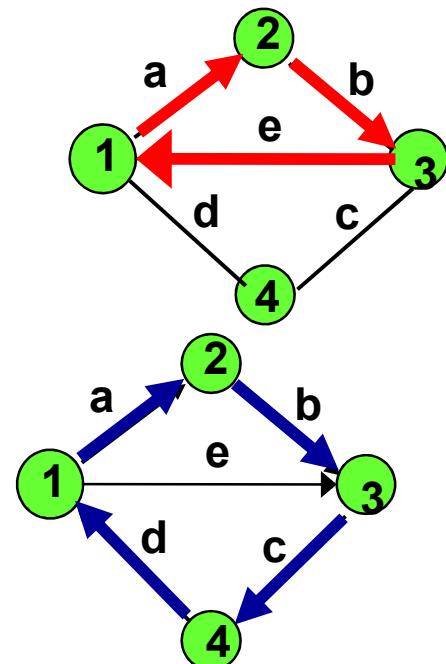
- **Chu trình:** là đường đi cơ bản có đỉnh đầu trùng với đỉnh cuối (tức là $u = v$).
- Chu trình được gọi là **đơn** nếu như ngoại trừ đỉnh đầu trùng với đỉnh cuối, không có đỉnh nào bị lặp lại.

Chu trình đơn

(1, 2, 3, 1) hay (1, a, 2, b, 3, e)

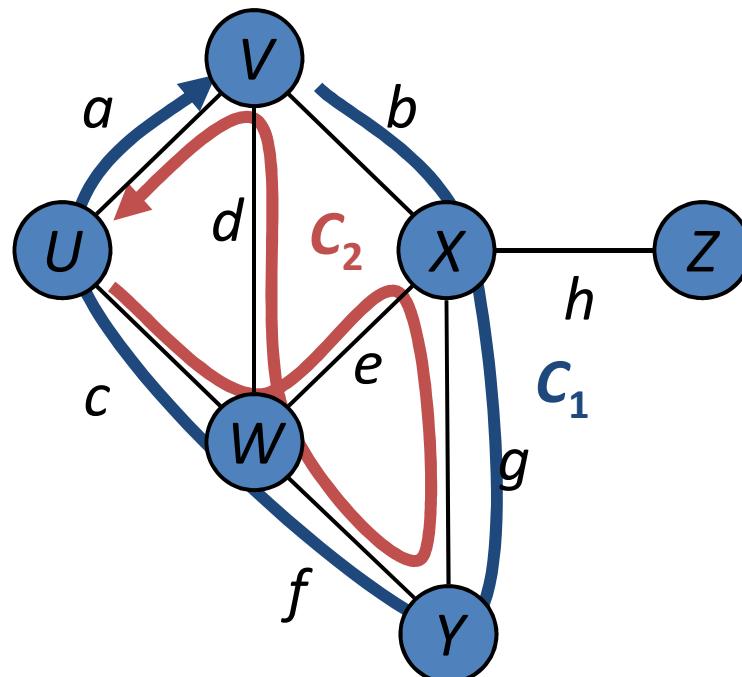
Chu trình đơn

(1, 2, 3, 4, 1) hay (1, a, 2, b, 3, c, 4, d, 1)



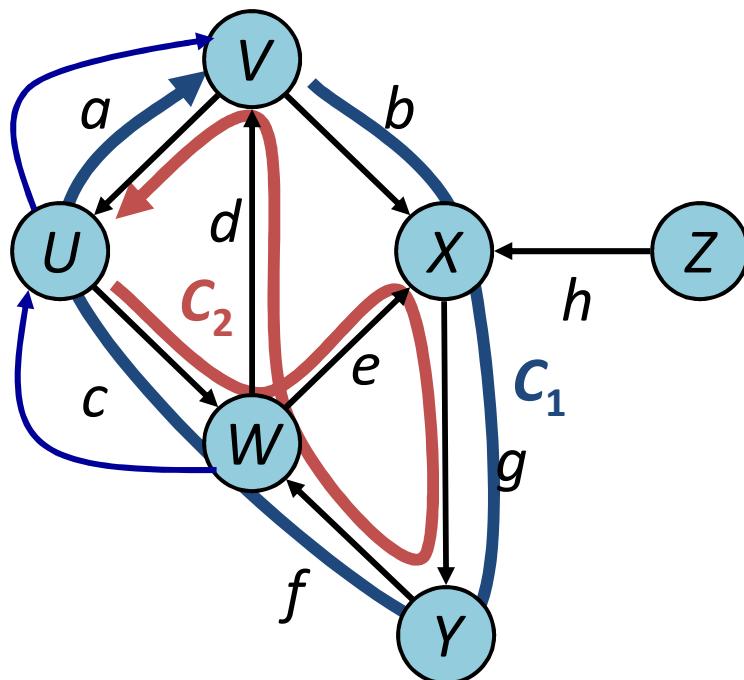
Ví dụ: Chu trình trên đồ thị vô hướng

- $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ là chu trình đơn
- $C_2 = (U, c, \textcolor{red}{W}, e, X, g, Y, f, \textcolor{red}{W}, d, V, a, U)$ là chu trình nhưng không là chu trình đơn



Ví dụ: Chu trình trên đồ thị có hướng

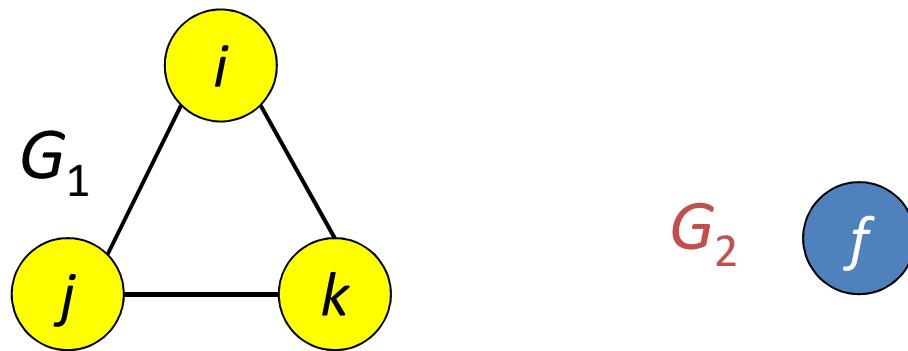
- $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ là chu trình đơn
- $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ là chu trình nhưng không là chu trình đơn



1.2.8. Tính liên thông (Connectedness)

- **Định nghĩa.** Đồ thị vô hướng được gọi là *liên thông* nếu luôn tìm được đường đi nối hai đỉnh bất kỳ của nó.

Ví dụ



- G_1 và G_2 là các đồ thị liên thông
- Đồ thị G bao gồm G_1 và G_2 không là đồ thị liên thông

Tính liên thông (Connectedness)

- **Mệnh đề:** Luôn tìm được đường đi đơn nối hai đỉnh bất kỳ của đồ thị vô hướng liên thông.

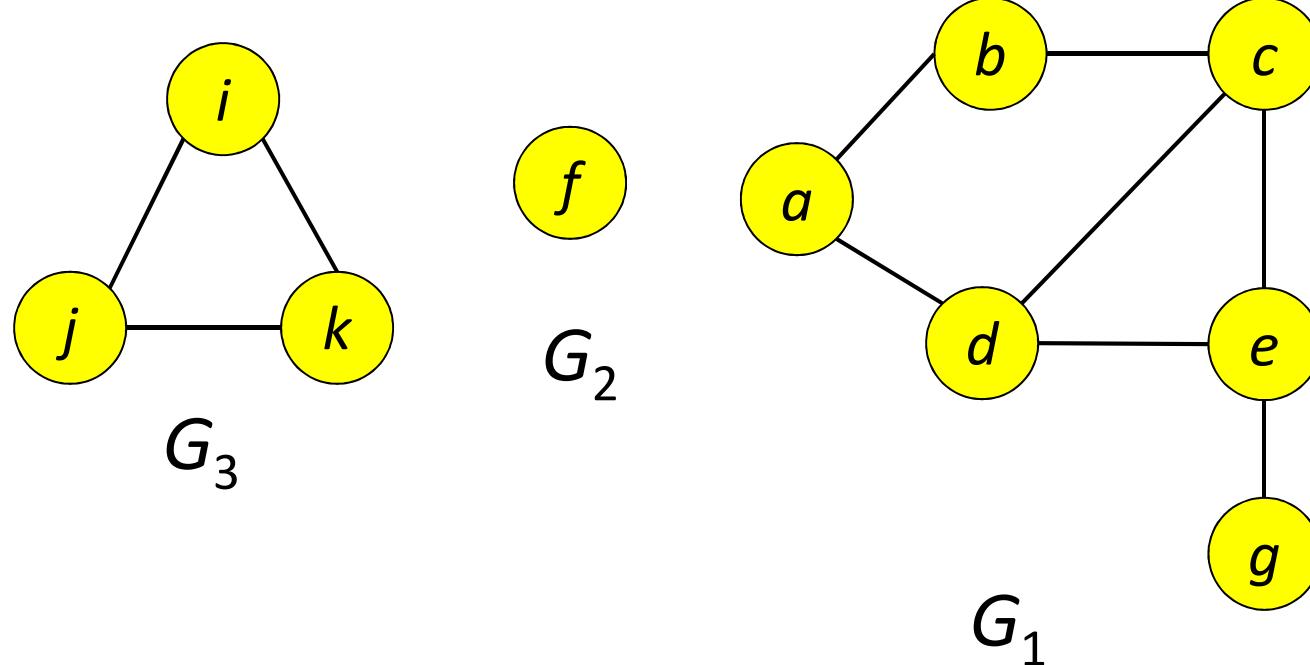
Chứng minh.

Theo định nghĩa tính liên thông của đồ thị: luôn tìm được đường đi nối hai đỉnh bất kỳ của đồ thị liên thông. Gọi P là đường đi ngắn nhất nối hai đỉnh u và v . Rõ ràng P phải là đường đi đơn.

Đường đi gọi là **đường đi đơn** nếu không có đỉnh nào bị lặp lại trên nó.

Tính liên thông (Connectedness)

- **Thành phần liên thông** (*Connected component*): Đồ thị con liên thông cực đại của đồ thị vô hướng G được gọi là thành phần liên thông của nó.
- **Ví dụ:** Đồ thị G có 3 thành phần liên thông G_1 , G_2 , G_3



Ví dụ

Cho G là đồ thị vô hướng $n \geq 2$ đỉnh. Biết rằng
 $d(G) = \min \{\deg(v): v \in V\} \geq (n-1)/2$.
Chứng minh rằng G liên thông.

Chứng minh.

Phản chứng. Giả sử G không liên thông, khi đó theo giả thiết:

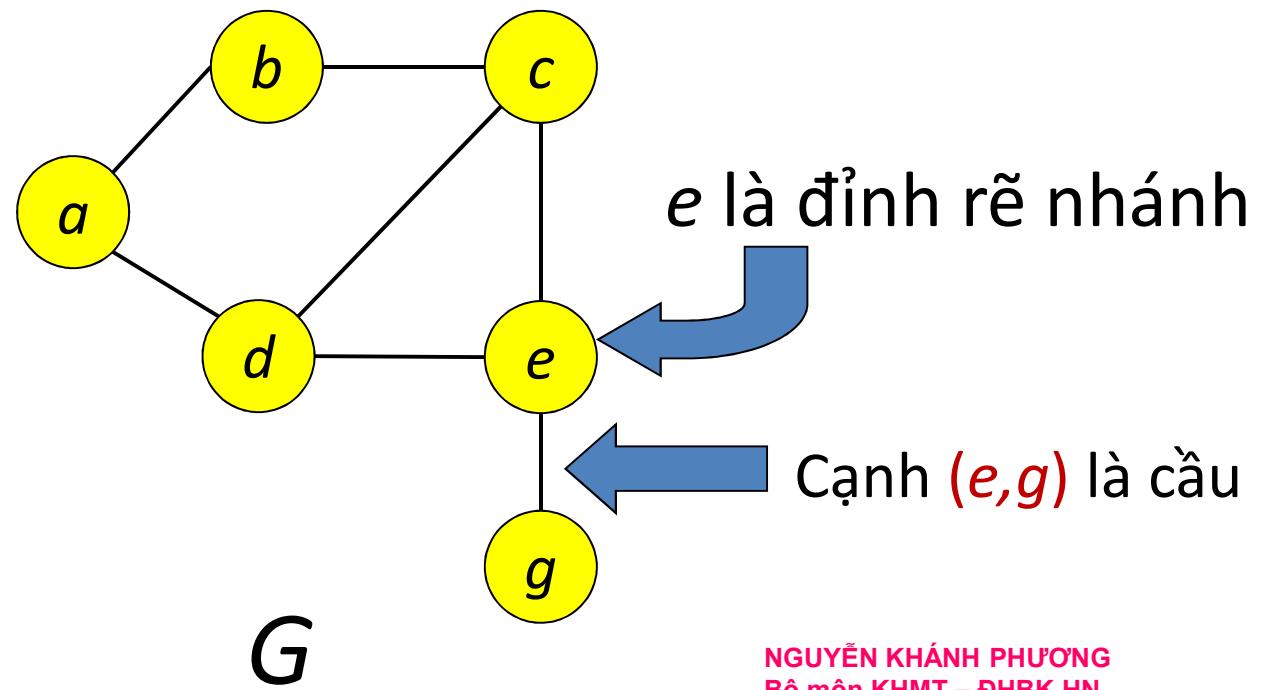
$$d(G) \geq (n-1)/2,$$

nên mỗi thành phần liên thông phải chứa ít ra
 $(n-1)/2+1 = (n+1)/2$ đỉnh.

Suy ra đồ thị có ít ra $(n+1)$ đỉnh?!

Đỉnh rẽ nhánh và cầu (Connectedness)

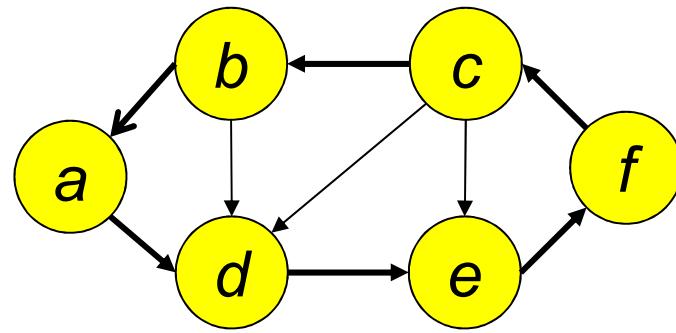
- *Đỉnh rẽ nhánh (cut vertex/ articulation point):* là đỉnh mà việc loại bỏ nó làm tăng số thành phần liên thông của đồ thị
- *Cầu (bridge):* Cạnh mà việc loại bỏ nó làm tăng số thành phần liên thông của đồ thị .
- **Ví dụ:**



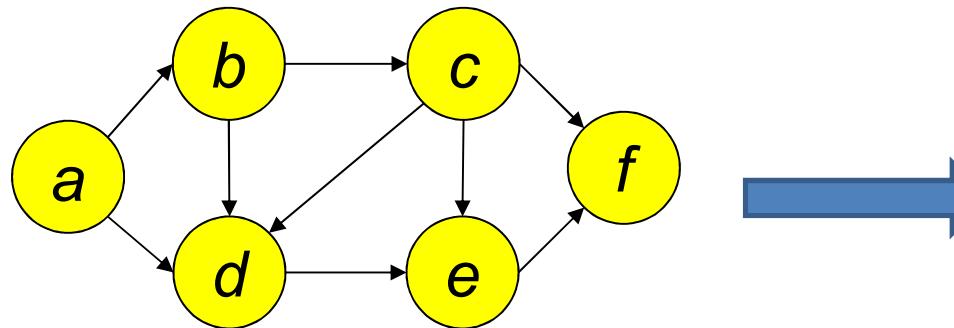
Tính liên thông của Đồ thị có hướng

- Đồ thị có hướng được gọi là **liên thông mạnh** (*strongly connected*) nếu như luôn tìm được đường đi nối hai đỉnh bất kỳ của nó.
- Đồ thị có hướng được gọi là **liên thông yếu** (*weakly connected*) nếu như đồ thị vô hướng thu được từ nó bởi việc bỏ qua hướng của tất cả các cạnh của nó là đồ thị vô hướng liên thông.
- Để thấy là nếu G là liên thông mạnh thì nó cũng là liên thông yếu, nhưng điều ngược lại không luôn đúng.

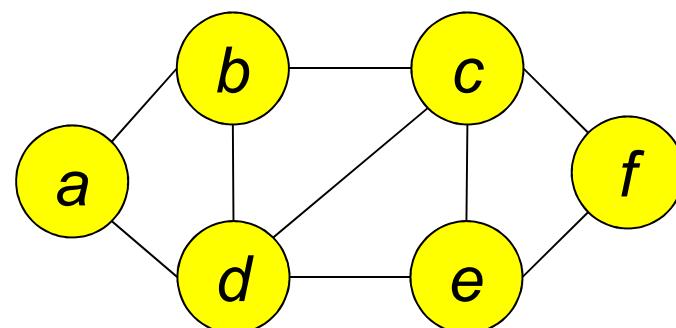
Ví dụ: đồ thị liên thông mạnh/yếu



Đồ thị liên thông mạnh



Đồ thị liên thông yếu



1. Đồ thị và cách biểu diễn đồ thị

1.1. Đồ thị vô hướng và có hướng

1.2. Một số khái niệm cơ bản trên đồ thị

1.3. Một số dạng đồ thị đặc biệt

1.4. Biểu diễn đồ thị

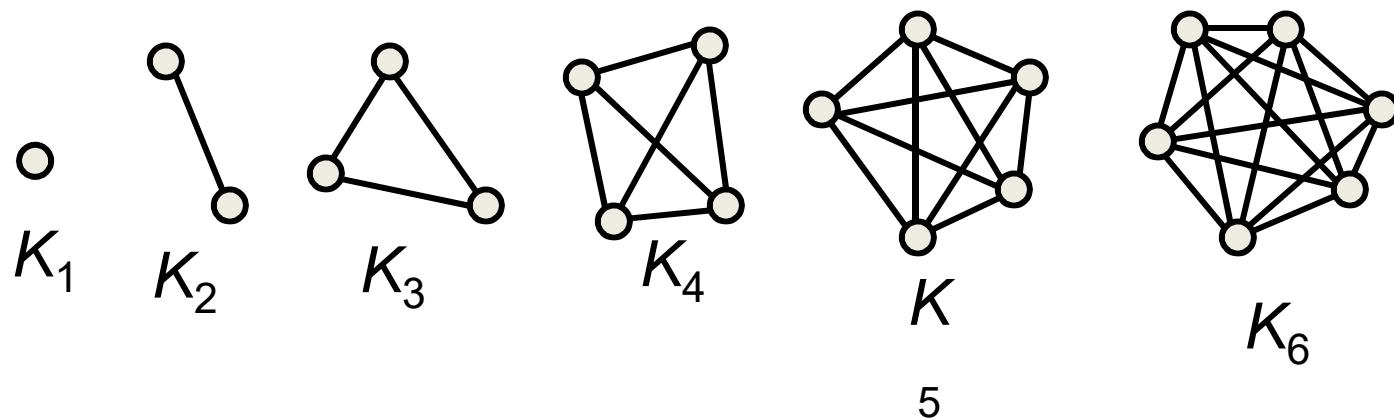


Một số dạng đơn đồ thị vô hướng đặc biệt

- Đồ thị đầy đủ (Complete graphs) K_n
- Chu trình (Cycles) C_n
- Bánh xe (Wheels) W_n
- n -Cubes Q_n
- Đồ thị hai phía (Bipartite graphs)
- Đồ thị hai phía đầy đủ (Complete bipartite graphs) $K_{m,n}$
- Đồ thị chính qui
- Cây và rừng

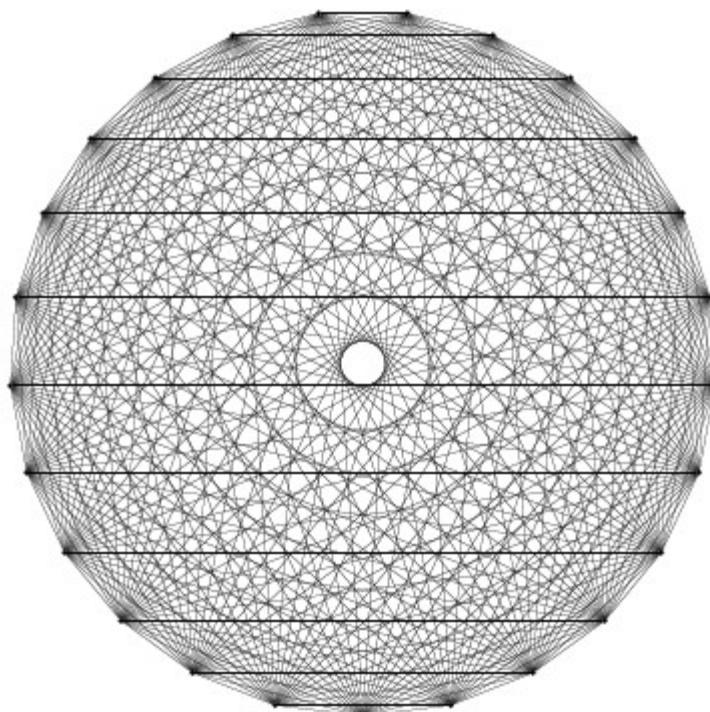
1.3.1. Đồ thị đầy đủ (Complete Graphs)

- Với $n \in \mathbb{N}$, đồ thị đầy đủ n đỉnh, K_n , là đơn đồ thị vô hướng với n đỉnh trong đó giữa hai đỉnh bất kỳ luôn có cạnh nối: $\forall u, v \in V: u \neq v \Leftrightarrow (u, v) \in E$.



Để ý là K_n có $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ cạnh.

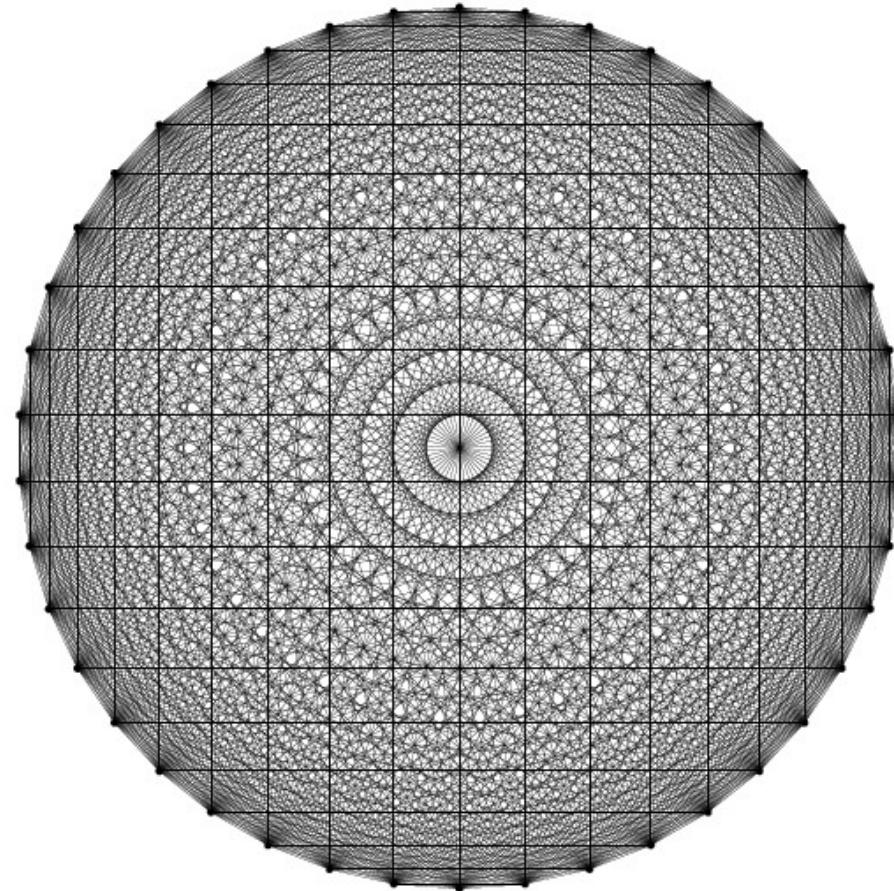
Đồ thị đầy đủ (Complete Graphs)



K_{25}

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Đồ thị đầy đủ (Complete Graphs)

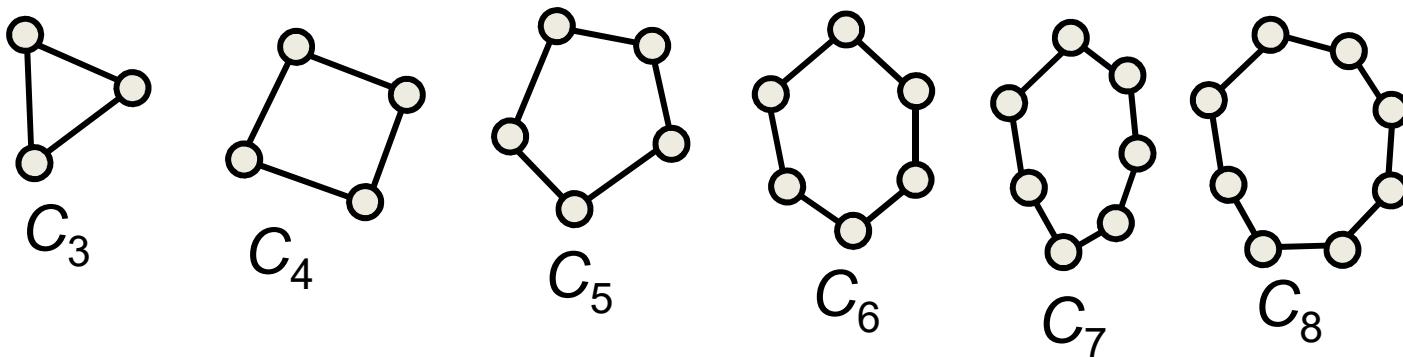


K_{42}

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

1.3.2. Chu trình (Cycles)

Giả sử $n \geq 3$. Chu trình n đỉnh, C_n , là đơn đồ thị vô hướng với $V = \{v_1, v_2, \dots, v_n\}$ và $E = \{(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n), (v_n, v_1)\}$.

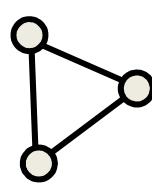
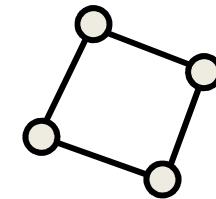
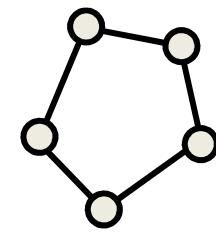
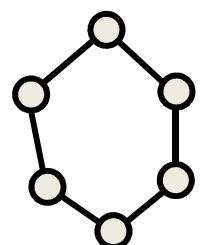
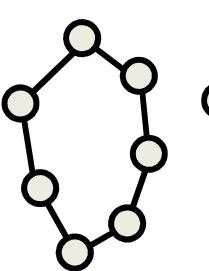
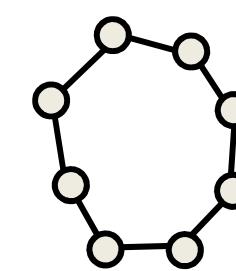
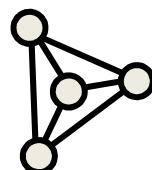
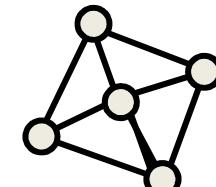
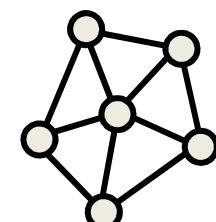
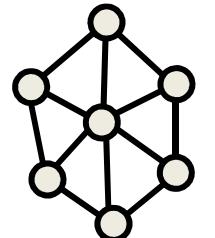
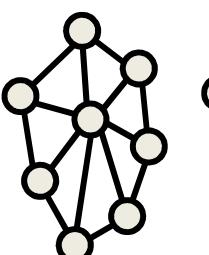
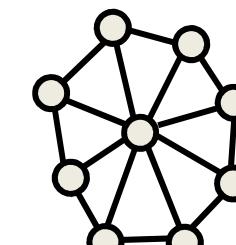


Có bao nhiêu cạnh trong C_n ?

1.3.3. Bánh xe (Wheels)

- Với $n \geq 3$, bánh xe W_n , là đơn đồ thị vô hướng thu được bằng cách bổ sung vào chu trình C_n một đỉnh v_{hub} và n cạnh nối

$$\{(v_{\text{hub}}, v_1), (v_{\text{hub}}, v_2), \dots, (v_{\text{hub}}, v_n)\}.$$

 C_3  C_4  C_5  C_6  C_7  C_8  W_3  W_4  W_5  W_6  W_7  W_8

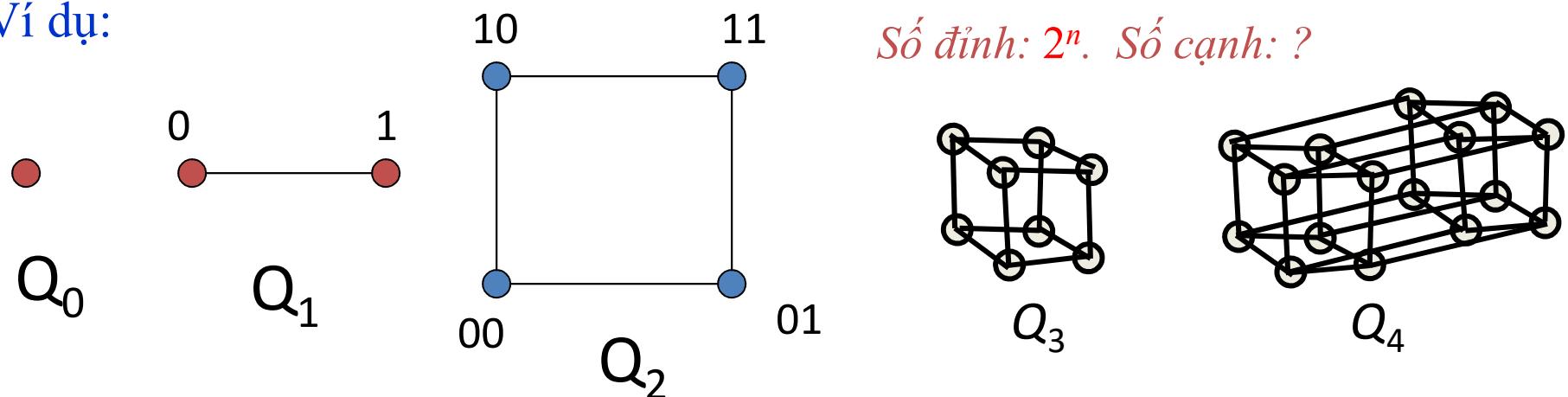
Có bao nhiêu cạnh trong W_n ?

1.3.4. N-cubes: siêu cúp

N-cubes (Q_n): các đỉnh được biểu diễn bởi 2^n xâu nhị phân độ dài n . Hai đỉnh kề nhau khi và chỉ khi hai xâu nhị phân tương ứng với chúng chỉ khác nhau đúng 1 bit

Ví dụ:

Số đỉnh: 2^n . Số cạnh: ?



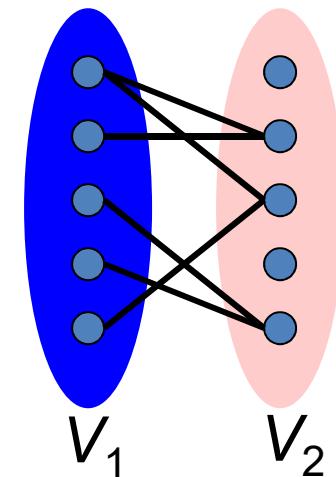
Với $n \in \mathbb{N}$, siêu cúp Q_{n+1} là đơn đồ thị vô hướng gồm hai bản sao của Q_n trong đó các đỉnh tương ứng được nối với nhau:

- $Q_0 = \{\{v_0\}, \emptyset\}$ (dù nhất 1 đỉnh, không có cạnh)
- $n \in \mathbb{N}$: nếu $Q_n = (V, E)$ với $V = \{v_1, \dots, v_a\}$, $E = \{e_1, \dots, e_b\}$, thì $Q_{n+1} = (V \cup \{v_1', \dots, v_a'\}, E \cup \{e_1', \dots, e_b'\} \cup \{\{v_1, v_1'\}, \{v_2, v_2'\}, \dots, \{v_a, v_a'\}\})$

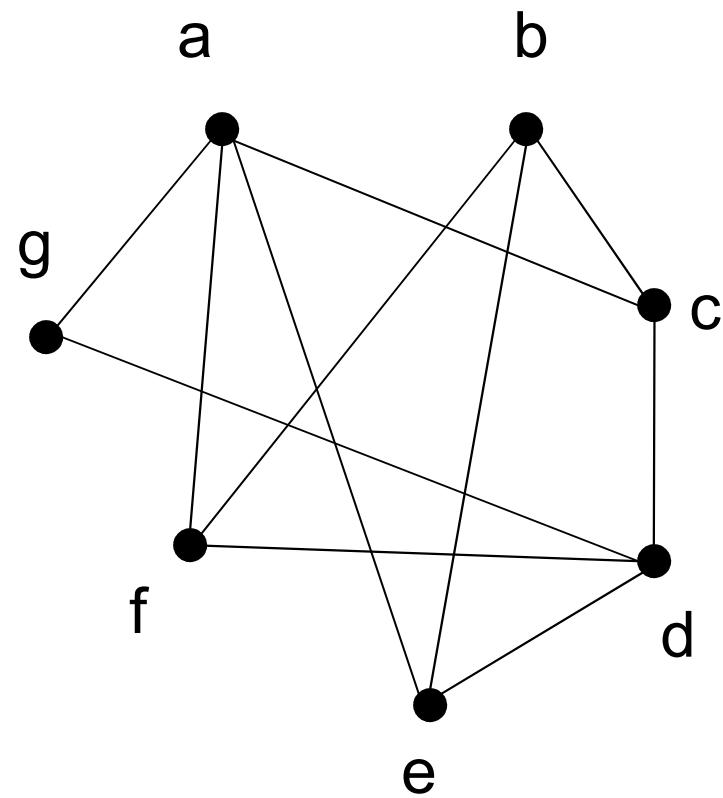
1.3.5. Đồ thị hai phía (Bipartite Graphs)

- **Định nghĩa.** Đồ thị $G=(V,E)$ là hai phía khi và chỉ khi $V = V_1 \cup V_2$ với $V_1 \cap V_2 = \emptyset$ và $\forall e \in E: \exists v_1 \in V_1, v_2 \in V_2: e = (v_1, v_2)$.
- **Bằng lời:** Có thể phân hoạch tập đỉnh thành hai tập sao cho mỗi cạnh nối hai đỉnh thuộc hai tập khác nhau.

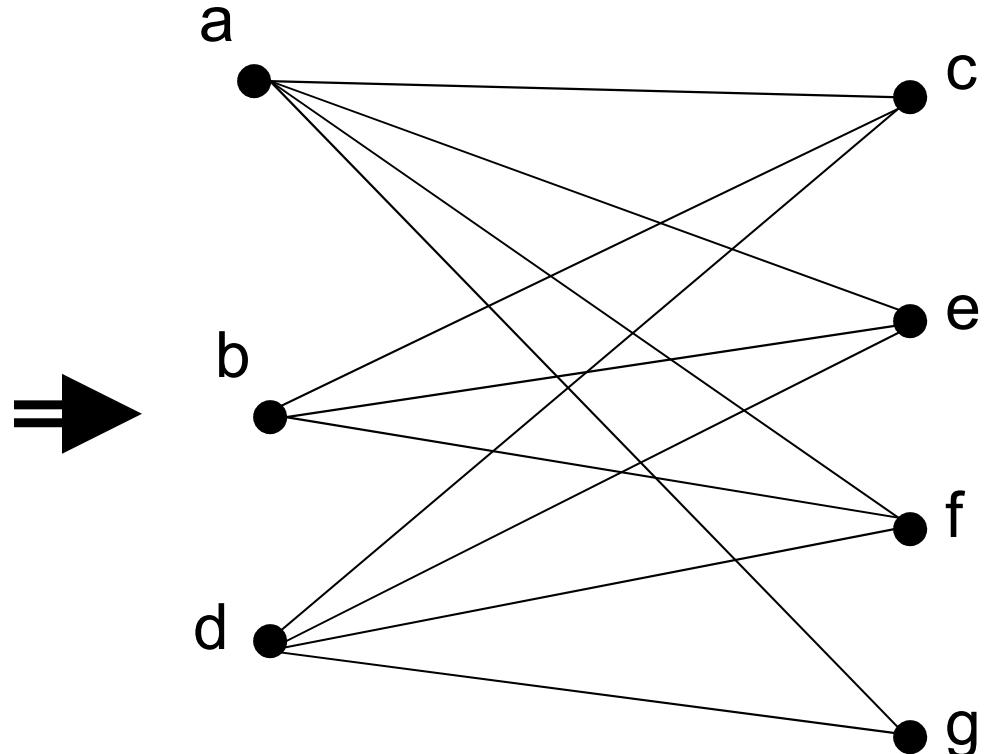
Định nghĩa này là chung cho cả đơn lans đa đồ thị vô hướng, có hướng.



Ví dụ. Đồ thị G dưới đây là đồ thị hai phái?



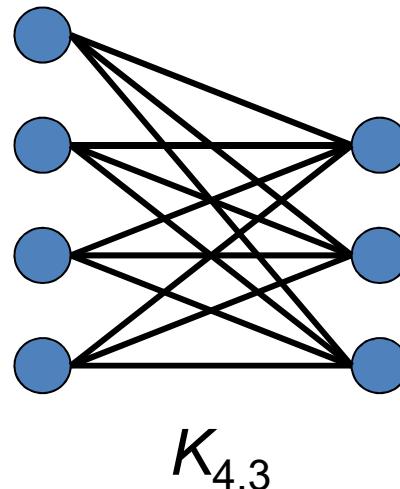
G



Yes !

Đồ thị hai phía đầy đủ (Complete Bipartite Graphs)

- Với $m, n \in \mathbb{N}$, đồ thị hai phía đầy đủ $K_{m,n}$ là đồ thị hai phía trong đó $|V_1| = m$, $|V_2| = n$, và
$$E = \{(v_1, v_2) | v_1 \in V_1 \text{ và } v_2 \in V_2\}.$$
- $K_{m,n}$ có m đỉnh ở tập bên trái, n đỉnh ở tập bên phải, và mỗi đỉnh ở phần bên trái được nối với mỗi đỉnh ở phần bên phải.

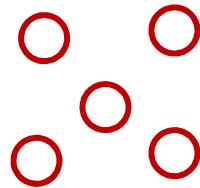


$K_{m,n}$ có _____ đỉnh
và _____ cạnh.

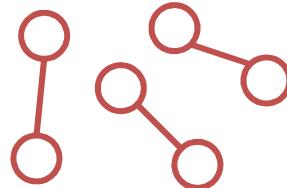
1.3.6. Đồ thị chính qui (*r-regular graph*)

- **Định nghĩa.** Đồ thị G được gọi là đồ thị chính qui bậc r nếu tất cả các đỉnh của nó có bậc bằng r .

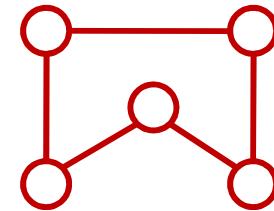
Ví dụ:



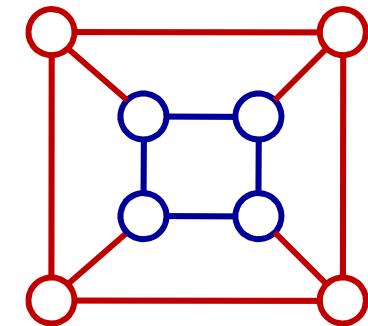
Đồ thị chính qui
bậc 0



Đồ thị chính qui
bậc 1



Đồ thị chính qui
bậc 2

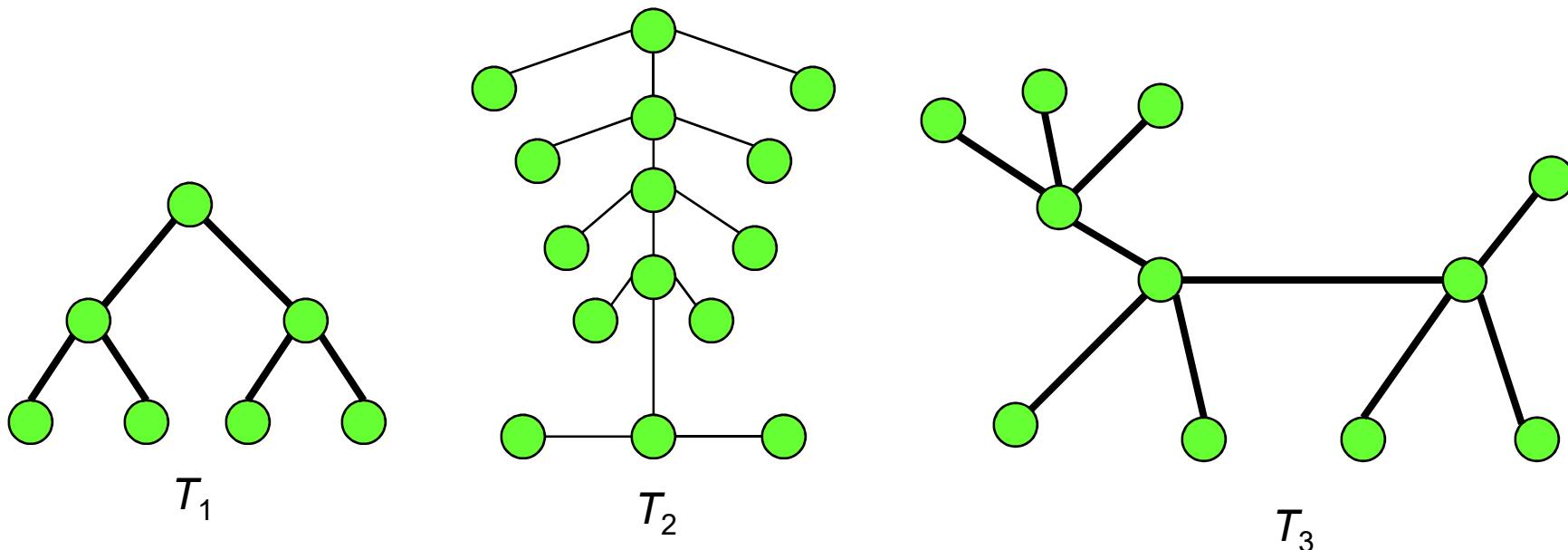


Đồ thị chính qui
bậc 3

1.3.7. Cây và rừng (Tree and Forest)

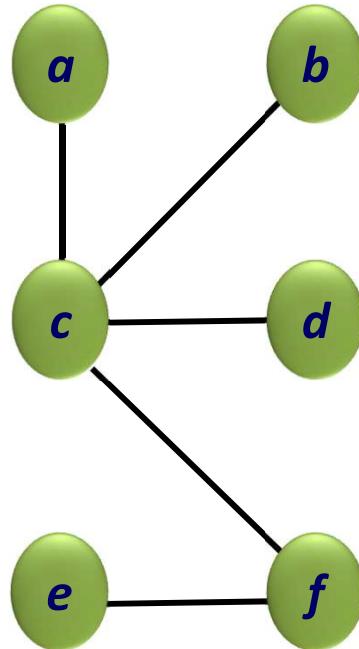
- **Cây:** đồ thị vô hướng liên thông không có chu trình.
- **Rừng:** đồ thị không có chu trình.

Như vậy, rừng là đồ thị mà mỗi thành phần liên thông của nó là một cây.

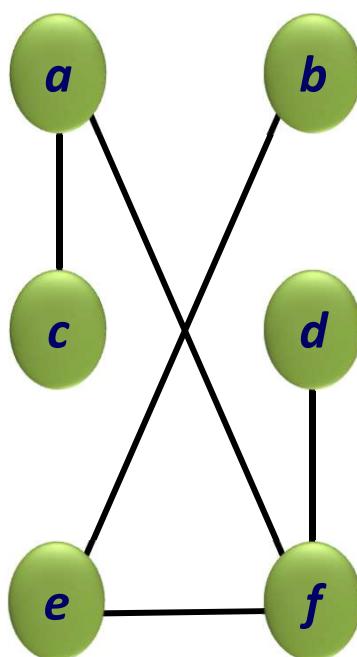


Rừng F gồm 3 cây T_1 , T_2 , T_3

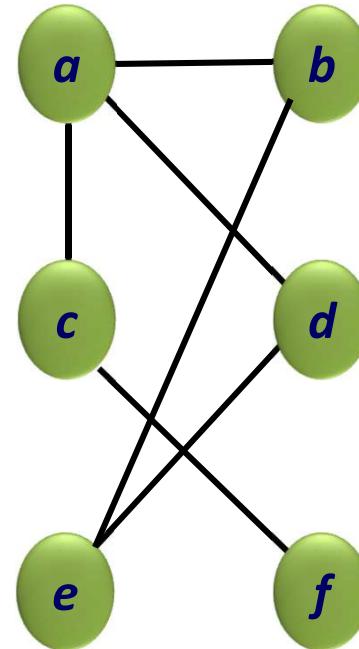
Ví dụ: Cây?



G_1

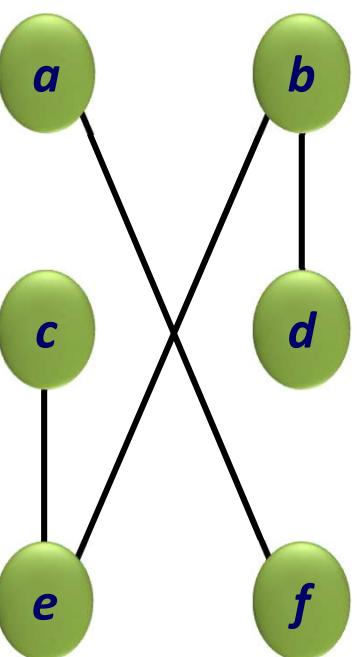


G_2



G_3

Chứa chu trình



G_4

Không liên thông

G_1 , G_2 là cây

G_3 , G_4 không là cây

G_4 là rừng

Cây (Tree)

- Một số tính chất của cây
 1. Hai đỉnh bất kỳ được nối với nhau bởi đúng 1 đường đi đơn.
 2. Mỗi cạnh của cây đều là cầu.
 3. Thêm vào một cạnh ta thu được đúng một chu trình
 4. Số lượng cạnh là $n - 1$
 - Cây là đồ thị liên thông ít cạnh nhất.
- Cây khung/bao trùm (Spanning tree):
 - Cho $G = (V, E)$, Cây khung của G là đồ thị con liên thông không chứa chu trình bao gồm tất cả các đỉnh trong V .
 - Cây khung không nhất thiết là duy nhất.

1. Đồ thị và cách biểu diễn đồ thị

1.1. Đồ thị vô hướng và có hướng

1.2. Một số khái niệm cơ bản trên đồ thị

1.3. Một số dạng đồ thị đặc biệt

1.4. Biểu diễn đồ thị



1.4. Biểu diễn đồ thị

- Có nhiều cách biểu diễn. Việc lựa chọn cách biểu diễn phụ thuộc vào từng bài toán cụ thể cần xét, thuật toán cụ thể cần cài đặt.
- Có hai vấn đề chính cần quan tâm khi lựa chọn cách biểu diễn:
 - Bộ nhớ mà cách biểu diễn đó đòi hỏi
 - Thời gian cần thiết để trả lời các truy vấn thường xuyên đối với đồ thị trong quá trình xử lý đồ thị:
 - Chẳng hạn:
 - Có cạnh nối hai đỉnh u, v ?
 - Liệt kê các đỉnh kề của đỉnh v ?

1.4. Biểu diễn đồ thị

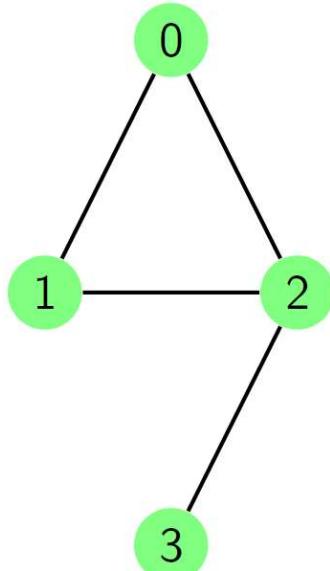
1.4.1. Ma trận kề

1.4.2. Danh sách kề

1.4.3. Danh sách các cạnh

1.4.4. Ma trận trọng số

1.4.1. Biểu diễn đồ thị bởi ma trận kề



| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |

```
bool adj [4] [4];
adj [0] [1] = true;
adj [0] [2] = true;
adj [1] [0] = true;
adj [1] [2] = true;
adj [2] [0] = true;
adj [2] [1] = true;
adj [2] [3] = true;
adj [3] [2] = true;
```

1.4. Biểu diễn đồ thị

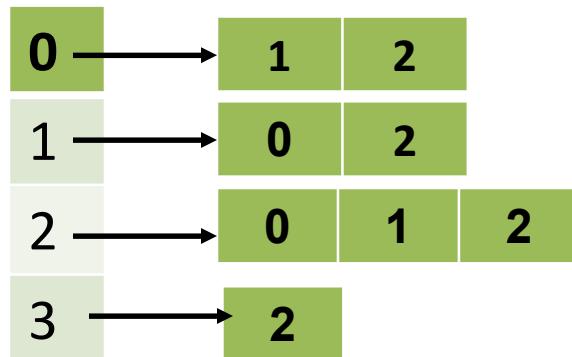
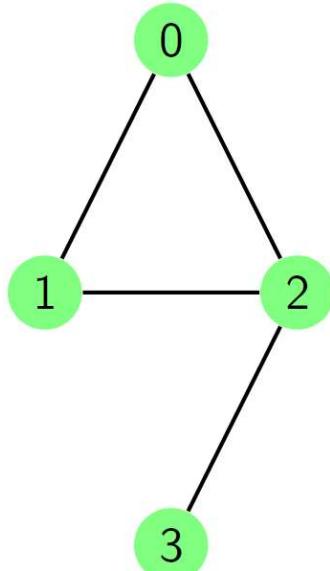
1.4.1. Ma trận kề

1.4.2. Danh sách kề

1.4.3. Danh sách các cạnh

1.4.4. Ma trận trọng số

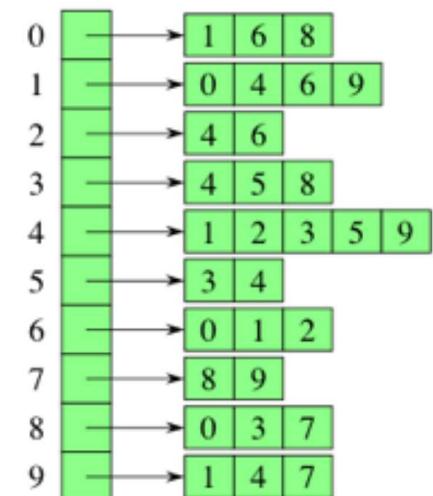
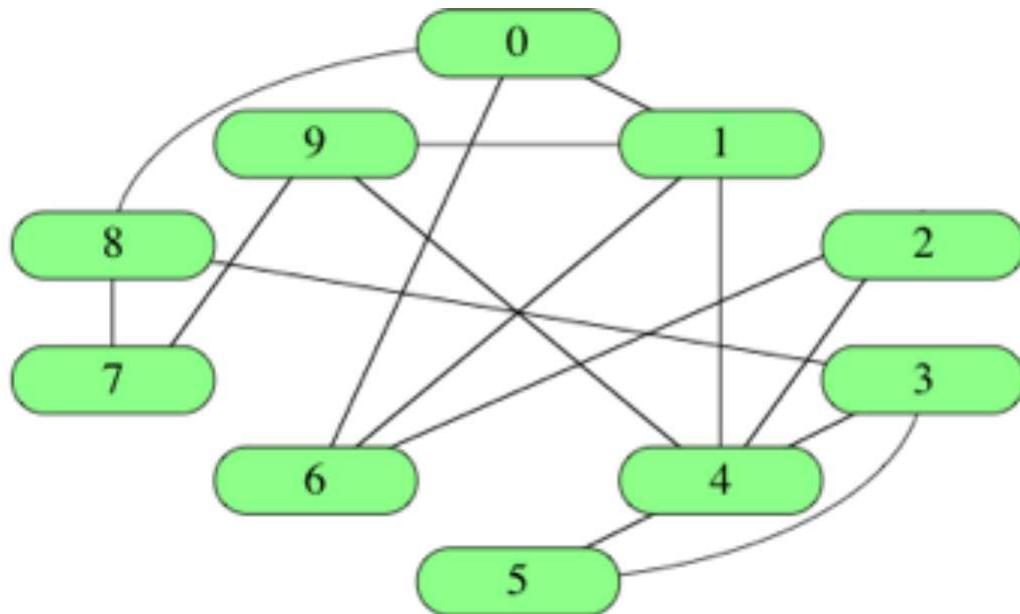
1.4.2. Biểu diễn đồ thị bởi danh sách kề



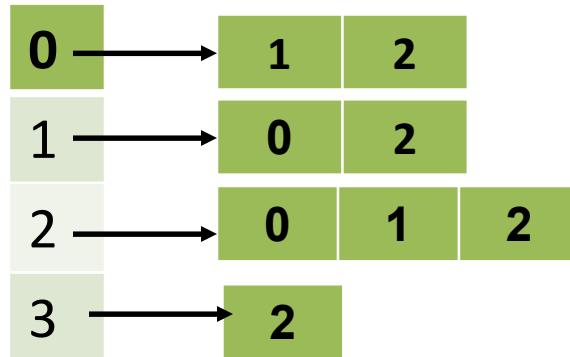
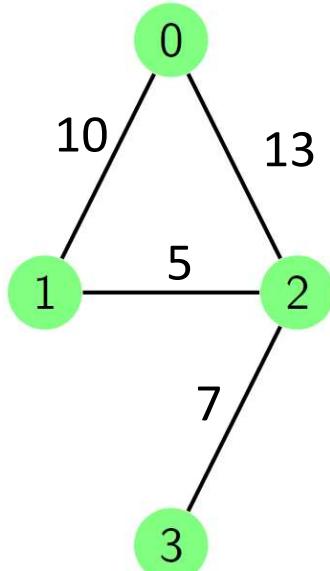
```
vector<int> adj[4];
adj[0].push_back(1);
adj[0].push_back(2);
adj[1].push_back(0);
adj[1].push_back(2);
adj[2].push_back(0);
adj[2].push_back(1);
adj[2].push_back(3);
adj[3].push_back(2);
```

1.4.2. Biểu diễn đồ thị bởi danh sách kề

Bài tập: viết sourcecode biểu diễn đồ thị sau bởi danh sách kề



1.4.2. Biểu diễn đồ thị bởi danh sách kề



```
vector <pair <int, int> > adj[4];
//hoặc list <pair <int, int> > adj[4];
adj[0].push_back(make_pair(1, 10));
adj[1].push_back(make_pair(0, 10)); //đồ thị vô hướng
adj[0].push_back(make_pair(2, 13));
adj[2].push_back(make_pair(0, 13)); //đồ thị vô hướng
adj[1].push_back(make_pair(2, 5));
adj[2].push_back(make_pair(1, 5)); //đồ thị vô hướng
adj[2].push_back(make_pair(3, 7));
adj[3].push_back(make_pair(2, 7)); //đồ thị vô hướng
```

```
typedef pair <int, int> iPair;
vector <pair <int, int> > *adj; //hoặc list <pair <int, int> > *adj;
adj = new vector <iPair> [4]; // hoặc adj = new list <iPair> [4];
```

1.4. Biểu diễn đồ thị

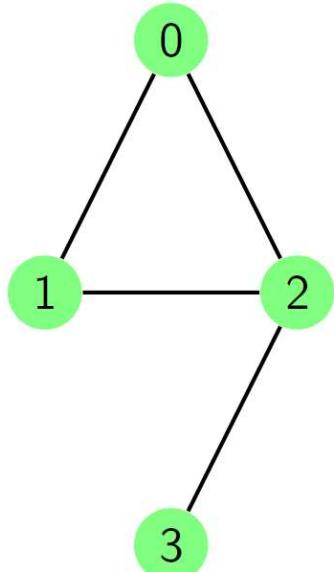
1.4.1. Ma trận kề

1.4.2. Danh sách kề

1.4.3. Danh sách các cạnh

1.4.4. Ma trận trọng số

1.4.3. Biểu diễn đồ thị bởi danh sách cạnh



(0, 1),
(0, 2),
(1, 2),
(2, 3)

```
vector<pair<int , int> > edges;  
edges.push_back(make_pair(0 , 1));  
edges.push_back(make_pair(0 , 2));  
edges.push_back(make_pair(1 , 2));  
edges.push_back(make_pair(2 , 3));
```

So sánh phân tích chi phí

| | Danh sách kề | Ma trận kề | Danh sách cạnh |
|---------------------------------|--------------|------------|----------------|
| Bộ nhớ lưu trữ | $O(V + E)$ | $O(V ^2)$ | $O(E)$ |
| Thêm đỉnh | $O(1)$ | $O(V ^2)$ | $O(1)$ |
| Thêm cạnh | $O(1)$ | $O(1)$ | $O(1)$ |
| Xóa đỉnh | $O(E)$ | $O(V ^2)$ | $O(E)$ |
| Xóa cạnh | $O(E)$ | $O(1)$ | $O(E)$ |
| Truy vấn: u, v có kề nhau không | $O(V)$ | $O(1)$ | $O(E)$ |

1.4. Biểu diễn đồ thị

1.4.1. Ma trận kề

1.4.2. Danh sách kề

1.4.3. Danh sách các cạnh

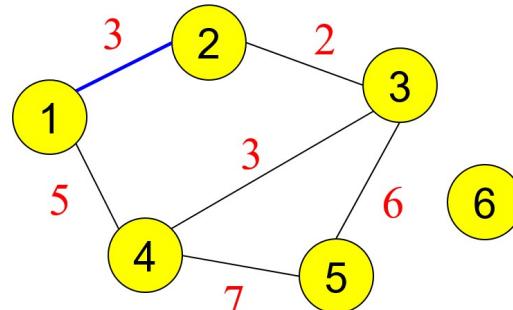
1.4.4. Ma trận trọng số

1.4.4. Ma trận trọng số

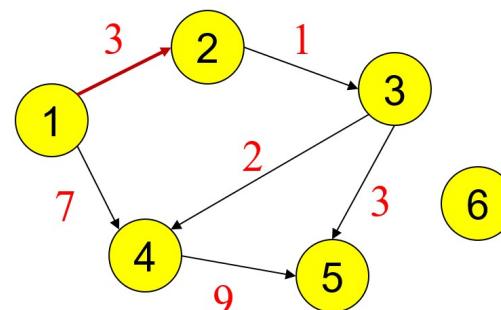
Trong trường hợp đồ thị có trọng số trên cạnh, thay vì ma trận kề, để biểu diễn đồ thị ta sử dụng **ma trận trọng số** $C = c[i, j]$, $i, j = 1, 2, \dots, n$,

với $c[i, j] = \begin{cases} c(i, j), & \text{nếu } (i, j) \in E \\ \theta, & \text{nếu } (i, j) \notin E, \end{cases}$

trong đó θ là giá trị đặc biệt để chỉ ra một cặp (i, j) không là cạnh, tùy từng trường hợp cụ thể, có thể được đặt bằng một trong các giá trị sau: 0, $+\infty$, $-\infty$.



$$\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 3 & 0 & 5 & 0 & 0 \\ 2 & 3 & 0 & 2 & 0 & 0 & 0 \\ 3 & 0 & 2 & 0 & 3 & 6 & 0 \\ 4 & 5 & 0 & 3 & 0 & 7 & 0 \\ 5 & 0 & 0 & 6 & 7 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$



$$\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 3 & 0 & 7 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 2 & 3 & 0 \\ 4 & 0 & 0 & 0 & 0 & 9 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Nội dung

1. Đồ thị và cách biểu diễn đồ thị

2. Duyệt đồ thị

3. Một số bài toán cơ bản

2. Duyệt đồ thị

2.1. Tìm kiếm theo chiều rộng

2.2. Tìm kiếm theo chiều sâu

2.3. Một số ứng dụng

2. Duyệt đồ thị

- Duyệt đồ thị: Graph Searching hoặc Graph Traversal
 - Duyệt qua mỗi đỉnh và mỗi cạnh của đồ thị
- Ứng dụng:
 - Cần để khảo sát các tính chất của đồ thị
 - Là thành phần cơ bản của nhiều thuật toán trên đồ thị
- Hai thuật toán duyệt cơ bản:
 - Tìm kiếm theo chiều rộng (Breadth First Search – BFS)
 - Tìm kiếm theo chiều sâu (Depth First Search – DFS)

2. Duyệt đồ thị

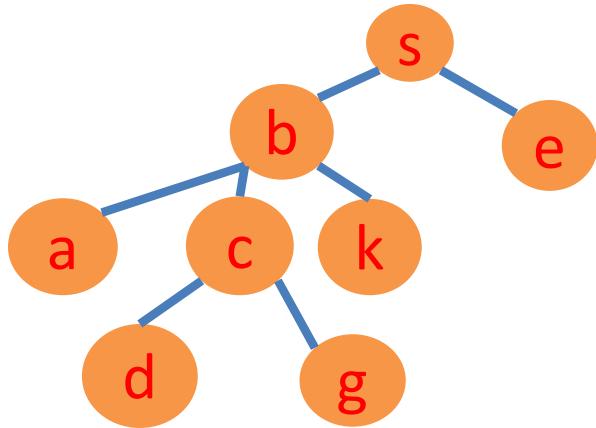
2.1. Tìm kiếm theo chiều rộng

2.2. Tìm kiếm theo chiều sâu

2.3. Một số ứng dụng

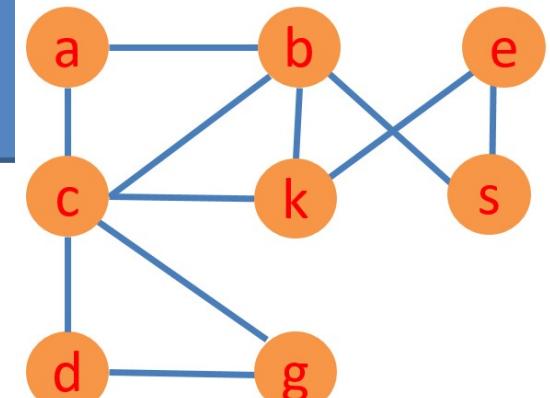
Breadth First Search

- Cho
 - Đồ thị $G=(V,E)$
 - 1 đỉnh nguồn s
- Thuật toán tìm kiếm theo chiều rộng sẽ đi qua các cạnh của G để thăm tất cả các đỉnh đạt được từ s .
- Với mỗi đỉnh v đạt được từ s , đường đi trên cây BFS tương ứng với đường đi ngắn nhất từ s đến v trên đồ thị G .
- Thuật toán làm việc trên đồ thị vô hướng và có hướng.



BFS(s) tree

BFS tạo nên **cây BFS** chứa đỉnh s là gốc và các đỉnh đạt được từ s



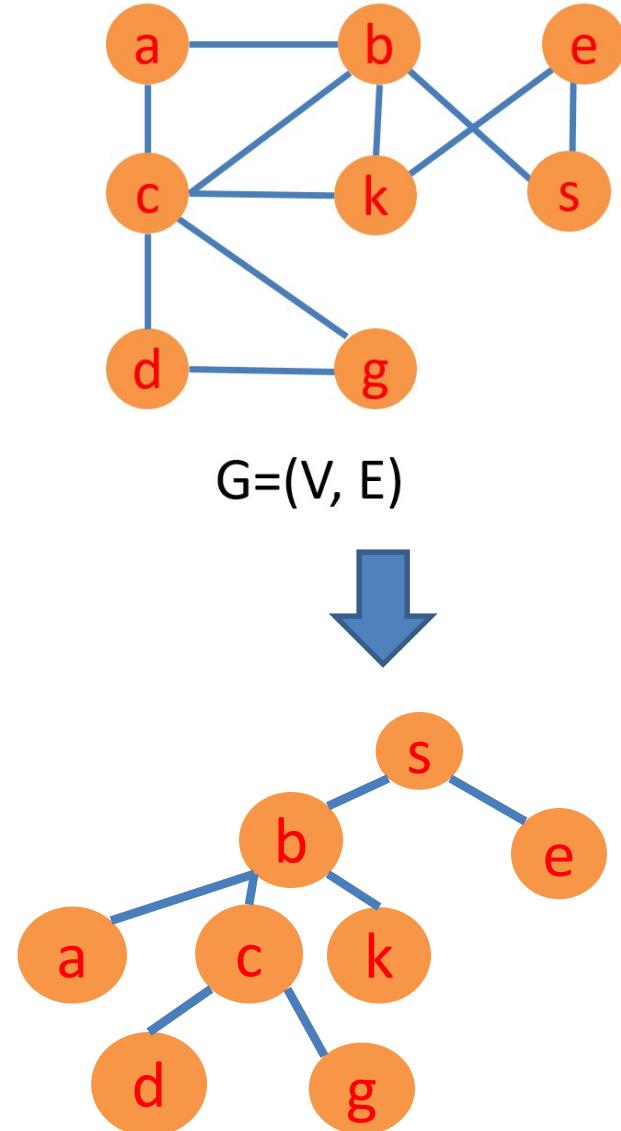
$$G=(V, E)$$

Danh sách kè của s

- Từ s : đến được b và e . Thăm chúng và cho chúng vào queue: $Q = \{b, e\}$
- Dequeue (Q): lấy b ra khỏi Q , lúc này $Q = \{e\}$
 - Từ b : đến được a , c , k , s . Nhưng s đã thăm, nên ta chỉ thăm a , c , k ; và cho chúng vào queue: $Q = \{e, a, c, k\}$
- Dequeue(Q): lấy e khỏi Q , lúc này $Q = \{a, c, k\}$
 - Từ e : đến được k , s . Nhưng tất cả các đỉnh này đã được thăm.
- Dequeue(Q): lấy a khỏi Q , lúc này $Q = \{c, k\}$
 - Từ a : đến được b , c . Nhưng tất cả các đỉnh này đã được thăm.
- Dequeue(Q): lấy c khỏi Q , lúc này $Q = \{k\}$.
 - Từ c : đến được a , b , d , g , k . Nhưng a , b , k đã thăm, nên ta chỉ thăm d , g ; và cho chúng vào queue: $Q = \{k, d, g\}$
- Dequeue(Q): lấy k khỏi Q , lúc này $Q = \{d, g\}$.
 - Từ k : đến được b , c , e . Nhưng tất cả các đỉnh này đã được thăm.
- Dequeue (Q): lấy d khỏi Q , lúc này $Q = \{g\}$
 - Từ d : đến được c , g . Nhưng tất cả các đỉnh này đã được thăm.
- Dequeue(Q): lấy g khỏi Q , lúc này $Q = \text{empty}$
 - Từ g : đến được d , c . Nhưng tất cả các đỉnh này đã được thăm.
- Q lúc này rỗng. Tất cả các đỉnh của đồ thị đã được thăm. Thuật toán kết thúc.

Tìm kiếm theo chiều rộng (Breadth-first Search)

```
void BFS(s) {  
    // Tìm kiếm theo chiều rộng bắt đầu từ đỉnh s  
    visited[s] ← 1; //đã thăm  
    Q ← ∅; enqueue(Q,s); // Nạp s vào Q  
    while (Q ≠ ∅)  
    {  
        u ← dequeue(Q); // Lấy u khỏi Q  
        for v ∈ Adj[u]  
            if (visited[v] == 0) //chưa thăm  
            {  
                visited[v] ← 1; //đã thăm  
                enqueue(Q,v) // Nạp v vào Q  
            }  
    }  
}  
  
void main ()  
{  
    for s ∈ V // Khởi tạo  
        visited[s] ← 0;  
  
    for s ∈ V  
        if (visited[s]==0) BFS(s);  
}
```



Tìm kiếm theo chiều rộng (Breadth-first Search)

```
#include <bits/stdc++.h>
using namespace std;

bool *visited;
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;     // Pointer to an array containing adjacency lists (double linked list)

public:
    int numVertex();
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void BFS(int s);          // prints BFS traversal from a given source s
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

int Graph::numVertex()
{
    return V;
}

void Graph::addEdge(int u, int v) //do thi co huong
{
    adj[u].push_back(v); // Add v to u's list.
    // adj[v].push_back(u);
}
```

Tìm kiếm theo chiều rộng (Breadth-first Search)

```
void Graph::BFS(int s)
{
    queue<int> q;
    // Mark the current node as visited and enqueue it
    visited[s] = true;
    q.push(s);
    // 'i' will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;
    while(!q.empty())
    {
        // Dequeue a vertex from queue and print it
        s = q.front();
        cout << s << " ";
        q.pop();

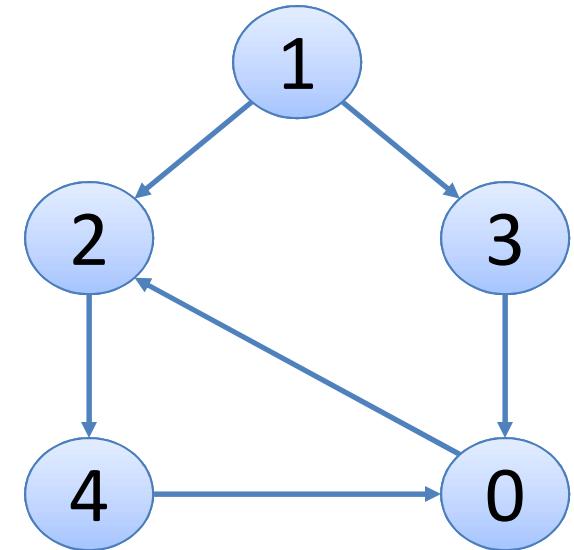
        //Duyệt qua danh sách kề của đỉnh s:
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            int v = *i;
            if (!visited[v])
            {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

Tìm kiếm theo chiều rộng (Breadth-first Search)

```
int main()
{
    // Tao do thi co huong gom 5 dinh va 6 canh
    Graph g(5);
    g.addEdge(1, 2); g.addEdge(1, 3);
    g.addEdge(2, 4); g.addEdge(4, 0);
    g.addEdge(3, 0); g.addEdge(0, 2);

    // Khoi tao:
    int numV = g.numVertex();
    visited = new bool[numV];
    for(int i = 0; i < numV; i++) visited[i] = false;

    cout << "BFS(1): ";
    g.BFS(1);
    return 0;
}
```



```
BFS(1): 1 2 3 4 0
```

Phân tích BFS

```
void BFS(s) {  
    // Tìm kiếm theo chiều rộng bắt đầu từ đỉnh s  
    visited[s] ← 1; //đã thăm  
    Q ← ∅; enqueue(Q,s); // Nạp s vào Q  
    while (Q ≠ ∅)  
    {  
        u ← dequeue(Q); // Lấy u khỏi Q  
        for v ∈ Adj[u]  
            if (visited[v] == 0) //chưa thăm  
            {  
                visited[v] ← 1; //đã thăm  
                enqueue(Q,v) // Nạp v vào Q  
            }  
    }  
}  
void main ()  
{  
    for s ∈ V // Khởi tạo  
        visited[s] ← 0;  
  
    for s ∈ V  
        if (visited[s]==0) BFS(s);  
}
```

- Việc khởi tạo đòi hỏi $O(|V|)$.
- Vòng lặp duyệt
 - Mỗi đỉnh được nạp vào và loại ra khỏi hàng đợi một lần, mỗi thao tác đòi hỏi thời gian $O(1)$. Như vậy tổng thời gian làm việc với hàng đợi là $O(|V|)$.
 - Danh sách kè của mỗi đỉnh được duyệt qua đúng một lần. Tổng độ dài của tất cả các danh sách kè là $O(|E|)$.
- Tổng cộng ta có thời gian tính của $BFS(s)$ là $O(|V|+|E|)$, là tuyến tính theo kích thước của danh sách kè biểu diễn đồ thị.

Tìm kiếm theo chiều rộng (Breadth-first Search)

- **Input:** Đồ thị $G = (V, E)$, vô hướng hoặc có hướng.
đỉnh $s \in V$: đỉnh xuất phát
- **Output:**
 - $d[v] =$ khoảng cách (độ dài của đường đi ngắn nhất) từ s đến v , với mọi $v \in V$. $d[v] = \infty$ nếu v không đạt tới được từ s .
 - $truoc[v] = u$ đỉnh đi trước v trong đường đi từ s đến v có độ dài $d[v]$.
 - Xây dựng cây BFS với gốc tại s chứa tất cả các đỉnh đạt tới được từ s .

Tìm kiếm theo chiều rộng (Breadth-first Search)

```
void BFS(s) {  
    // Tìm kiếm theo chiều rộng bắt đầu từ đỉnh s  
    visited[s] ← 1; //đã thăm  
    d[s] ← 0; truoc[s] ← null;  
    Q ← ∅; enqueue(Q,s); // Nạp s vào Q  
    while (Q ≠ ∅)  
    {  
        u ← dequeue(Q); // Lấy u khỏi Q  
        for v ∈ Adj[u]  
            if (visited[v] == 0) //chưa thăm  
            {  
                visited[v] ← 1; //đã thăm  
                d[v] ← d[u] + 1; truoc[v] ← u;  
                enqueue(Q,v) // Nạp v vào Q  
            }  
    }  
}  
void main ()  
{  
    for s ∈ V // Khởi tạo  
    {  
        visited[s] ← 0; d[s] ← ∞; truoc[s] ← null;  
    }  
    for s ∈ V  
        if (visited[s]==0) BFS(s);  
}
```

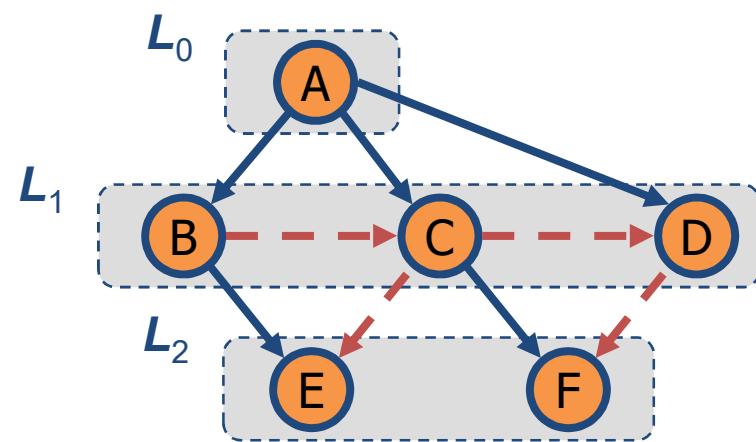
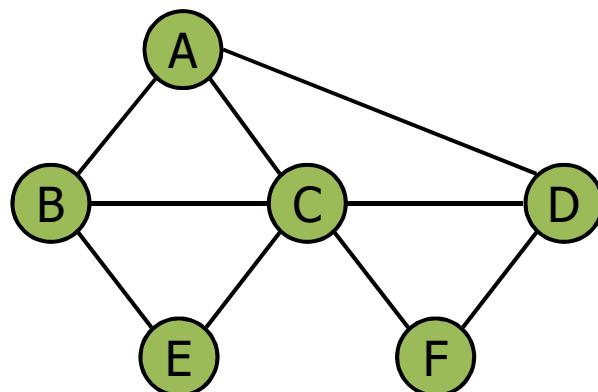
Q : hàng đợi các đỉnh được thăm
 $visited[v]$: đánh dấu thăm đỉnh v
 $d[v]$: khoảng cách từ s đến v
 $truoc[u]$: đỉnh đi trước v

Cây BFS(s)

- Đối với đồ thị $G = (V, E)$ và đỉnh s . Sau khi thực hiện $\text{BFS}(s)$, thu được đồ thị con $G_\pi = (V_\pi, E_\pi)$ trong đó
 - $V_\pi = \{v \in V : \text{truoc}[v] \neq \text{NULL}\} \cup \{s\}$
 - $E_\pi = \{(\text{truoc}[v], v) \in E : v \in V_\pi \setminus \{s\}\}$
- $G_\pi = (V_\pi, E_\pi)$ là cây và được gọi là cây $\text{BFS}(s)$
- Các cạnh trong E_π được gọi là cạnh của cây. $|E_\pi| = |V_\pi| - 1$.
- $\text{BFS}(s)$ cho phép đến thăm tất cả các đỉnh đạt tới được từ s .
- Trình tự thăm các đỉnh khi thực hiện $\text{BFS}(s)$: Đầu tiên đến thăm các đỉnh đạt được từ s bởi đường đi qua 1 cạnh, sau đó là thăm các đỉnh đạt được từ s bởi đường đi qua 2 cạnh, ... Do đó nếu đỉnh t được thăm trong $\text{BFS}(s)$ thì nó sẽ được thăm theo đường đi ngắn nhất theo số cạnh.

BFS – Loang trên đồ thị

- Thứ tự thăm đỉnh nhờ thực hiện BFS(A)



2. Duyệt đồ thị

2.1. Tìm kiếm theo chiều rộng

2.2. Tìm kiếm theo chiều sâu

2.3. Một số ứng dụng

Tìm kiếm theo chiều sâu (Depth-First Search)

(* Main Program*)

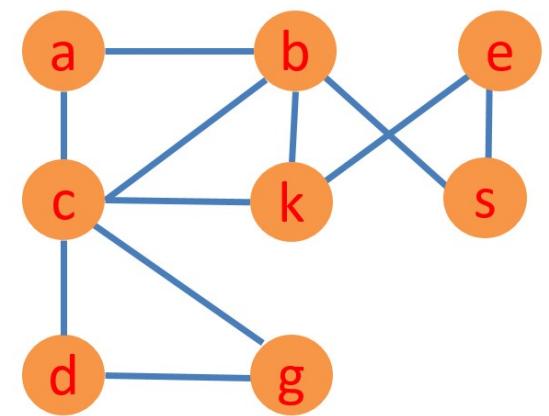
1. **for** $s \in V$
2. $\text{visited}[s] \leftarrow \text{false}$
3. **for** $s \in V$
4. **if** ($\text{visited}[s] == \text{false}$)
5. $\text{DFS}(s)$

DFS(s)

1. $\text{visited}[s] \leftarrow \text{true}$ // Thăm đỉnh s
2. **for each** $v \in \text{Adj}[s]$
3. **if** ($\text{visited}[v] == \text{false}$)
4. $\text{DFS}(v)$

```
void Graph::DFS(int s)
{
    visited[s] = true;
    cout << s << " ";

    // 'i' will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;
    //Duyệt qua danh sách kề của đỉnh u:
    for (i = adj[s].begin(); i != adj[s].end(); ++i)
    {
        int v = *i;
        if (!visited[v])
        {
            visited[v] = true;
            DFS(v);
        }
    }
}
```



$G=(V, E)$

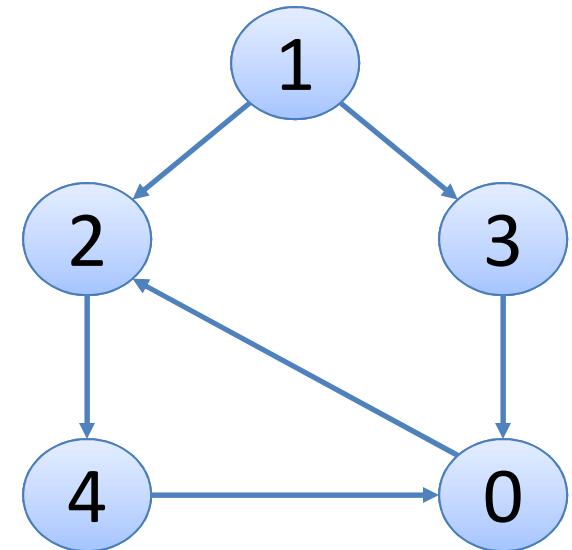
Tìm kiếm theo chiều sâu (Depth-First Search)

```
int main()
{
    // Tao do thi co huong gom 5 dinh va 6 canh
    Graph g(5);
    g.addEdge(1, 2); g.addEdge(1, 3);
    g.addEdge(2, 4); g.addEdge(4, 0);
    g.addEdge(3, 0); g.addEdge(0, 2);

    // Khoi tao:
    int numV = g.numVertex();
    visited = new bool[numV];
    for(int i = 0; i < numV; i++) visited[i] = false;

    cout << "BFS(1): ";
    g.BFS(1);
    cout<<endl;

    for(int i = 0; i < numV; i++) visited[i] = false;
    cout << "DFS(1): ";
    g.DFS(1);
    return 0;
}
```



| |
|-------------------|
| BFS(1): 1 2 3 4 0 |
| DFS(1): 1 2 4 0 3 |

Phân tích DFS

(* Main Program*)

```
1. for  $s \in V$ 
2.    $\text{visited}[s] \leftarrow \text{false}$ 
3. for  $s \in V$ 
4.   if ( $\text{visited}[s] == \text{false}$ )
5.     DFS( $s$ )
```

DFS(s)

```
1.    $\text{visited}[s] \leftarrow \text{true}$  // Thăm đỉnh  $s$ 
2.   for each  $v \in \text{Adj}[s]$ 
3.     if ( $\text{visited}[v] == \text{false}$ )
4.       DFS( $v$ )
```

- Ở main: vòng lặp trên các dòng 1-2 và 3-5 đòi hỏi thời gian $O(|V|)$, chưa tính thời gian thực hiện lệnh DFS (s).
- Vòng lặp trong DFS (s) thực hiện việc duyệt cạnh của đồ thị:
 - Các dòng 3-4 của DFS (s) sẽ thực hiện $|\text{Adj}[s]|$ lần
 - Mỗi cạnh được duyệt qua đúng một lần nếu đồ thị là có hướng và 2 lần nếu đồ thị là vô hướng

Vậy thời gian tổng cộng của DFS (s) trong chương trình chính là $\sum_{s \in V} |\text{Adj}[s]| = O(|E|)$

- Do đó, thời gian của DFS là $O(|V| + |E|)$.
- Như vậy, DFS có cùng độ phức tạp như BFS.

DFS: Nếu muốn đưa ra đường đi từ s đến các đỉnh còn lại của đồ thị

(* Main Program*)

1. **for** $s \in V$
2. $\text{visited}[s] \leftarrow \text{false}$
3. **for** $s \in V$
4. **if** ($\text{visited}[s] == \text{false}$)
5. $\text{DFS}(s)$

DFS(s)

1. $\text{visited}[s] \leftarrow \text{true}$ // Thăm đỉnh s
2. **for each** $v \in \text{Adj}[s]$
3. **if** ($\text{visited}[v] == \text{false}$)
4. $\text{DFS}(v)$



void main()

1. **for each** $s \in V$
2. $\text{truoc}[s] = \text{NULL};$
3. $\text{visited}[s] = \text{false};$
4. $\text{DFS}(s);$

DFS(s)

1. $\text{visited}[s] = \text{true};$ //Thăm đỉnh s
2. **for each** $v \in \text{Adj}[s]$
3. **if** ($\text{visited}[v] == \text{false}$) {
4. $\text{truoc}[v] \leftarrow s;$
5. $\text{DFS}(v);$
6. }

DFS: Phân loại cạnh

void main()

1. **for each** $s \in V$
2. $truoc[s] = \text{NULL};$
3. $\text{visited}[s] = \text{false};$
4. $time = 0$
5. **for each** $s \in V$
6. **if** ($\text{visited}[s] == \text{false}$) DFS(s);

DFS(s)

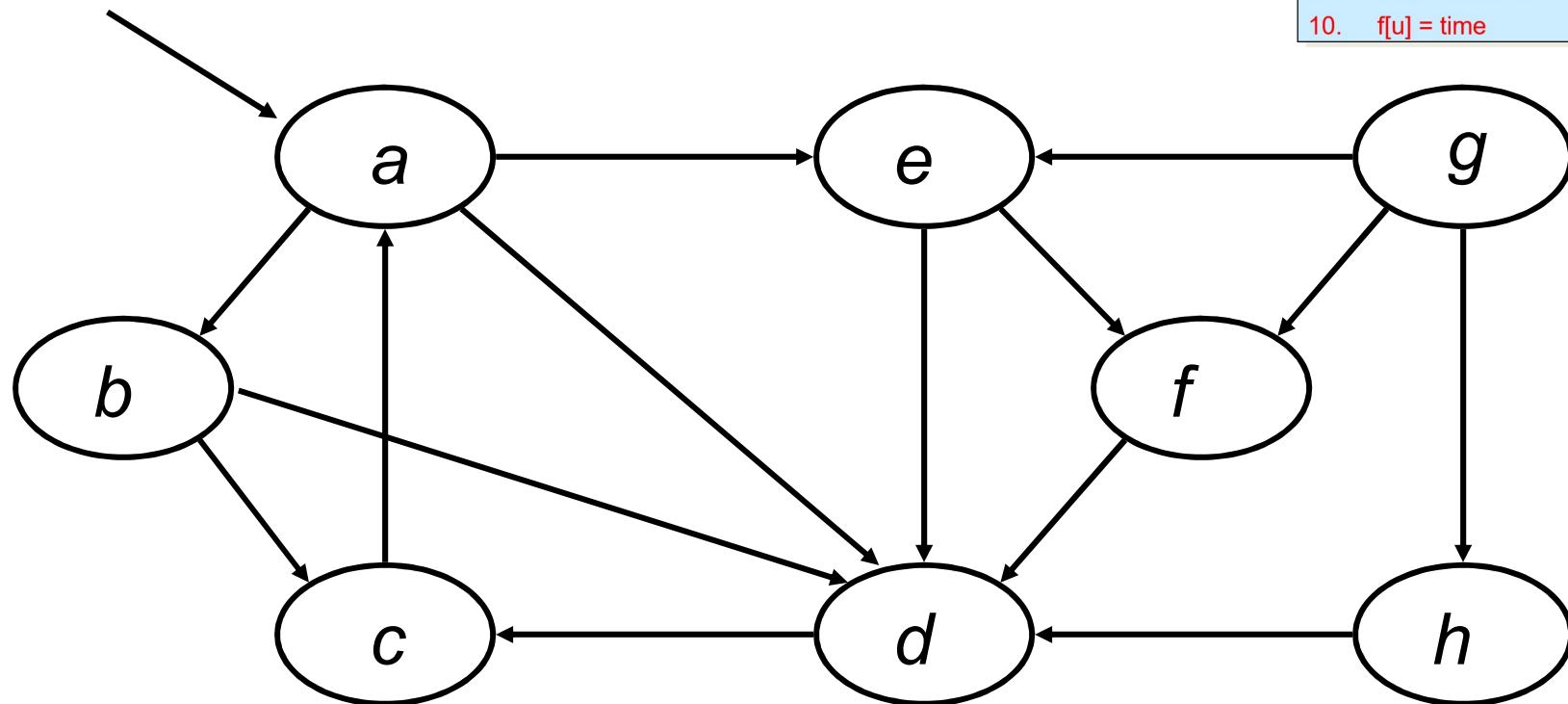
1. $\text{visited}[s] = \text{true}; //\text{Thăm đỉnh } s$
2. $time = time + 1$
3. $d[u] = time$
4. **for each** $v \in Adj[s]$
5. **if** ($\text{visited}[v] == \text{false}$) {
6. $truoc[v] \leftarrow s;$
7. DFS(v);
8. }
9. $time = time + 1$
10. $f[u] = time$

Với mỗi đỉnh u của đồ thị:

- $d[u]$: thời điểm bắt đầu thăm đỉnh u
- $f[u]$: thời điểm kết thúc thăm đỉnh u

Ví dụ: DFS

**Đỉnh xuất phát tìm kiếm
(Source vertex)**



void main()

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

DFS(s)

```
1.   visited[s] = true; //Thăm đỉnh s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

Để hoạt động của thuật toán là xác định, giả thiết rằng ta duyệt các đỉnh trong danh sách kè của một đỉnh theo thứ tự từ điển

Ví dụ: DFS

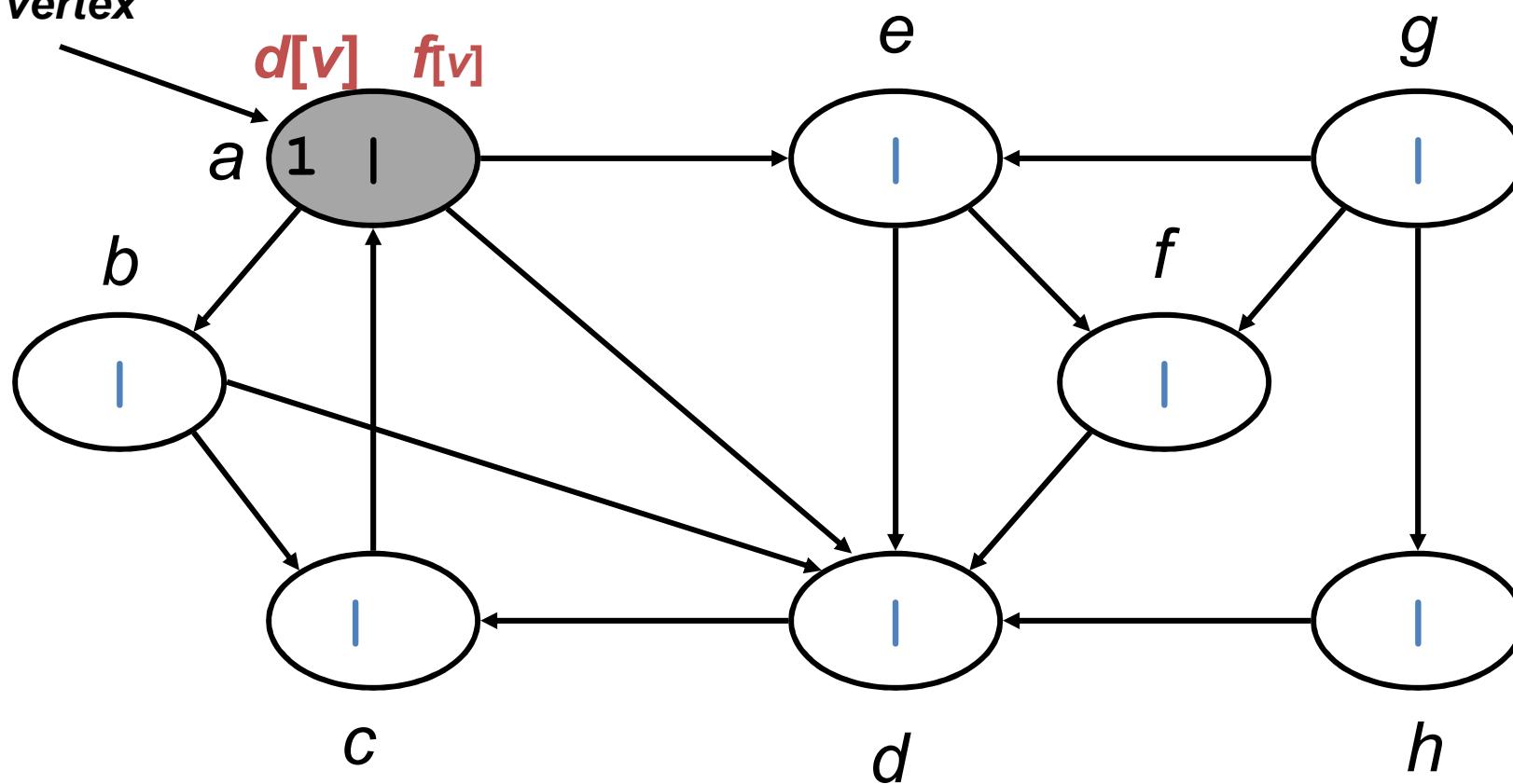
void main()

1. for each $s \in V$
2. $trouc[s] = \text{NULL};$
3. $visited[s] = \text{false};$
4. $time = 0$
5. for each $s \in V$
6. if ($visited[s] == \text{false}$) DFS(s);

DFS(s)

1. $visited[s] = \text{true}; //\text{Thăm đỉnh } s$
2. $time = time + 1$
3. $d[u] = time$
4. for each $v \in Adj[s]$
5. if ($visited[v] == \text{false}$) {
6. $trouc[v] \leftarrow s;$
7. DFS(v);
8. }
9. $time = time + 1$
10. $f[u] = time$

source
vertex



Ví dụ: DFS

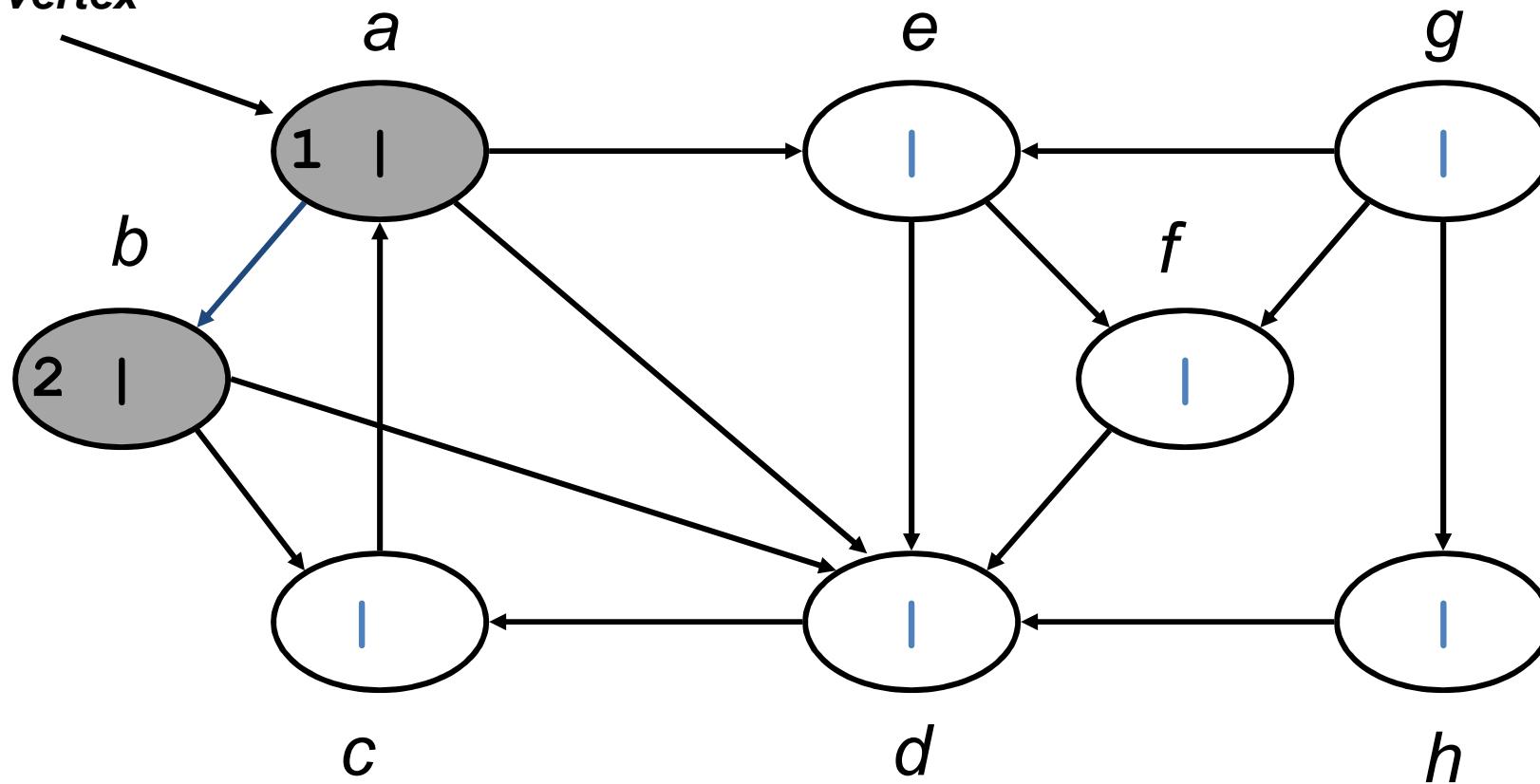
```
void main()
```

1. for each $s \in V$
2. $truc[s] = \text{NULL}$;
3. $visited[s] = \text{false}$;
4. $time = 0$
5. for each $s \in V$
6. if ($visited[s] == \text{false}$) DFS(s);

```
DFS( $s$ )
```

1. $visited[s] = \text{true}$; //Thăm đỉnh s
2. $time = time + 1$
3. $d[u] = time$
4. for each $v \in Adj[s]$
5. if ($visited[v] == \text{false}$) {
6. $truc[v] \leftarrow s$;
7. DFS(v);
8. }
9. $time = time + 1$
10. $f[u] = time$

source
vertex



Ví dụ: DFS

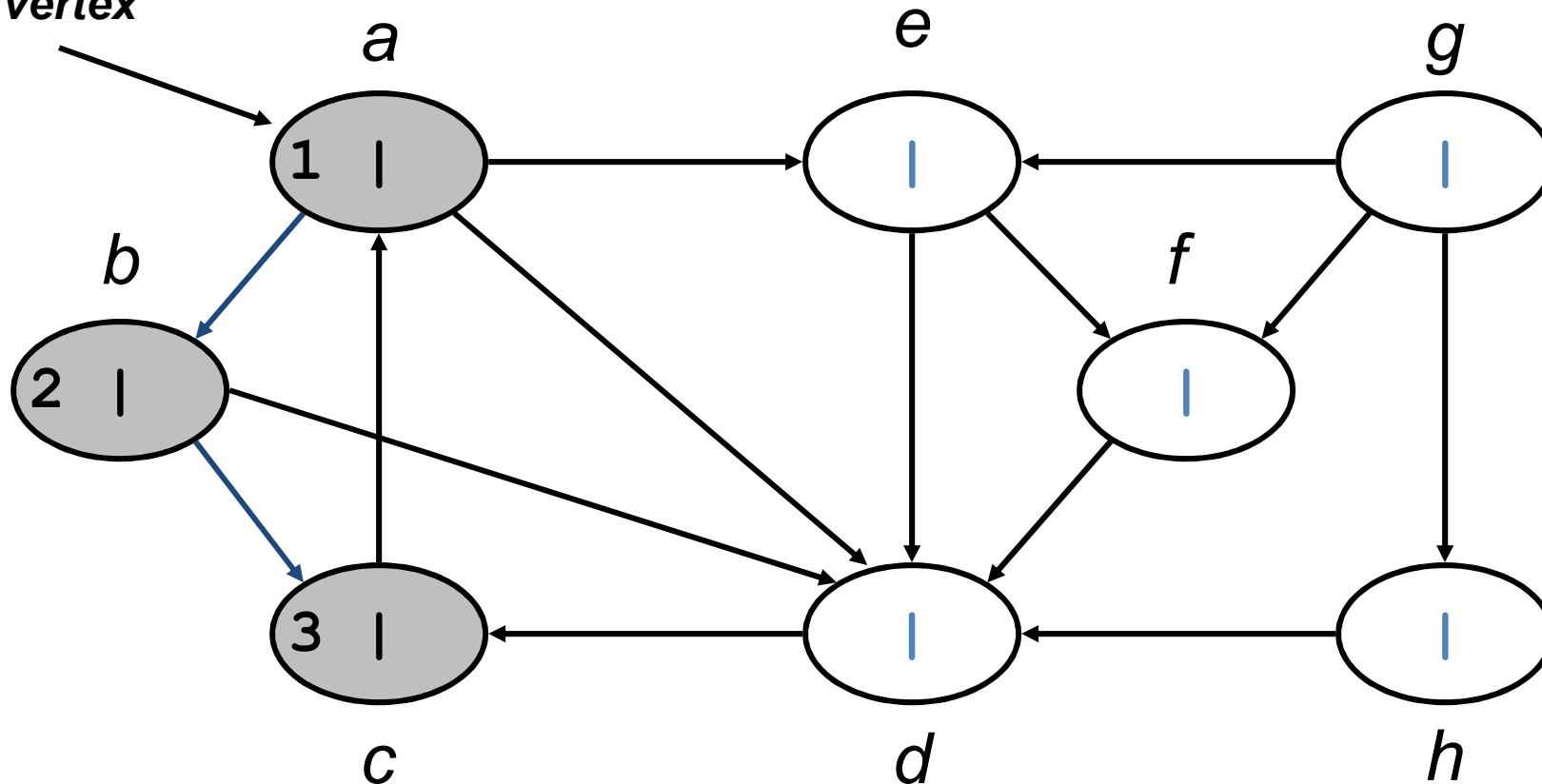
void main()

1. for each $s \in V$
2. $\text{truoc}[s] = \text{NULL};$
3. $\text{visited}[s] = \text{false};$
4. $\text{time} = 0$
5. for each $s \in V$
6. if ($\text{visited}[s] == \text{false}$) DFS(s);

DFS(s)

1. $\text{visited}[s] = \text{true}; //\text{Thăm đỉnh } s$
2. $\text{time} = \text{time} + 1$
3. $d[u] = \text{time}$
4. for each $v \in \text{Adj}[s]$
5. if ($\text{visited}[v] == \text{false}$) {
6. $\text{truoc}[v] \leftarrow s;$
7. DFS(v);
8. }
9. $\text{time} = \text{time} + 1$
10. $f[u] = \text{time}$

source
vertex



Ví dụ: DFS

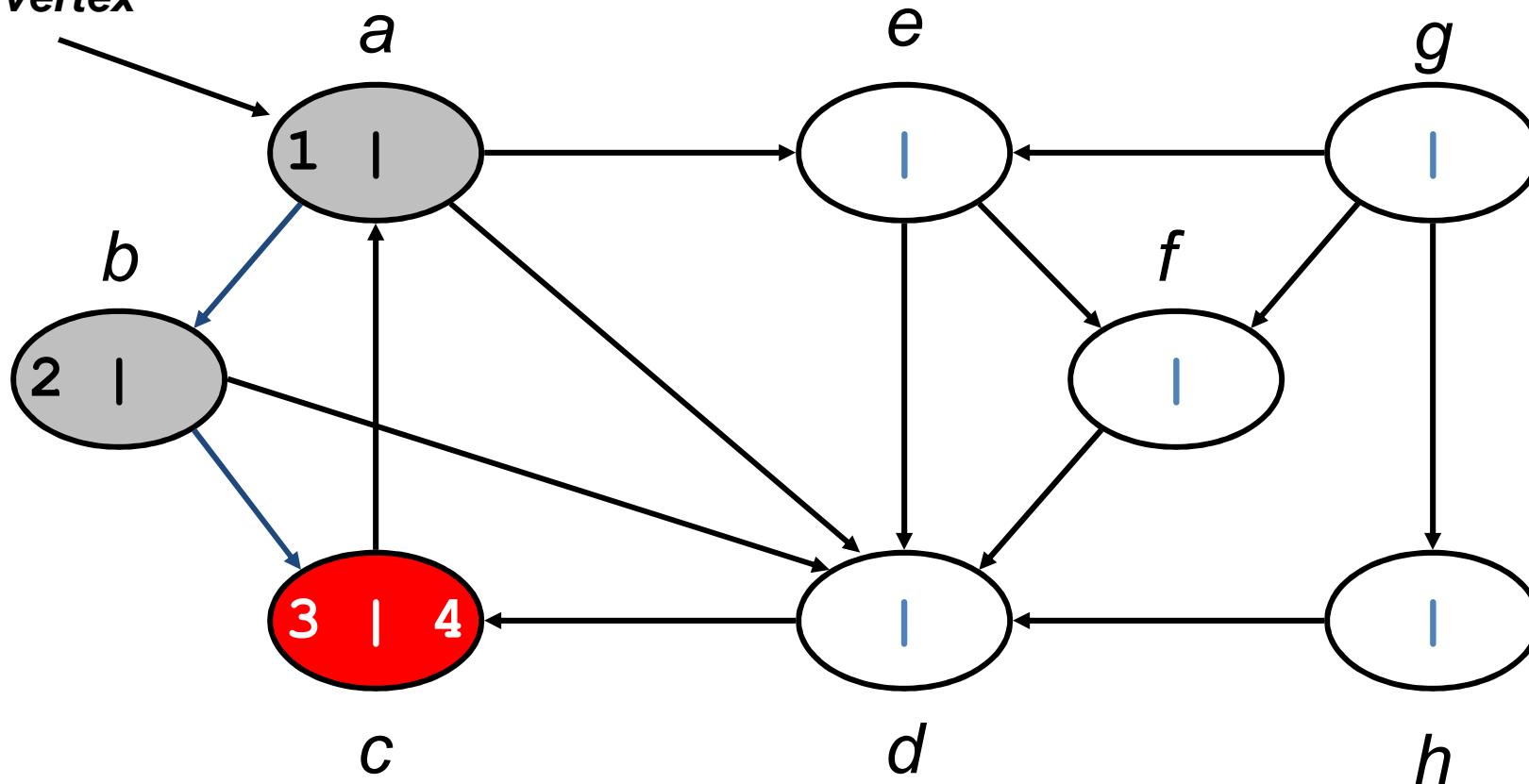
void main()

1. for each $s \in V$
2. $\text{truoc}[s] = \text{NULL};$
3. $\text{visited}[s] = \text{false};$
4. $\text{time} = 0$
5. for each $s \in V$
6. if ($\text{visited}[s] == \text{false}$) DFS(s);

DFS(s)

1. $\text{visited}[s] = \text{true};$ //Thăm đỉnh s
2. $\text{time} = \text{time} + 1$
3. $d[u] = \text{time}$
4. for each $v \in \text{Adj}[s]$
5. if ($\text{visited}[v] == \text{false}$) {
6. $\text{truoc}[v] \leftarrow s;$
7. DFS(v);
8. }
9. $\text{time} = \text{time} + 1$
10. $f[u] = \text{time}$

source
vertex



Ví dụ: DFS

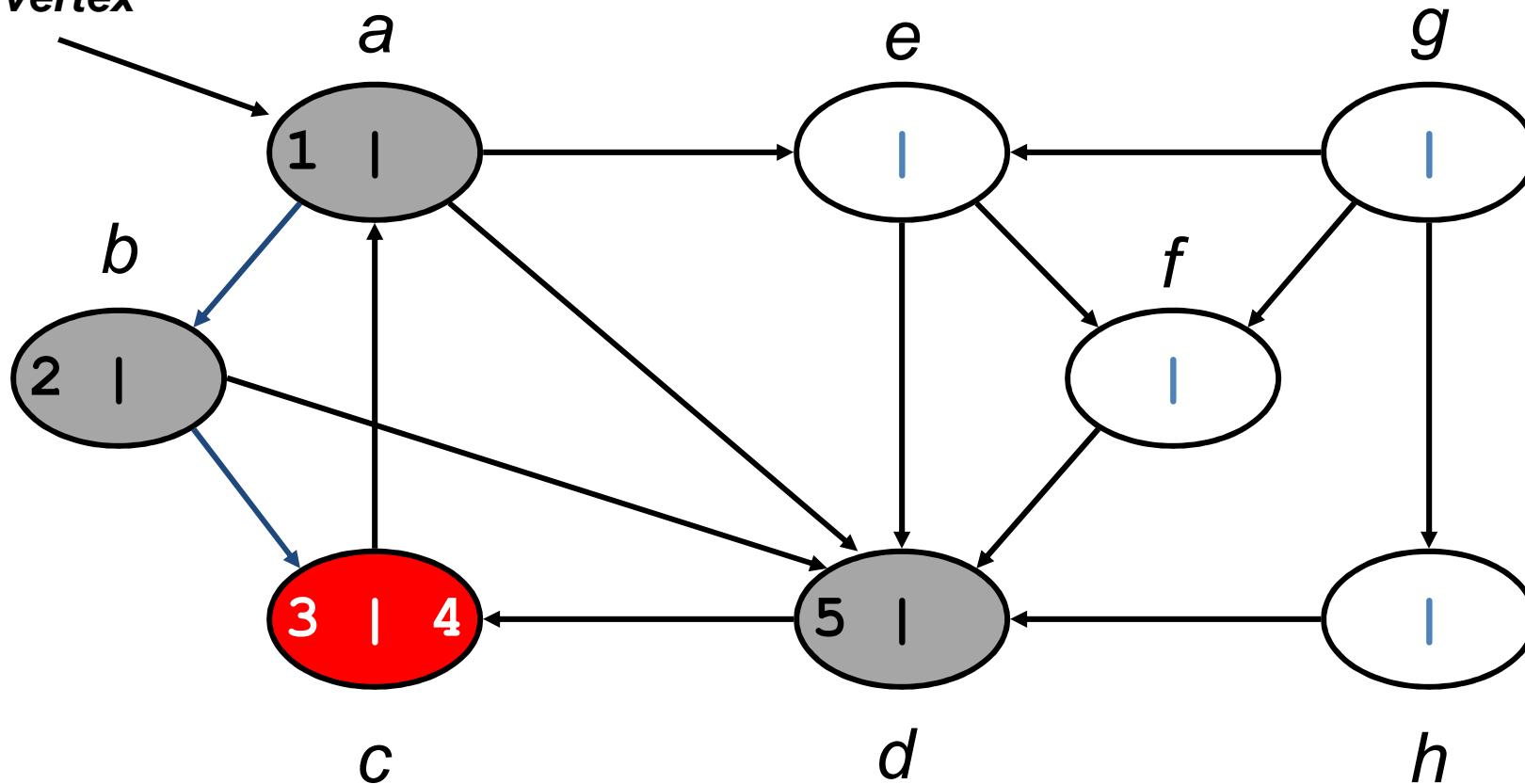
void main()

1. for each $s \in V$
2. $truoc[s] = \text{NULL}$;
3. $visited[s] = \text{false}$;
4. $time = 0$
5. for each $s \in V$
6. if ($visited[s] == \text{false}$) DFS(s);

DFS(s)

1. $visited[s] = \text{true}$; //Thăm đỉnh s
2. $time = time + 1$
3. $d[u] = time$
4. for each $v \in Adj[s]$
5. if ($visited[v] == \text{false}$) {
6. $truoc[v] \leftarrow s$;
7. DFS(v);
8. }
9. $time = time + 1$
10. $f[u] = time$

source
vertex



Ví dụ: DFS

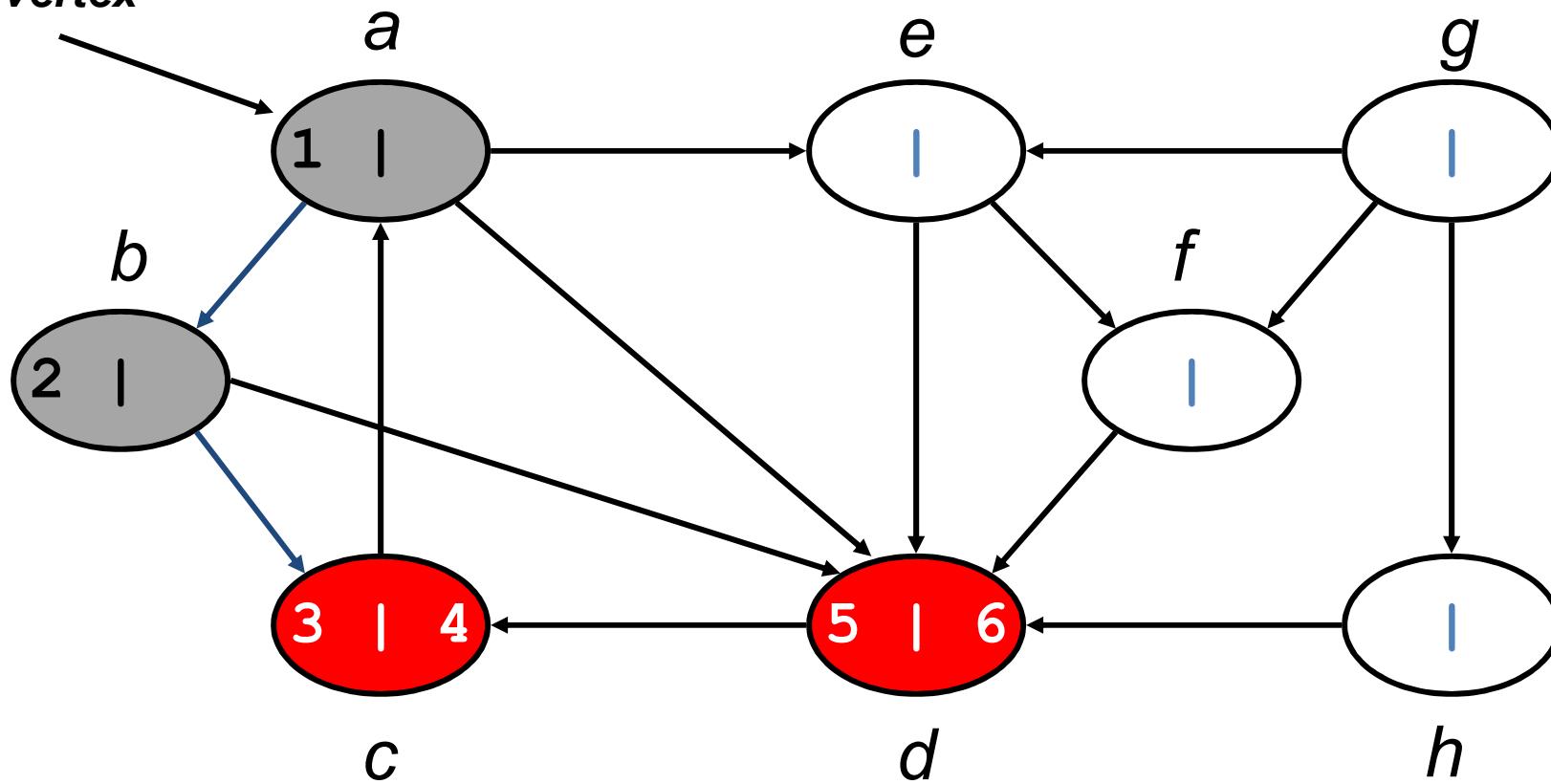
```
void main()
```

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

```
DFS( $s$ )
```

```
1.   visited[s] = true; //Thăm đỉnh s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

source
vertex



Ví dụ: DFS

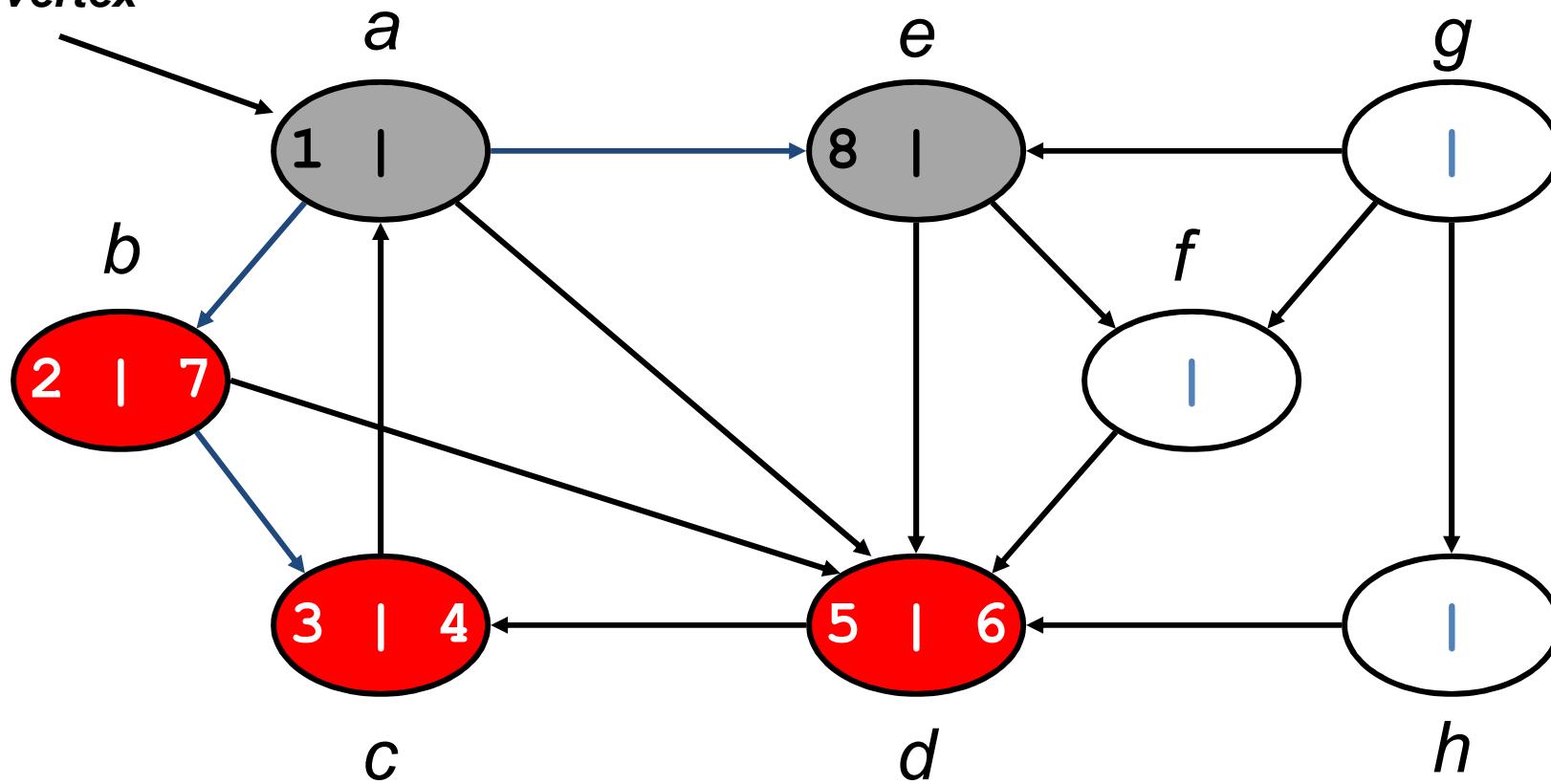
void main()

1. for each $s \in V$
2. $truc[s] = \text{NULL}$;
3. $visited[s] = \text{false}$;
4. $time = 0$
5. for each $s \in V$
6. if ($visited[s] == \text{false}$) DFS(s);

DFS(s)

1. $visited[s] = \text{true}$; //Thăm đỉnh s
2. $time = time + 1$
3. $d[u] = time$
4. for each $v \in Adj[s]$
5. if ($visited[v] == \text{false}$) {
6. $truc[v] \leftarrow s$;
7. DFS(v);
8. }
9. $time = time + 1$
10. $f[u] = time$

source
vertex



Ví dụ: DFS

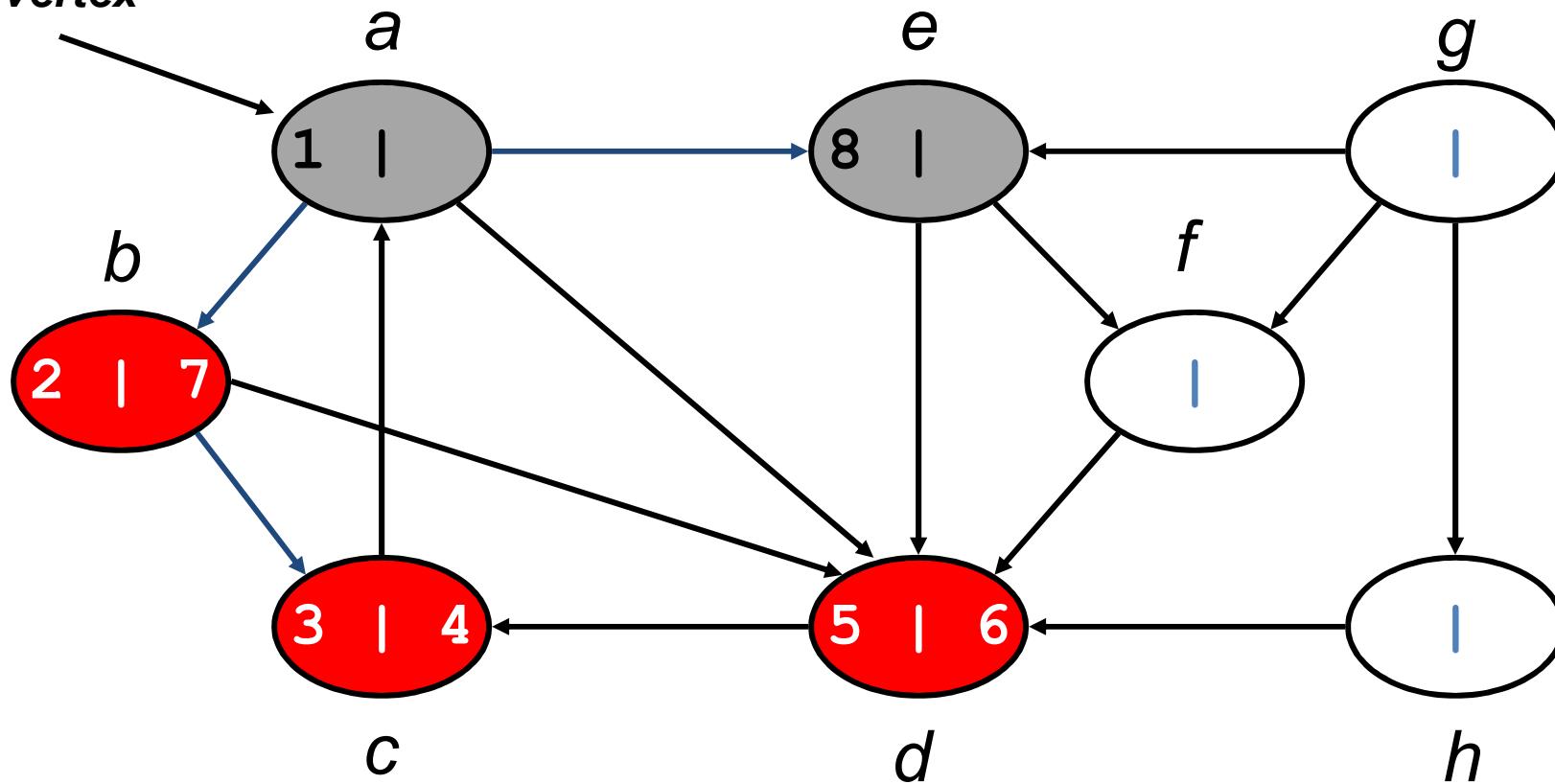
```
void main()
```

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

```
DFS( $s$ )
```

```
1.   visited[s] = true; //Thăm đính s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

source
vertex



Ví dụ: DFS

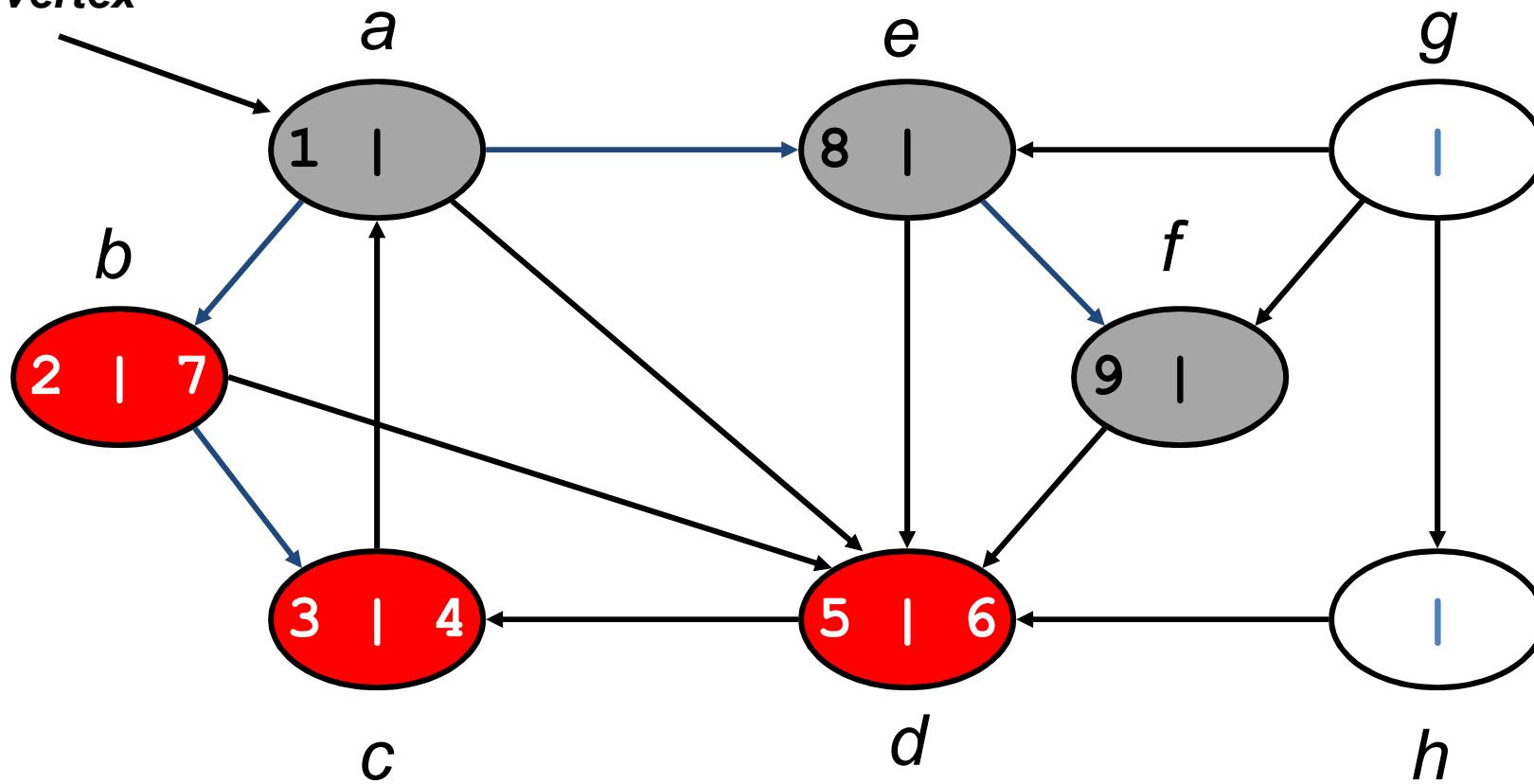
```
void main()
```

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

```
DFS(s)
```

```
1.   visited[s] = true; //Thăm đỉnh s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

source
vertex



Ví dụ: DFS

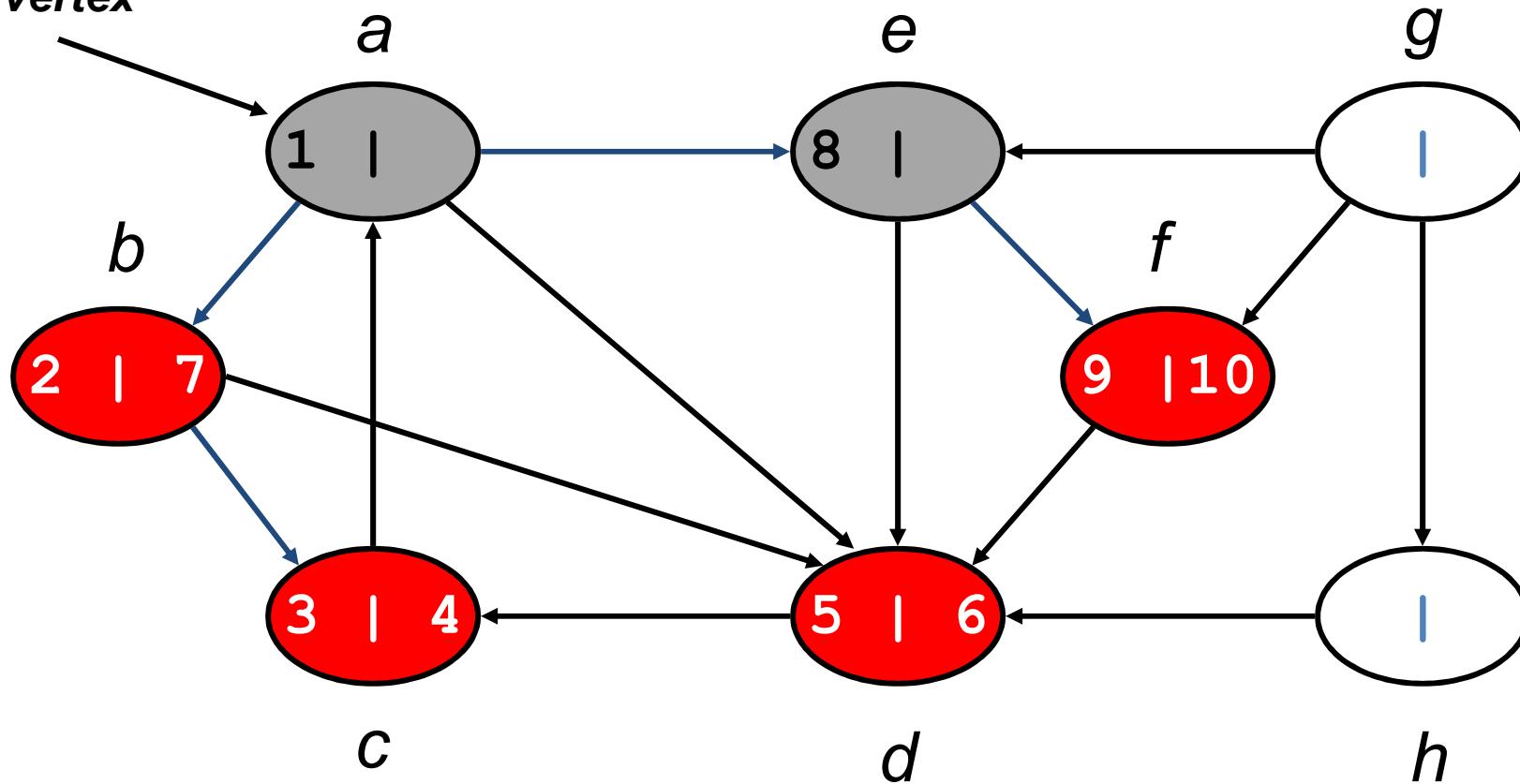
```
void main()
```

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

```
DFS( $s$ )
```

```
1.   visited[s] = true; //Thăm đính s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

source
vertex



Ví dụ: DFS

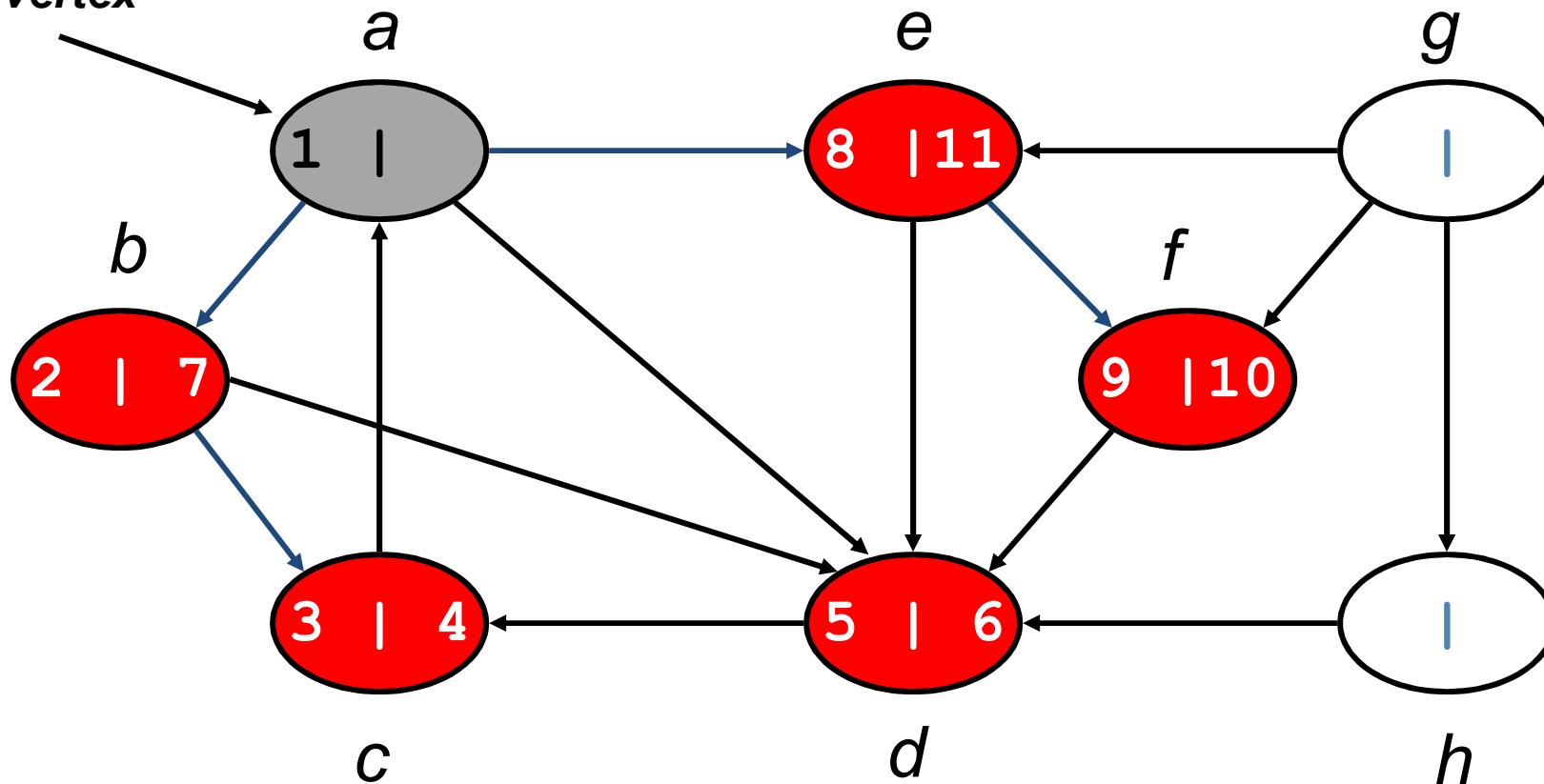
```
void main()
```

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

```
DFS(s)
```

```
1.   visited[s] = true; //Thăm đỉnh s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

source
vertex



Ví dụ: DFS

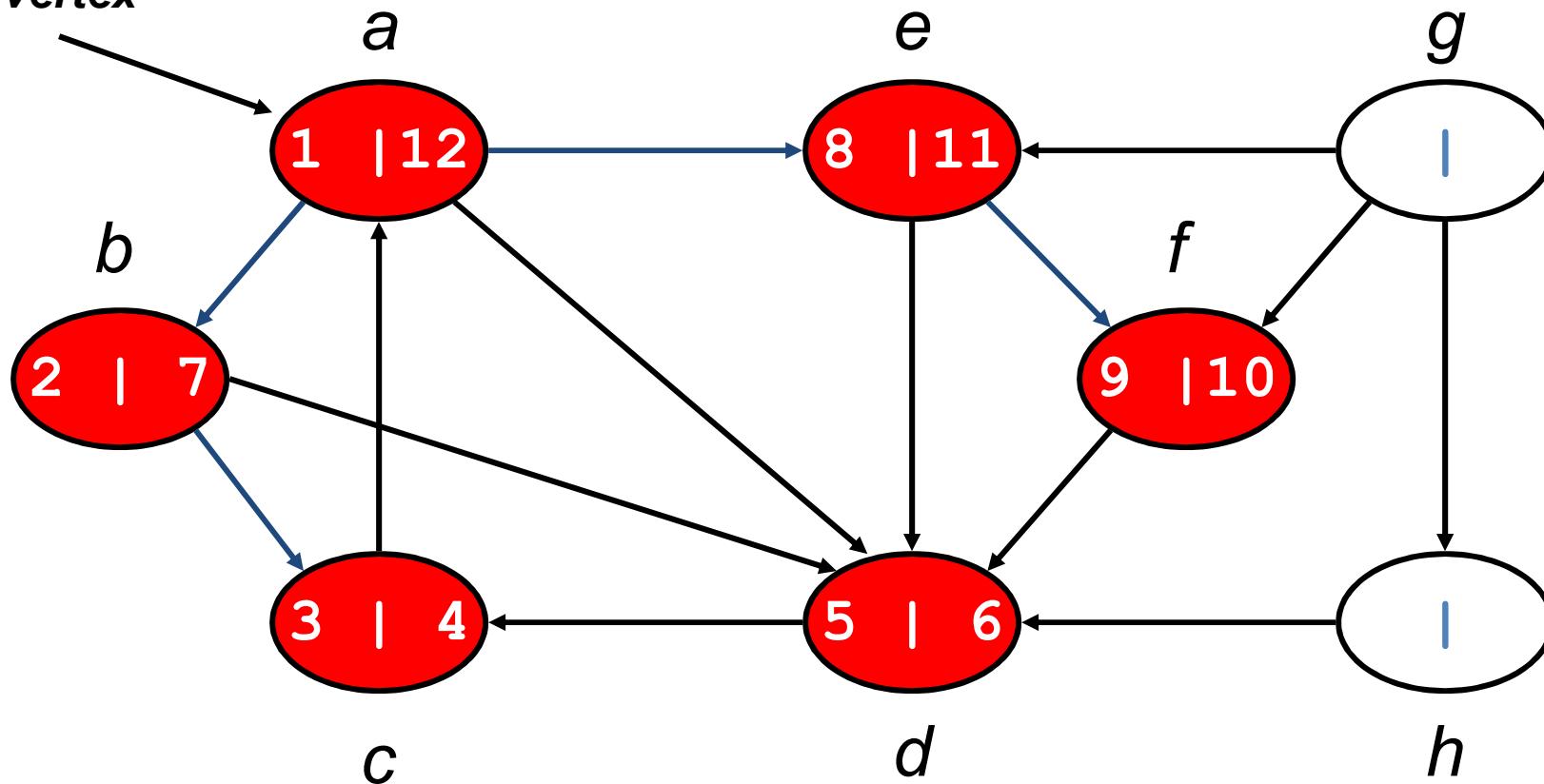
void main()

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

DFS(s)

```
1.   visited[s] = true; //Thăm đỉnh s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

source
vertex



Ví dụ: DFS

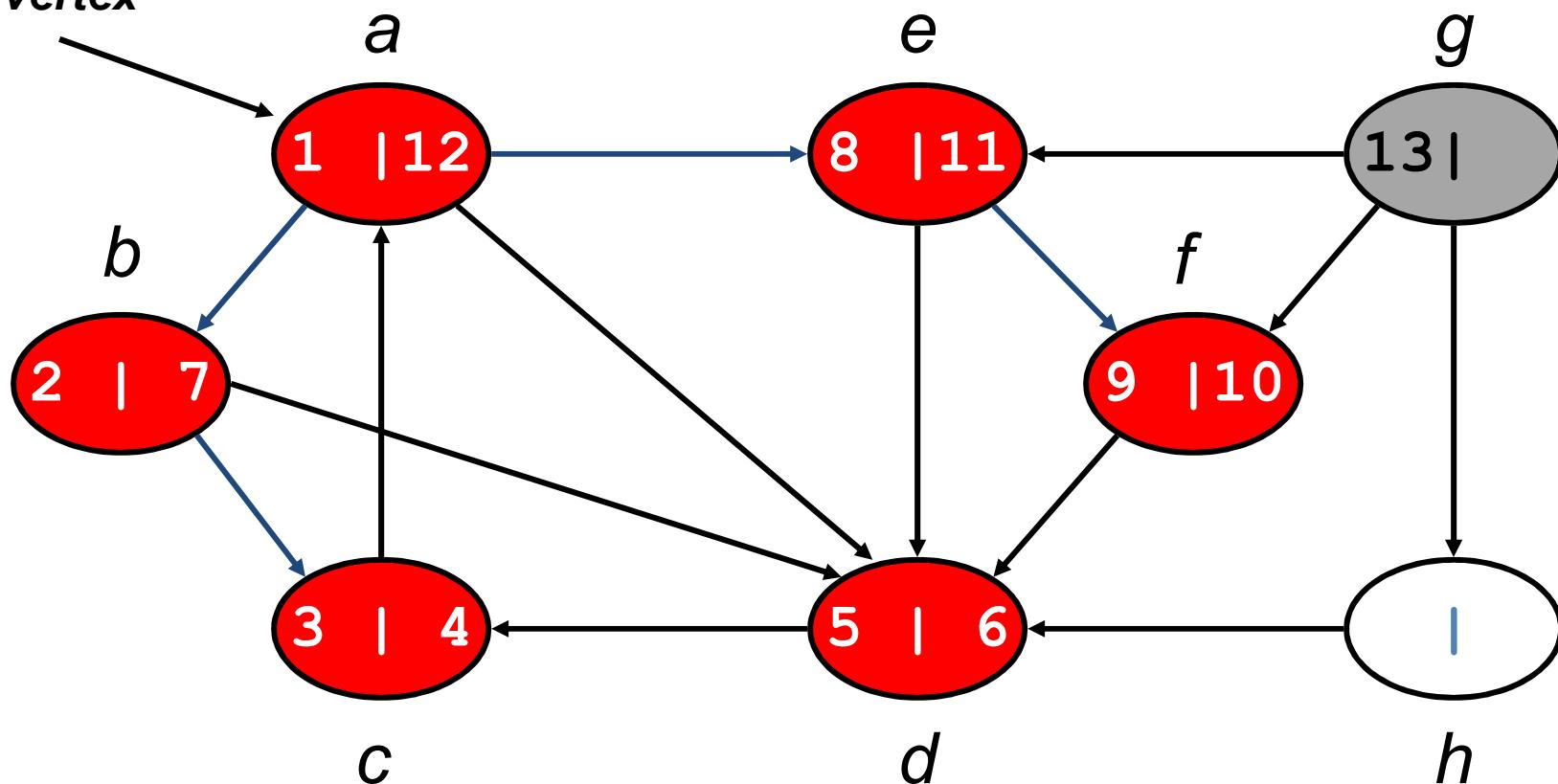
```
void main()
```

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

```
DFS( $s$ )
```

```
1.   visited[s] = true; //Thăm đỉnh s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

source
vertex



Ví dụ: DFS

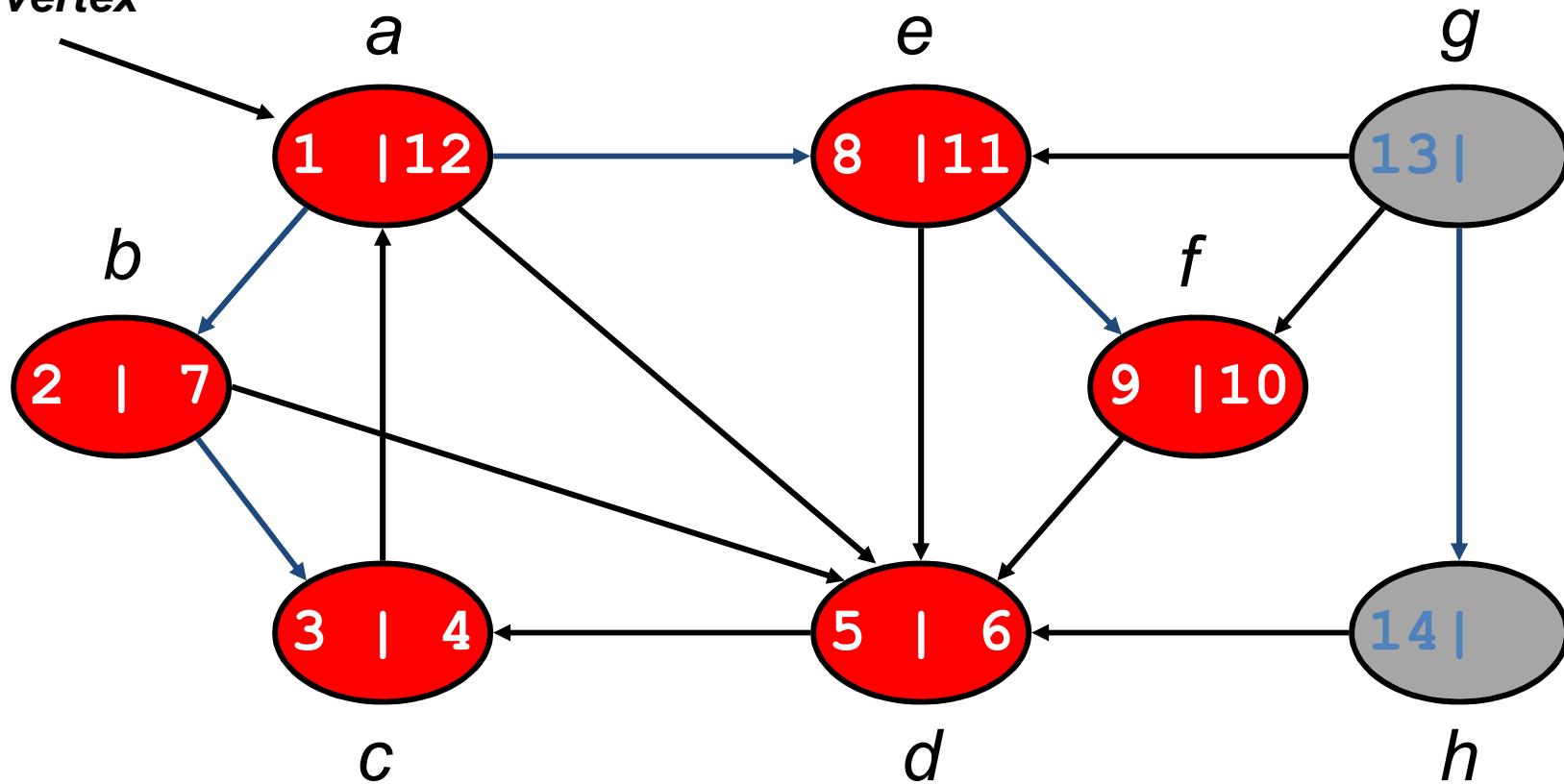
```
void main()
```

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

```
DFS( $s$ )
```

```
1.   visited[s] = true; //Thăm đỉnh s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

source
vertex



Ví dụ: DFS

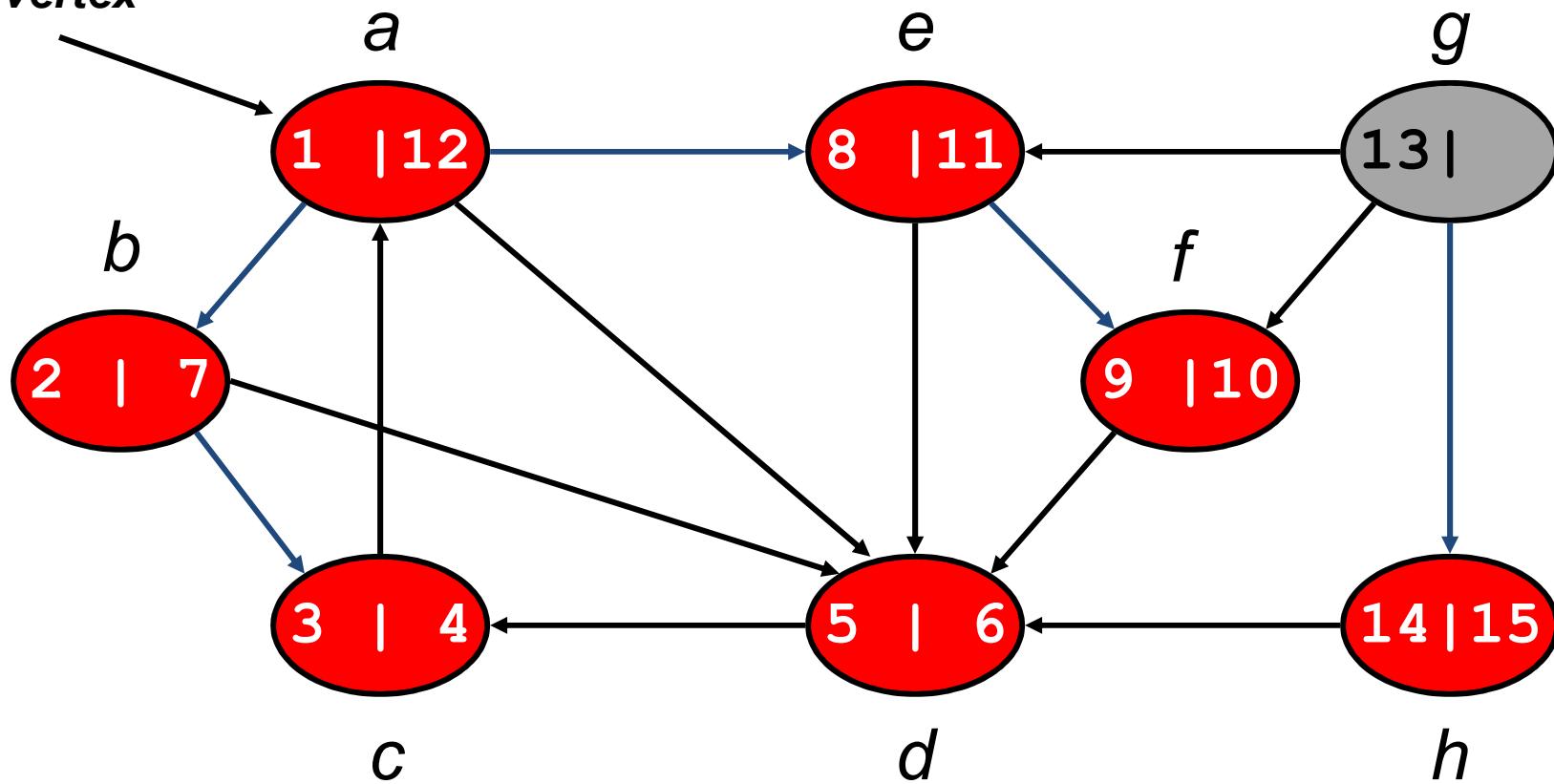
```
void main()
```

```
1.   for each  $s \in V$ 
2.       truoc[s] = NULL;
3.       visited[s] = false;
4.   time = 0
5.   for each  $s \in V$ 
6.       if (visited[s] == false) DFS(s);
```

```
DFS( $s$ )
```

```
1.   visited[s] = true; //Thăm đỉnh s
2.   time = time +1
3.   d[u] = time
4.   for each  $v \in Adj[s]$ 
5.       if (visited[v] == false) {
6.           truoc[v] ← s;
7.           DFS(v);
8.       }
9.   time = time +1
10.  f[u] = time
```

source
vertex



Ví dụ: DFS

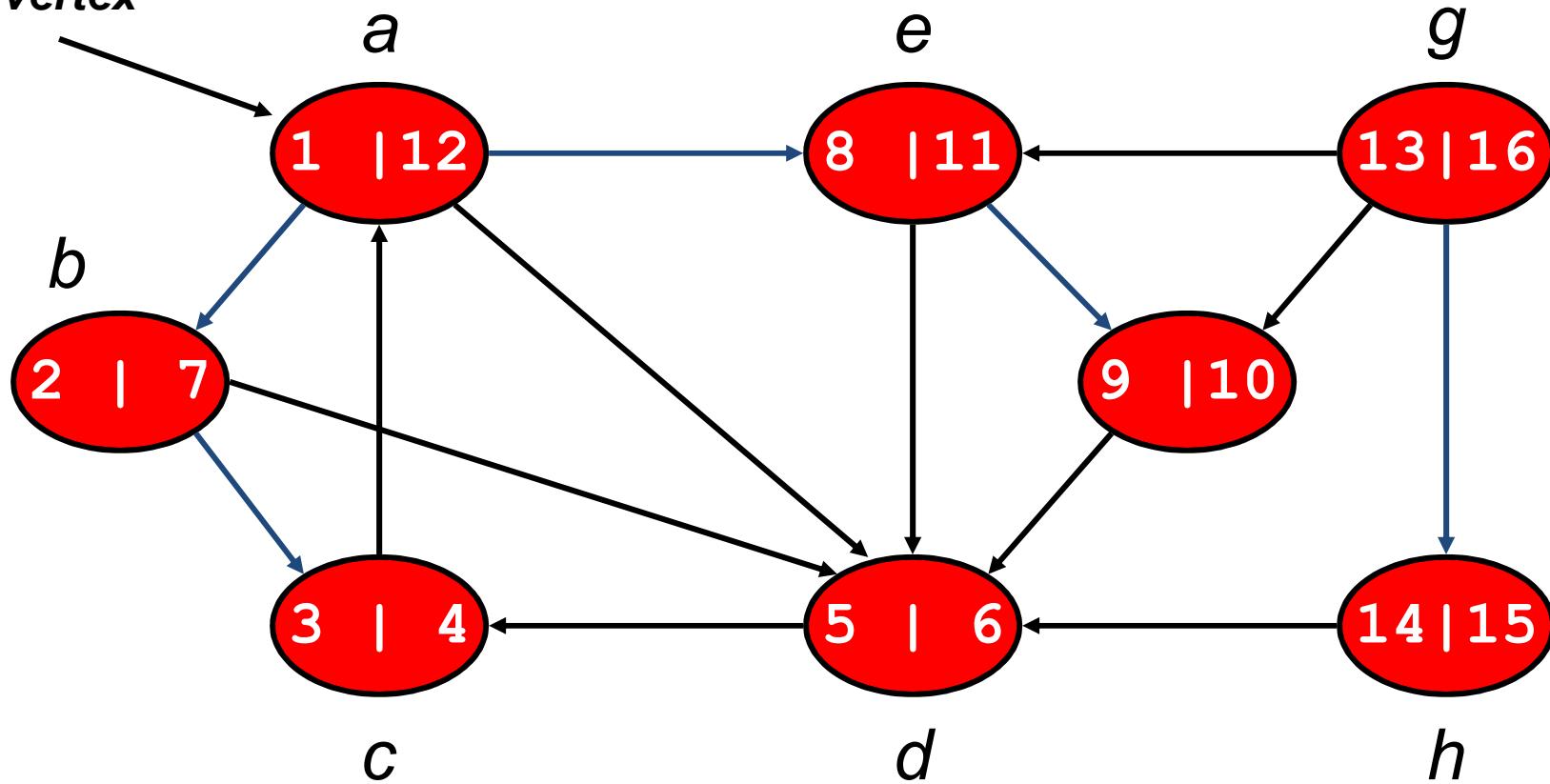
```
void main()
```

1. **for** each $s \in V$
2. **truoc**[s] = NULL;
3. **visited**[s] = false;
4. **time** = 0
5. **for** each $s \in V$
6. **if** (**visited**[s] == false) DFS(s);

```
DFS( $s$ )
```

1. **visited**[s] = true; //Thăm đỉnh s
2. **time** = **time** + 1
3. **d**[u] = **time**
4. **for** each $v \in Adj[s]$
5. **if** (**visited**[v] == false) {
6. **truoc**[v] $\leftarrow s$;
7. DFS(v);
8. }
9. **time** = **time** + 1
10. **f**[u] = **time**

source
vertex

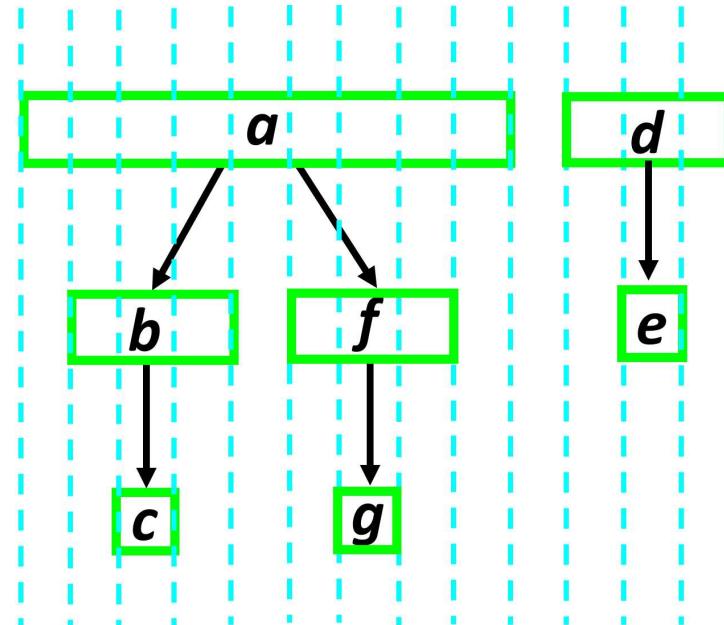
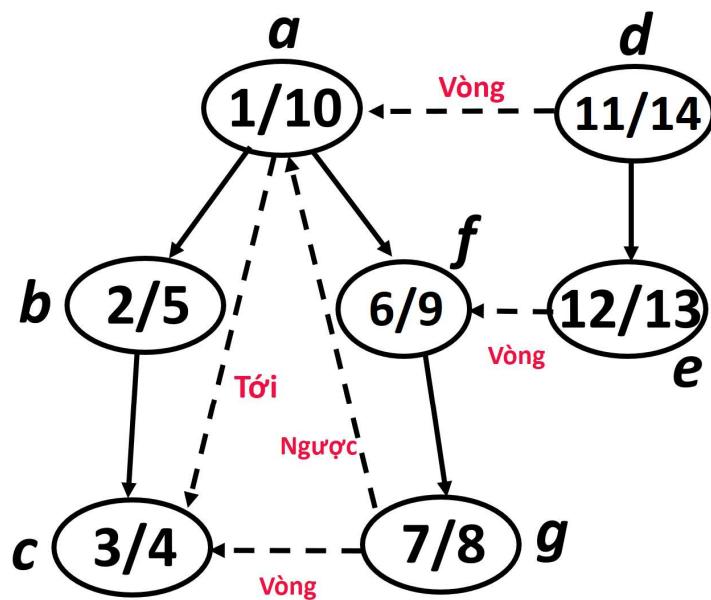


Bỏ đề về các khoảng lồng nhau

Cho đồ thị có hướng $G = (V, E)$, và cây DFS bất kỳ của G và hai đỉnh u, v tùy ý của nó. Khi đó

- u là con cháu của v khi và chỉ khi $[d[u], f[u]] \subseteq [d[v], f[v]]$
- u là tổ tiên của v khi và chỉ khi $[d[u], f[u]] \supseteq [d[v], f[v]]$
- u và v không có quan hệ họ hàng khi và chỉ khi $[d[u], f[u]]$ và $[d[v], f[v]]$ là không giao nhau.

- *Tree edge (cạnh cây)*: cạnh theo đó từ một đỉnh đến thăm đỉnh mới
- *Back edge (cạnh ngược)*: đi từ con cháu đến tổ tiên
- *Forward edge (cạnh tới)*: đi từ tổ tiên đến con cháu
- *Cross edge (cạnh vòng)*: giữa hai đỉnh không có họ hàng

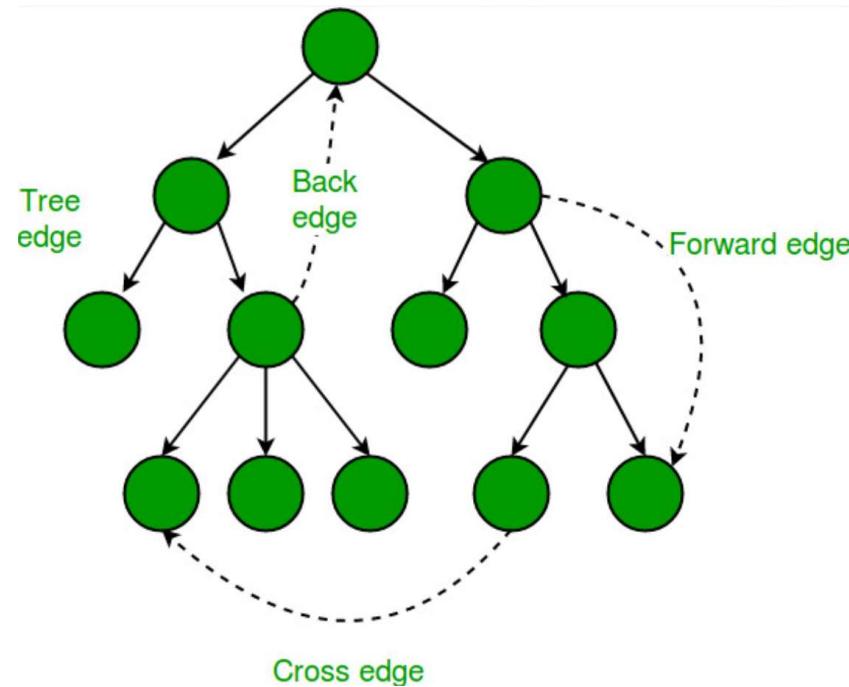
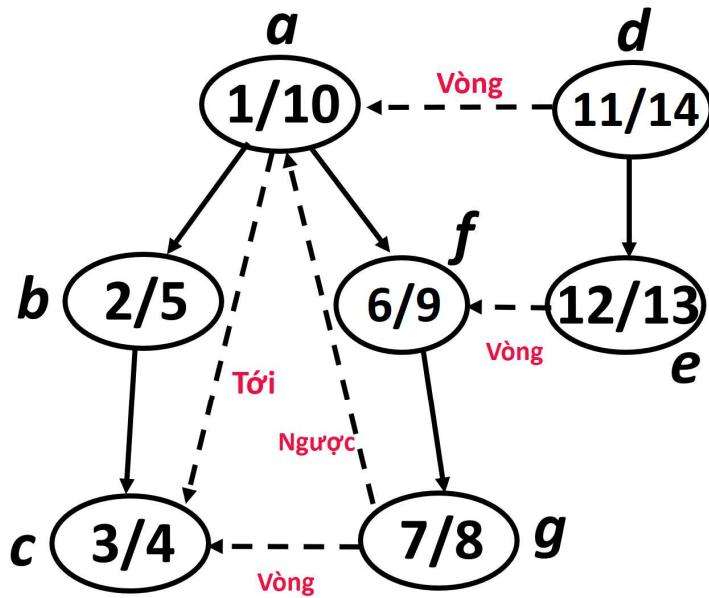


Bỏ đề về các khoảng lồng nhau

Cho đồ thị có hướng $G = (V, E)$, và cây DFS bất kỳ của G và hai đỉnh u, v tùy ý của nó. Khi đó

- u là con cháu của v khi và chỉ khi $[d[u], f[u]] \subseteq [d[v], f[v]]$
- u là tổ tiên của v khi và chỉ khi $[d[u], f[u]] \supseteq [d[v], f[v]]$
- u và v không có quan hệ họ hàng khi và chỉ khi $[d[u], f[u]]$ và $[d[v], f[v]]$ là không giao nhau.

- *Tree edge (cạnh cây)*: cạnh theo đó từ một đỉnh đến thăm đỉnh mới
- *Back edge (cạnh ngược)*: đi từ con cháu đến tổ tiên
- *Forward edge (cạnh tới)*: đi từ tổ tiên đến con cháu
- *Cross edge (cạnh vòng)*: giữa hai đỉnh không có họ hàng



DFS: Các loại cạnh

- DFS tạo ra một cách phân loại các cạnh của đồ thị đã cho:
 - *Tree edge (cạnh cây)*: cạnh theo đó từ một đỉnh đến thăm đỉnh mới
 - *Back edge (cạnh ngược)*: đi từ con cháu đến tổ tiên
 - *Forward edge (cạnh tới)*: đi từ tổ tiên đến con cháu
 - *Cross edge (cạnh vòng)*: giữa hai đỉnh không có họ hàng
- **Chú ý:** Cạnh của cây & cạnh ngược là quan trọng; nhiều thuật toán không đòi hỏi phân biệt cạnh tới và cạnh vòng

2. Duyệt đồ thị

2.1. Tìm kiếm theo chiều rộng

2.2. Tìm kiếm theo chiều sâu

2.3. Một số ứng dụng

Các ứng dụng của BFS/DFS

1. Sắp xếp topo
2. Định hướng đồ thị
3. Đường đi từ s đến t
4. Đồ thị hai phía
5. Tìm đường đi dài nhất trên cây
6. Bài toán tìm bao đóng truyền ứng
7. Phát hiện chu trình
8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông
9. Tính liên thông của đồ thị
10. Kiểm tra tính liên thông mạnh

1. Sắp xếp topo

Một ứng dụng điển hình của thứ tự tô pô là lập kế hoạch cho một chuỗi các công việc:

- Mỗi đỉnh của đồ thị biểu diễn 1 công việc cần thực hiện.
- Các công việc có phụ thuộc lẫn nhau. Do đó 1 số việc không thể thực hiện trước khi việc khác hoàn thành.

Ví dụ:

- IT3302 (C basic) là điều kiện bắt buộc khi học IT3312 (C advanced)



Cho đồ thị có hướng không có chu trình, hãy đưa ra 1 trật tự các công việc sao cho các ràng buộc về trình tự thực hiện các công việc (được thể hiện qua hướng ở các cung trên đồ thị) được bảo toàn.

Chú ý: có thể có hơn 1 lịch trình thực hiện

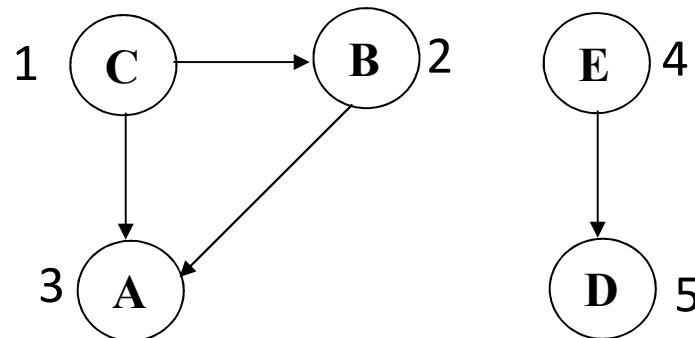
1. Sắp xếp topo

Thứ tự tô pô của một đồ thị có hướng là một thứ tự sắp xếp của các đỉnh sao cho với mọi cung từ u đến v trong đồ thị, u luôn nằm trước v . Thuật toán để tìm thứ tự tô pô gọi là **thuật toán sắp xếp tô pô**. Thứ tự tô pô tồn tại khi và chỉ khi đồ thị không có chu trình (viết tắt là DAG - directed acyclic graph). Đồ thị có hướng không có chu trình luôn có ít nhất một thứ tự tô pô.

1. Bài toán sắp xếp tôpô (Topological Sort)

- **Bài toán đặt ra là:** Cho đồ thị có hướng không có chu trình (directed acyclic graph) $G = (V, E)$. Hãy tìm cách sắp xếp các đỉnh sao cho nếu có cạnh (u, v) thì u phải đi trước v trong thứ tự đó (nói cách khác, cần tìm cách đánh số các đỉnh của đồ thị sao cho mỗi cung của đồ thị luôn hướng từ đỉnh có chỉ số nhỏ hơn đến đỉnh có chỉ số lớn hơn).

Ví dụ:



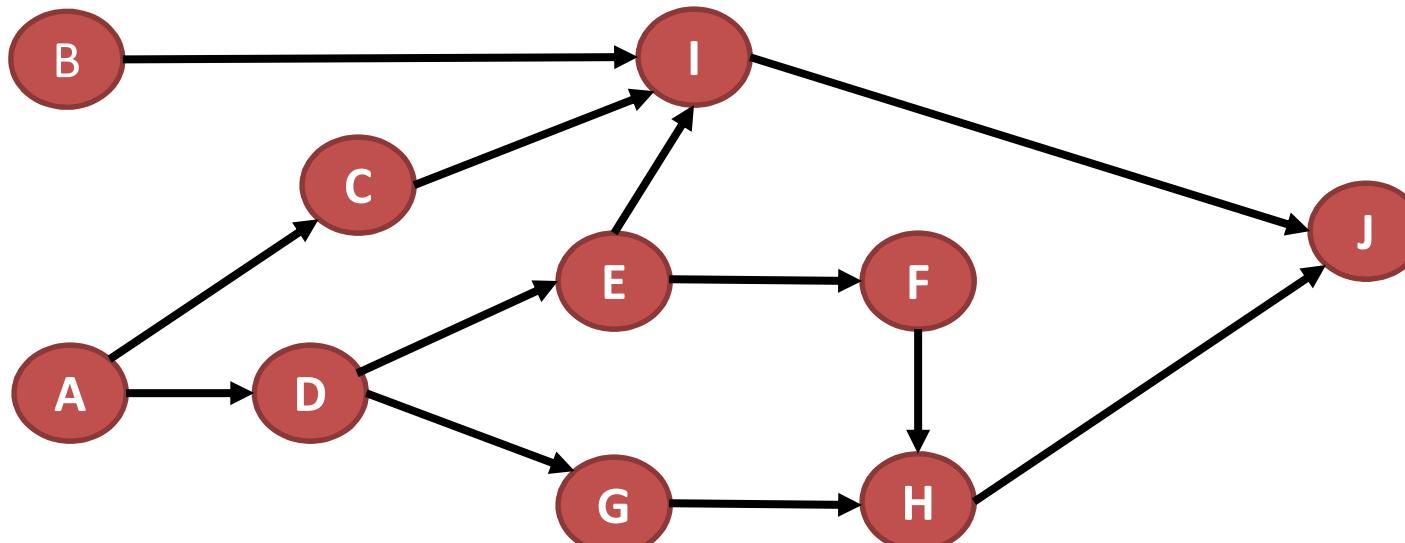
2 cách:

- Thuật toán DFS
- Thuật toán xóa dần đỉnh (chỉnh sửa BFS)

Ví dụ: Cải tạo văn phòng cho thuê

| Công đoạn | Các công đoạn trước | Thời gian(Tuần) |
|---|---------------------|-----------------|
| A: Chuẩn bị thiết kế ban đầu | - | 3 |
| B: Tìm kiếm khách hàng | - | 5 |
| C: Thiết kế tờ rơi | A | 3 |
| D: Chuẩn bị bản thiết kế cuối cùng | A | 8 |
| E: Xin giấy phép sửa chữa | D | 2 |
| F: Nhận tiền đầu tư từ ngân hàng | E | 3 |
| G: Lựa chọn khách hàng | D | 4 |
| H: Xây dựng sửa chữa | G, F | 17 |
| I: Hoàn thiện hợp đồng thuê mặt bằng | B, C, E | 13 |
| J: Khách hàng chuyển vào | I, H | 2 |

Hãy đưa ra trình tự thực hiện các công việc



(1) Dùng DFS

Thuật toán có thể mô tả ngắn tắt như sau:

- Thực hiện $\text{DFS}(G)$, khi mỗi đỉnh được duyệt xong ta đưa nó vào đầu danh sách liên kết (điều đó có nghĩa là những đỉnh kết thúc thăm càng muộn sẽ càng ở gần đầu danh sách hơn).
- Danh sách liên kết thu được khi kết thúc $\text{DFS}(G)$ sẽ cho ta thứ tự cần tìm.

Thuật toán sắp xếp tốpô: dùng DFS

(* Main Program*)

```
1. for  $s \in V$ 
2.   visited[ $s$ ]  $\leftarrow$  false
3. for  $s \in V$ 
4.   if (visited[ $s$ ] == false)
5.     DFS( $s$ )
```

DFS(s)

```
1.   visited[ $s$ ]  $\leftarrow$  true // Thăm đỉnh  $s$ 
2.   for each  $v \in Adj[s]$ 
3.     if (visited[ $v$ ] == false)
4.       DFS( $v$ )
```



TopoSort(G)

```
1. for  $s \in V$  visited[ $s$ ] = false; // khởi tạo
2.  $L = \text{new(linked\_list)}$ ; // khởi tạo danh sách liên kết rỗng  $L$ 
3. for  $s \in V$ 
4.   if (visited[ $s$ ] == false) DFS( $s$ );
5. return  $L$ ; //  $L$  cho thứ tự cần tìm
```

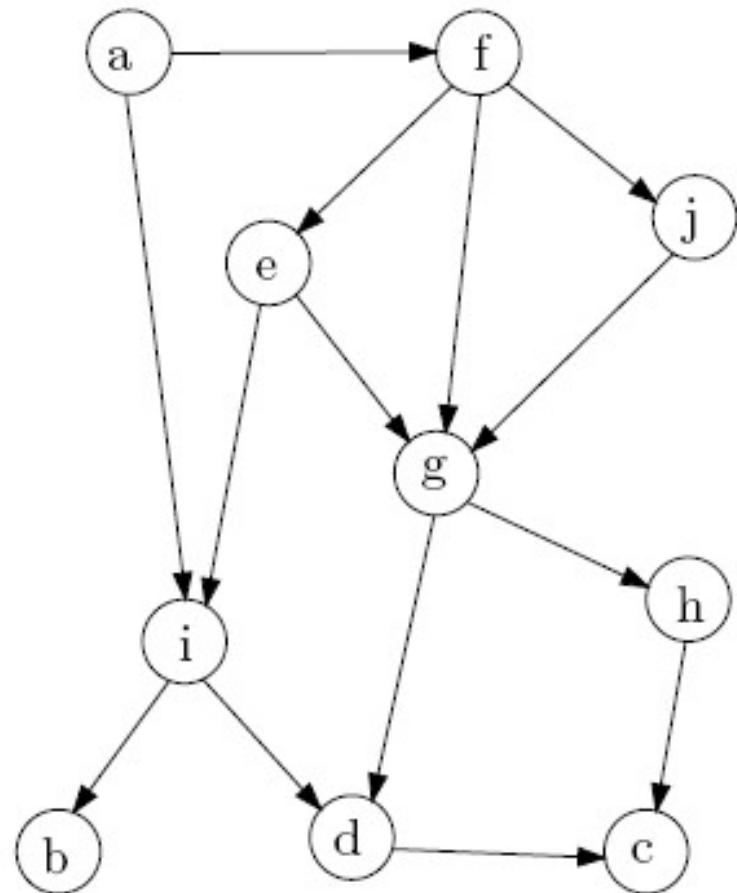
DFS (s) { // Bắt đầu tìm kiếm từ s

```
1. visited[ $s$ ] = true; // Đánh dấu  $s$  là đã thăm
2. for  $v \in Adj(s)$ 
3.   if (visited[ $v$ ] == false) DFS( $v$ );
4. Nạp  $s$  vào đầu danh sách  $L$  //  $s$  đã duyệt xong
```

}

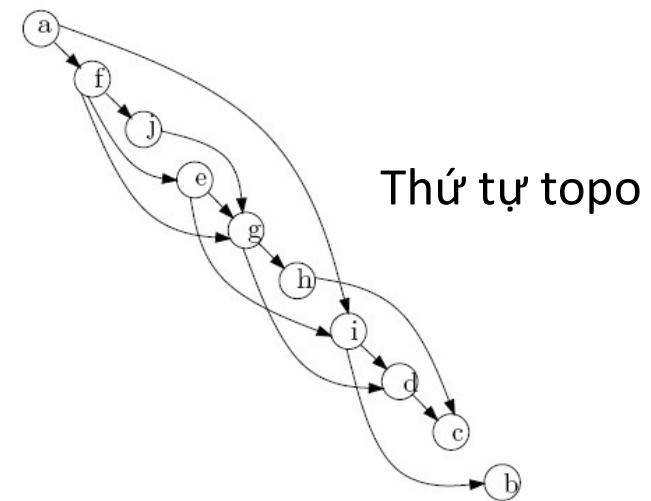
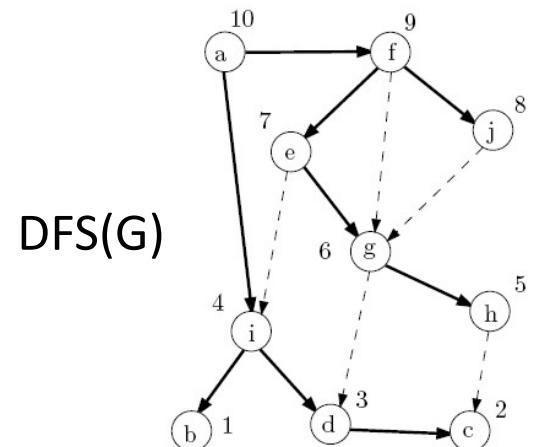
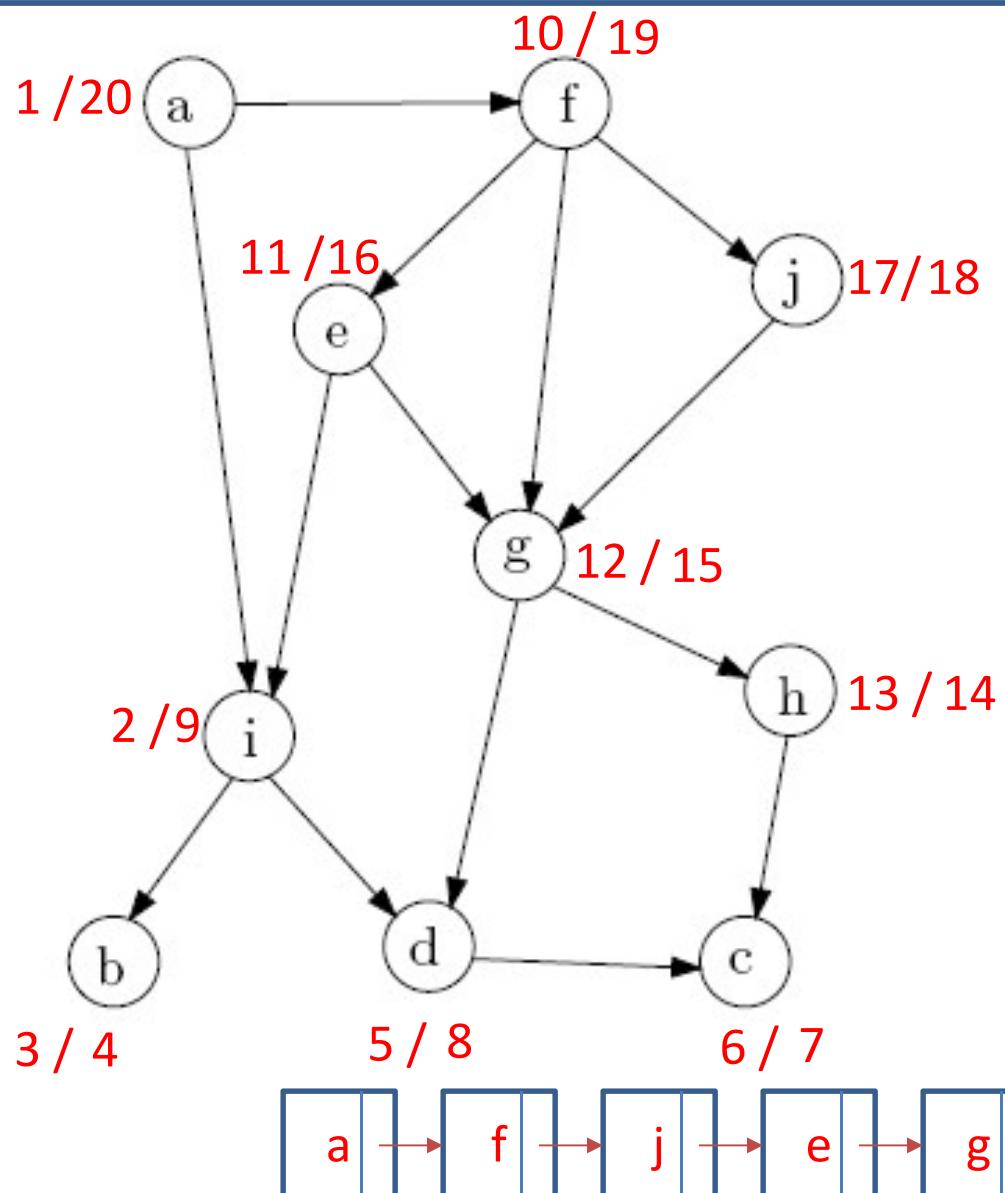
Thời gian tính của TopoSort(G) là $O(|V|+|E|)$.

Ví dụ: Thực hiện sắp xếp topo trên đồ thị G bằng thuật toán DFS

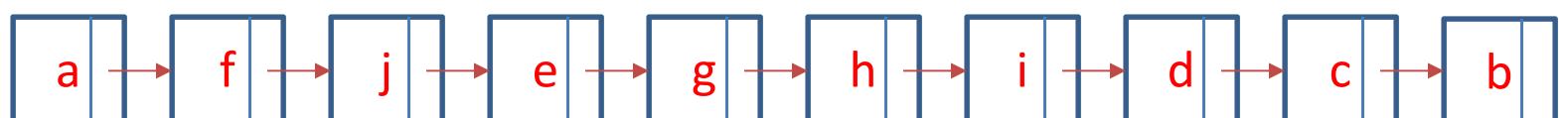


Đồ thị G

Ví dụ: Thực hiện sắp xếp topo trên đồ thị G bằng thuật toán DFS



Danh sách liên kết L:



Ví dụ: Thực hiện sắp xếp topo trên đồ thị G bằng thuật toán DFS

DFS(0)
DFS(2)
DFS(4)
DFS(3)
DFS(7)

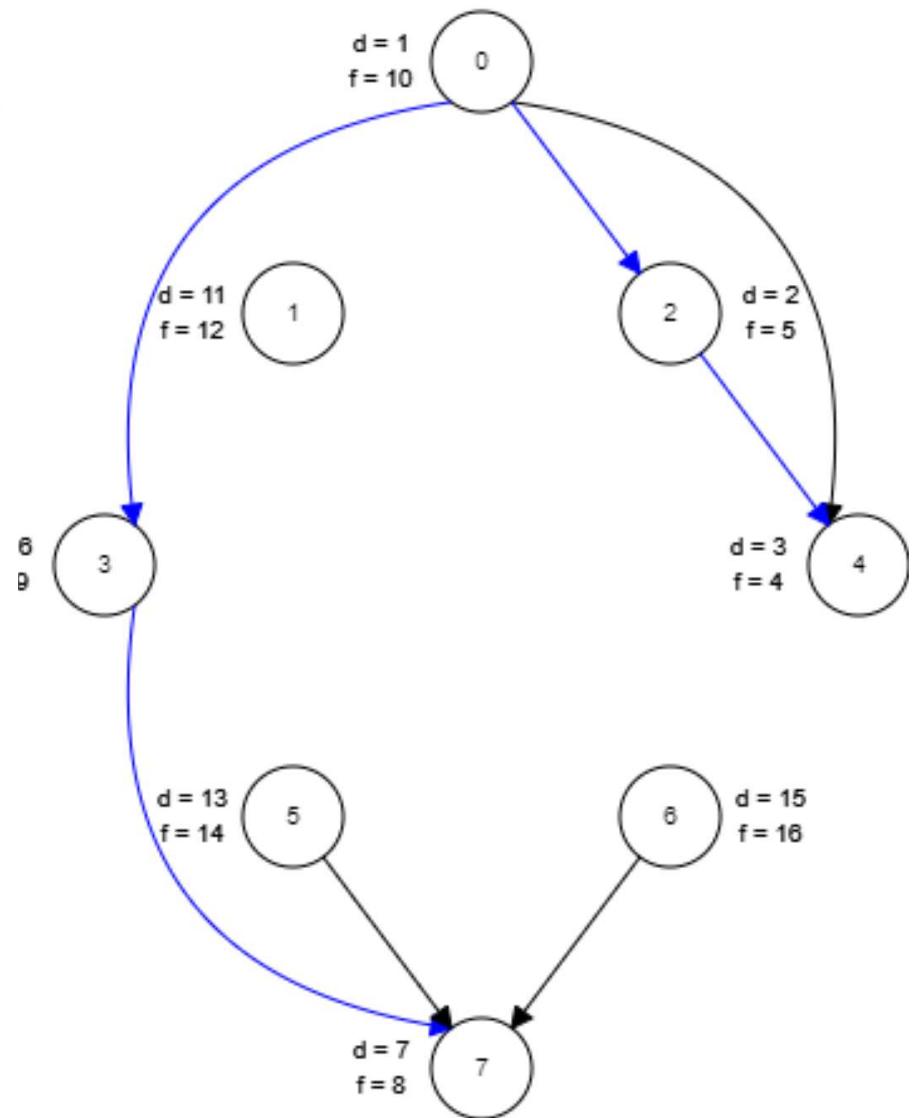
DFS(1)

DFS(5)

DFS(6)

Topological Order

| |
|---|
| 6 |
| 5 |
| 1 |
| 0 |
| 3 |
| 7 |
| 2 |
| 4 |



TopoSort(G)

```
1. for  $s \in V$  visited[ $s$ ] = false; // khởi tạo  
2.  $L$  = new(linked_list); // khởi tạo danh sách liên kết rỗng  $L$   
3. for  $s \in V$   
4.     if (visited[ $s$ ] == false) DFS( $s$ );  
5. return  $L$ ; //  $L$  cho thứ tự cần tìm
```

```
DFS ( $s$ ) { // Bắt đầu tìm kiếm từ  $s$   
1. visited[ $s$ ] = true; // Đánh dấu  $s$  là đã thăm  
2. for  $v \in \text{Adj}(s)$   
3.     if (visited[ $v$ ] == false) DFS( $v$ );  
4. Nạp  $s$  vào đầu danh sách  $L$  //  $s$  đã duyệt xong  
}
```

```
void Graph::topoSort()  
{  
    for (int  $s = 0$ ;  $s < V$ ;  $s++$ ) visited[ $s$ ] = false;  
    stack<int> Stack;  
  
    for (int  $s = 0$ ;  $s < V$ ;  $s++$ )  
        if (visited[ $s$ ] == false) DFS_topo( $s$ , Stack);  
  
    // In kết quả:  
    while (Stack.empty() == false)  
    {  
        cout << Stack.top() << " ";  
        Stack.pop();  
    }  
}
```

```
void Graph::DFS_topo(int  $s$ , stack<int> &Stack)  
{  
    visited[ $s$ ] = true;  
    list<int>::iterator i;  
    // Duyệt qua danh sách kề của đỉnh  $s$ :  
    for (i = adj[ $s$ ].begin(); i != adj[ $s$ ].end(); ++i)  
    {  
        int v = *i;  
        if (!visited[v]) DFS_topo(v, Stack);  
    }  
    Stack.push( $s$ ); // Day đỉnh  $s$  vào ngăn xếp
```

Tìm kiếm theo chiều sâu (Depth-First Search)

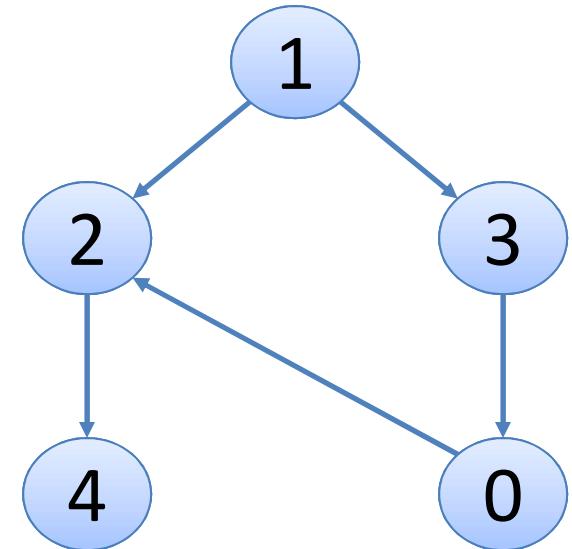
```
int main()
{
    // Tao do thi co huong gom 5 dinh va 5 canh
    Graph g(5);
    g.addEdge(1, 2); g.addEdge(1, 3);
    g.addEdge(2, 4); g.addEdge(3, 0); g.addEdge(0, 2);

    // Khoi tao:
    int numV = g.numVertex();
    visited = new bool[numV];
    for(int i = 0; i < numV; i++) visited[i] = false;
    cout << "BFS(1): ";
    g.BFS(1); cout<<endl;

    for(int i = 0; i < numV; i++) visited[i] = false;
    cout << "DFS(1): ";
    g.DFS(1); cout<<endl;

    cout << "Thu tu topo : ";
    g.topoSort();

    return 0;
}
```



```
BFS(1): 1 2 3 4 0
DFS(1): 1 2 4 3 0
Thu tu topo : 1 3 0 2 4
```

(2) Dùng thuật toán xoá dần đỉnh

Một thuật toán khác để thực hiện sắp xếp tôpô được xây dựng dựa trên mệnh đề sau:

- **Mệnh đề.** Giả sử G là đồ thị có hướng không có chu trình.

Khi đó

- 1) Mọi đồ thị con H của G đều là đồ thị phi chu trình.
- 2) Bao giờ cũng tìm được đỉnh có bán bậc vào bằng 0.

Sắp xếp tôpô: dùng thuật toán xoá dần đỉnh

- Từ mệnh đề ta suy ra thuật toán xoá dần đỉnh để thực hiện sắp xếp tôpô sau đây:
 - Thoạt tiên, tìm các đỉnh có bán bậc vào bằng 0. Rõ ràng ta có thể đánh số chúng theo một thứ tự tùy ý bắt đầu từ 1.
 - Tiếp theo, loại bỏ khỏi đồ thị những đỉnh đã được đánh số cùng các cung đi ra khỏi chúng, ta thu được đồ thị mới cũng không có chu trình, và thủ tục được lặp lại với đồ thị mới này.
 - Quá trình đó sẽ được tiếp tục cho đến khi tất cả các đỉnh của đồ thị được đánh số.

Sắp xếp tôpô: dùng thuật toán xoá dần đỉnh

for $v \in V$ **do**

Tính Indegree[v] – bán bậc vào của đỉnh v ;

Q = hàng đợi chứa tất cả các đỉnh có bán bậc vào = 0;

$num=0$;

while $Q \neq \emptyset$ **do**

$v = \text{dequeue}(Q)$; $num=num+1$;

Đánh số đỉnh v bởi num ;

for $u \in \text{Adj}(v)$ **do**

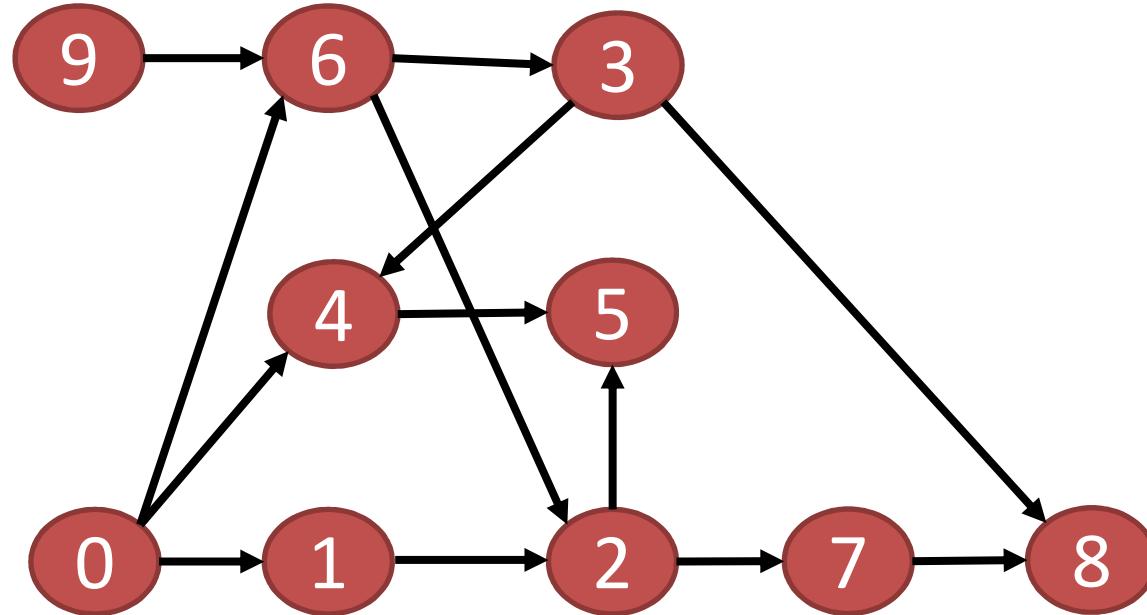
Degree[u]=Degree[u] -1;

if Degree[u]==0

Enqueue(Q,u);

Thời gian tính: $\Theta(|V|+|E|)$

Sắp xếp topo: Ví dụ

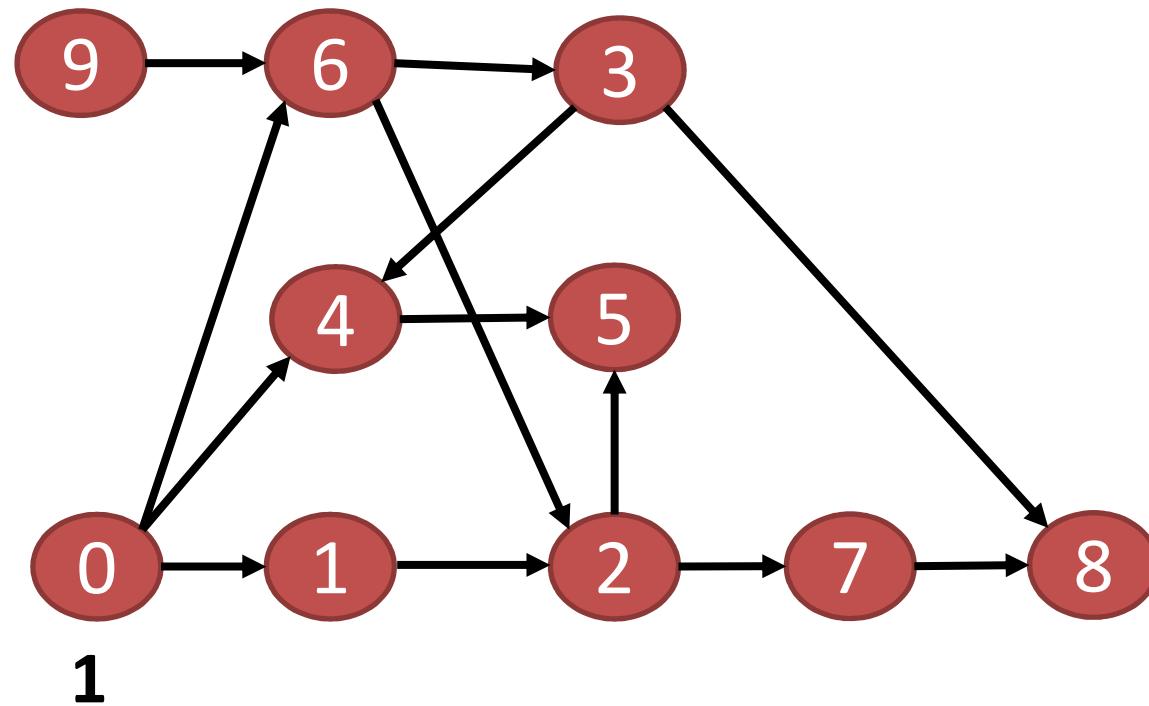


$$Q = \{0, 9\}$$

Output:

| | Indegree | | |
|---|----------|---|---|
| 0 | 0 | 6 | 1 |
| 1 | 1 | 2 | |
| 2 | 2 | 7 | 5 |
| 3 | 1 | 8 | 4 |
| 4 | 2 | 5 | |
| 5 | 2 | | |
| 6 | 2 | 3 | 2 |
| 7 | 1 | 8 | |
| 8 | 2 | | |
| 9 | 0 | 6 | |

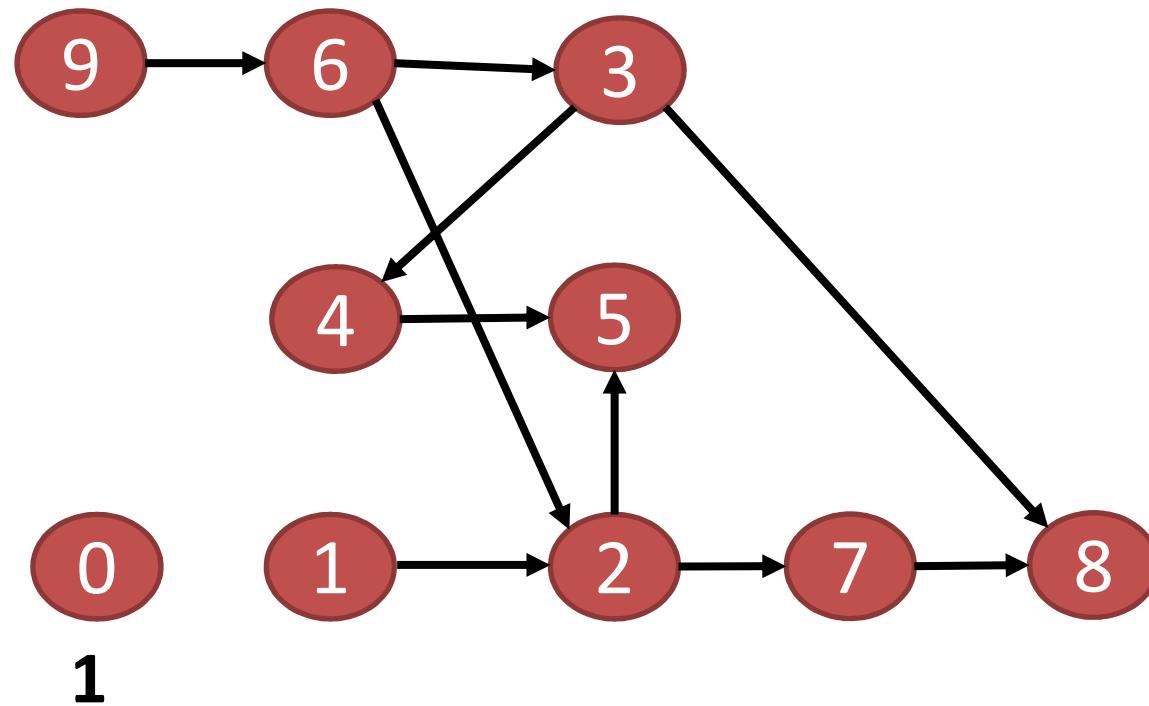
Sắp xếp topo: Ví dụ



Output: 0

| | Indegree | | |
|---|----------|---|---|
| 0 | 6 | 1 | 4 |
| 1 | 2 | | |
| 2 | 7 | 5 | |
| 3 | 8 | 4 | |
| 4 | 5 | | |
| 5 | | | |
| 6 | 3 | 2 | |
| 7 | 8 | | |
| 8 | | | |
| 9 | 6 | | |

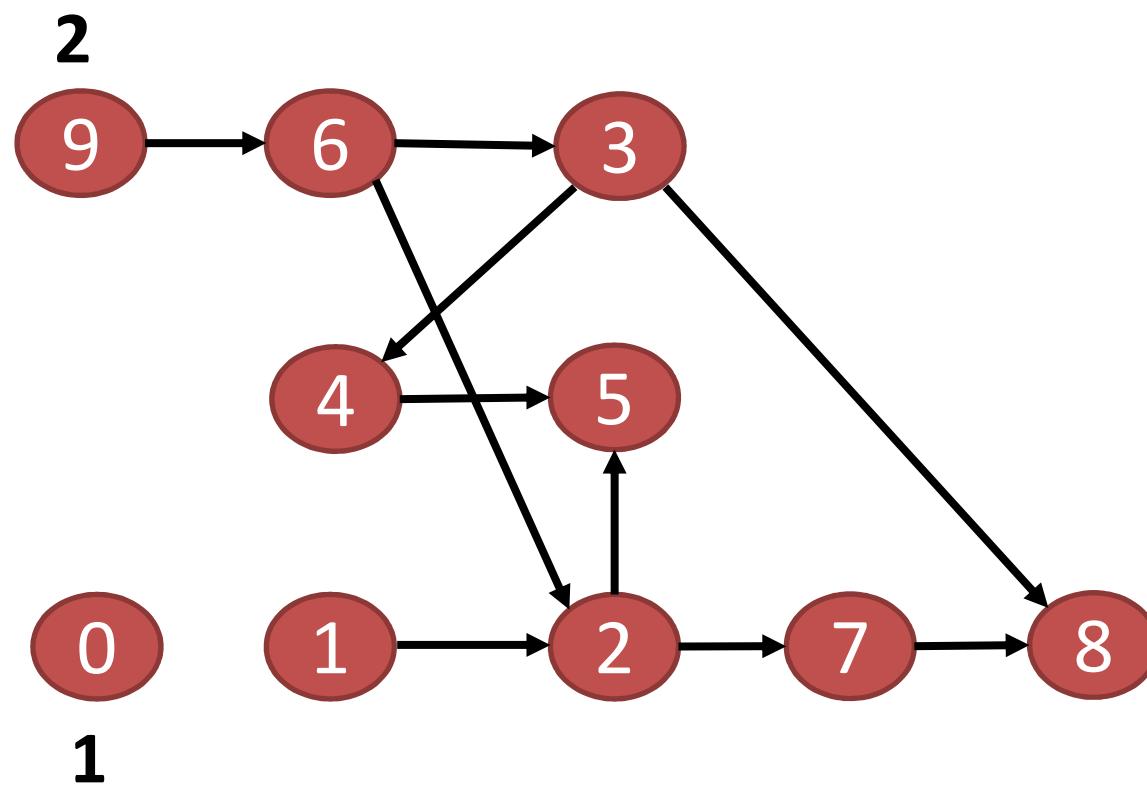
Sắp xếp topo: Ví dụ



Output: 0

| | Indegree | | |
|---|----------|---|---|
| 0 | 6 | 1 | 4 |
| 1 | 2 | | |
| 2 | 7 | 5 | |
| 3 | 8 | 4 | |
| 4 | 5 | | |
| 5 | | | |
| 6 | 3 | 2 | |
| 7 | 8 | | |
| 8 | | | |
| 9 | 6 | | |

Sắp xếp topo: Ví dụ

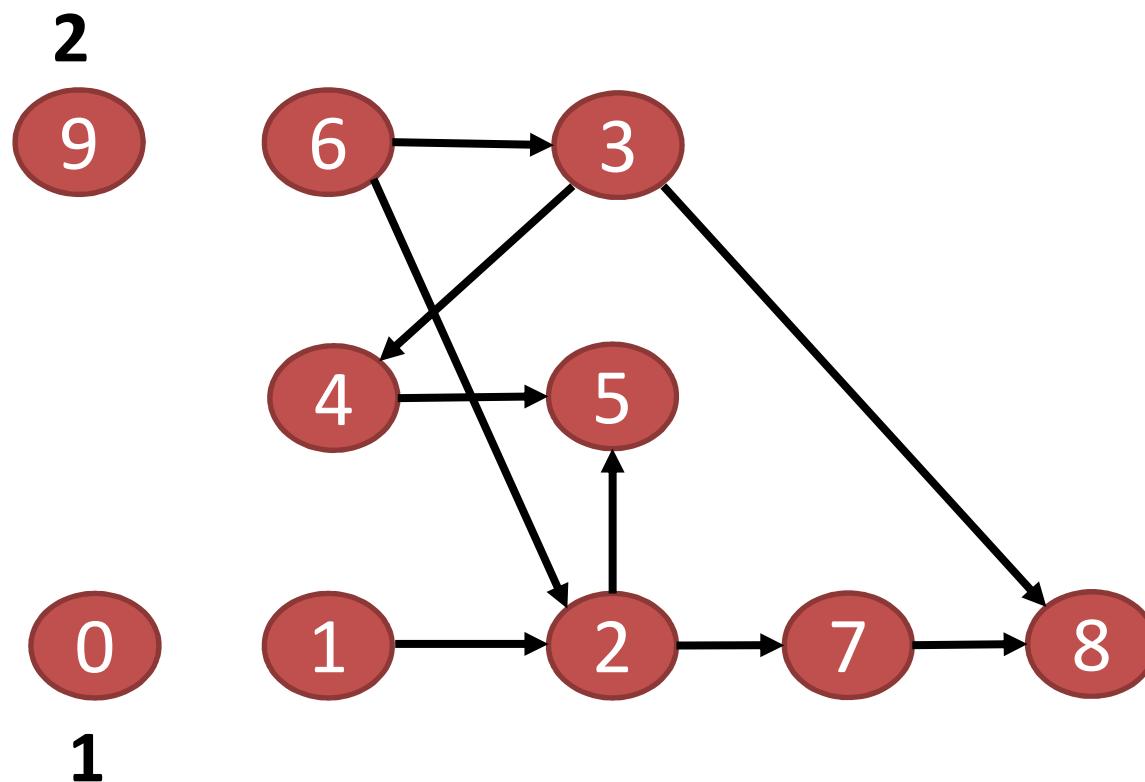


| | Indegree | | |
|---|----------|---|---|
| 0 | 6 | 1 | 4 |
| 1 | 2 | | |
| 2 | 7 | 5 | |
| 3 | 8 | 4 | |
| 4 | 5 | | |
| 5 | | | |
| 6 | 3 | 2 | |
| 7 | 8 | | |
| 8 | | | |
| 9 | 6 | | |

-1

Output: 0 9

Sắp xếp topo: Ví dụ

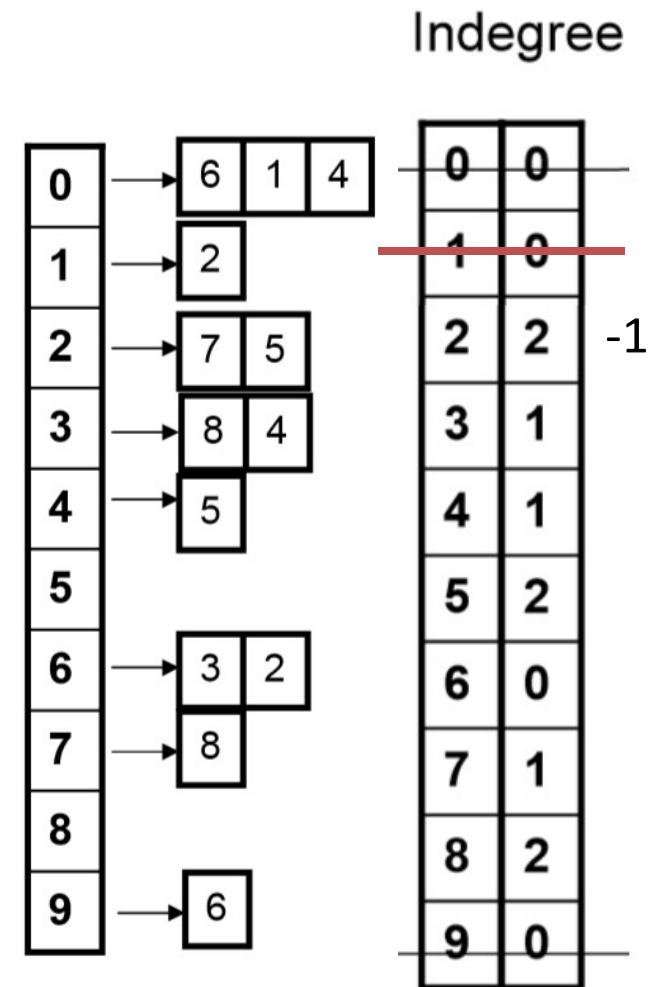
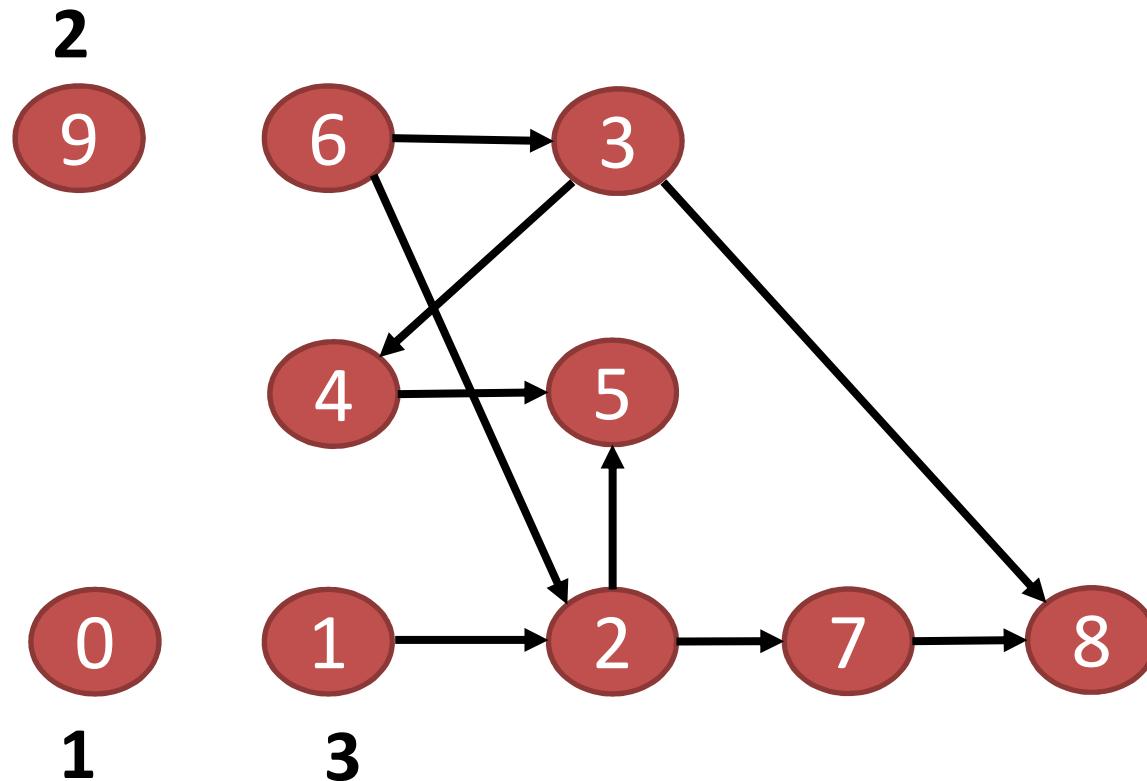


Dequeue 9; Q={1}6}

Output: 0 9

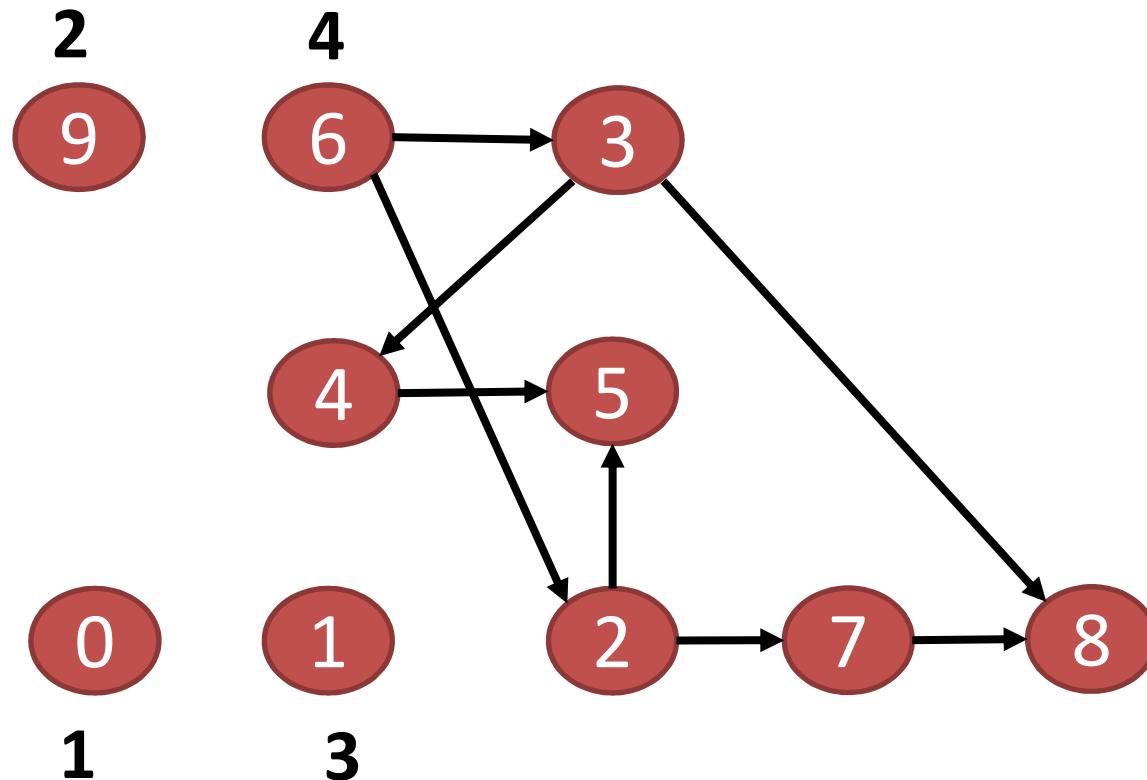
| | Indegree | | |
|---|----------|---|---|
| 0 | 0 | 6 | 1 |
| 1 | 1 | 0 | 2 |
| 2 | 2 | 2 | 3 |
| 3 | 3 | 1 | 4 |
| 4 | 4 | 1 | 5 |
| 5 | 5 | 2 | 6 |
| 6 | 6 | 1 | 7 |
| 7 | 7 | 1 | 8 |
| 8 | 8 | 2 | 9 |
| 9 | 9 | 0 | |

Sắp xếp topo: Ví dụ



Output: 0 9 1

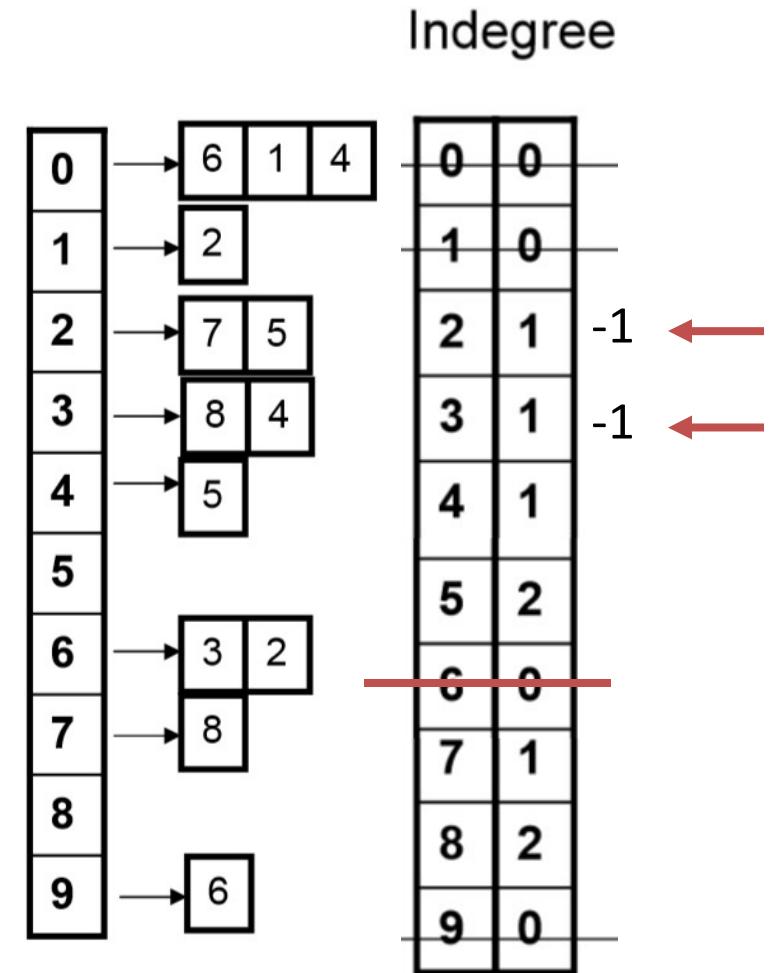
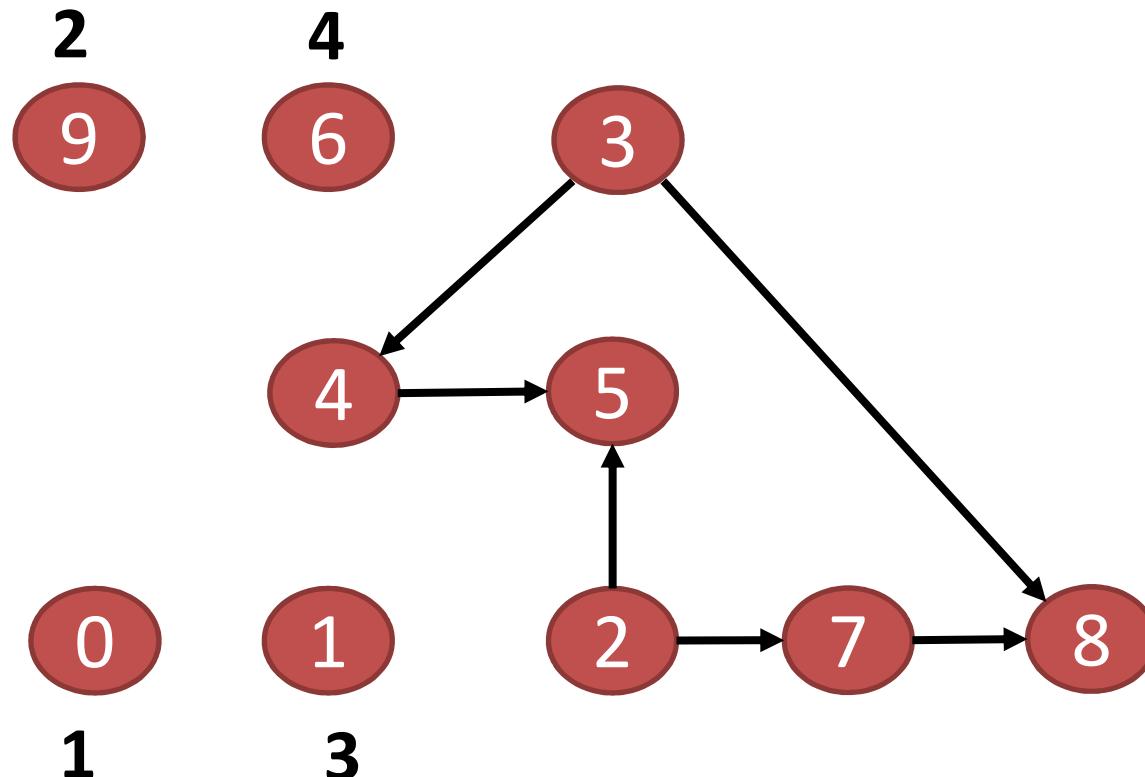
Sắp xếp topo: Ví dụ



| | Indegree | | |
|---|----------|---|----|
| 0 | 0 | 0 | -1 |
| 1 | 1 | 0 | -1 |
| 2 | 2 | 1 | -1 |
| 3 | 3 | 1 | -1 |
| 4 | 4 | 1 | |
| 5 | 5 | 2 | |
| 6 | 3 | 2 | |
| 7 | 8 | 1 | |
| 8 | 8 | 2 | |
| 9 | 9 | 0 | |

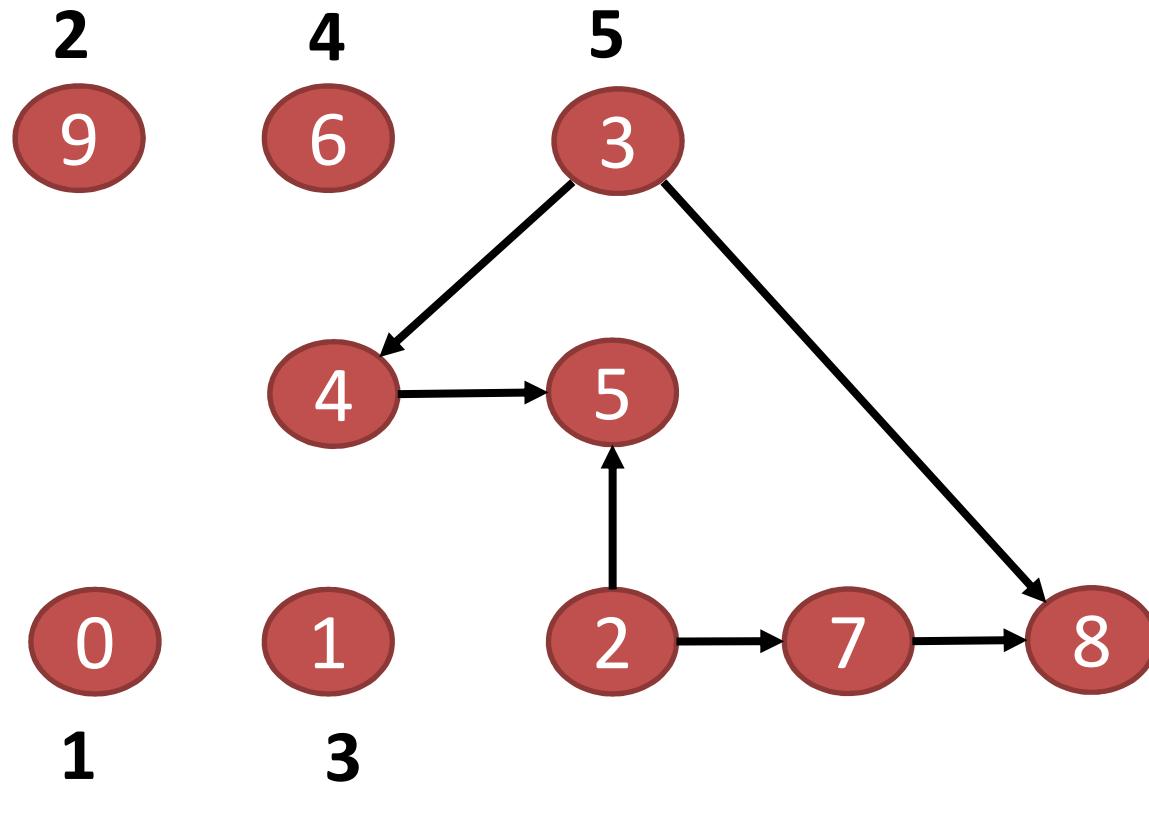
Output: 0 9 1 6

Sắp xếp topo: Ví dụ



Output: 0 9 1 6

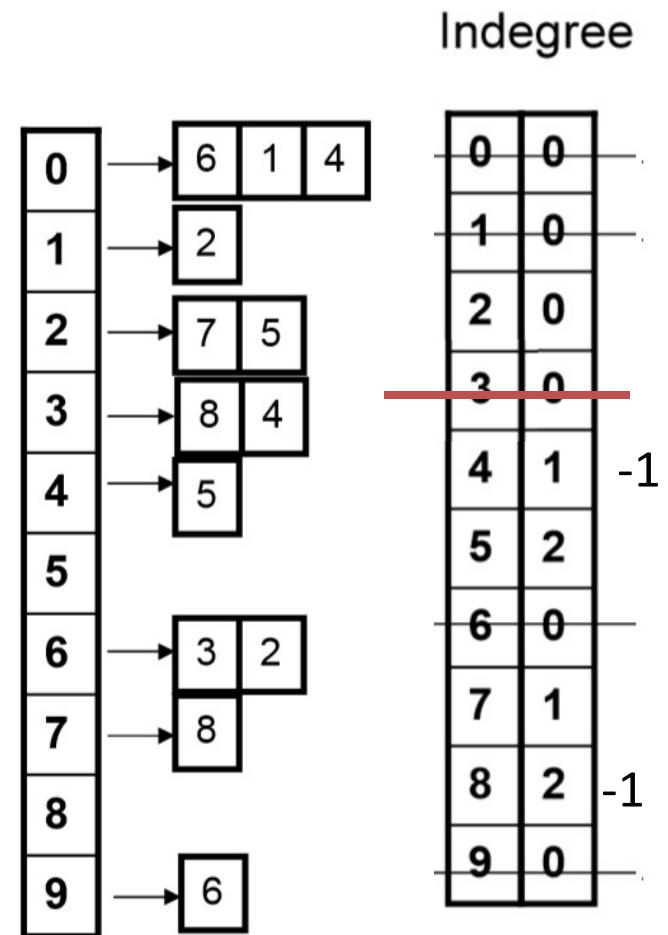
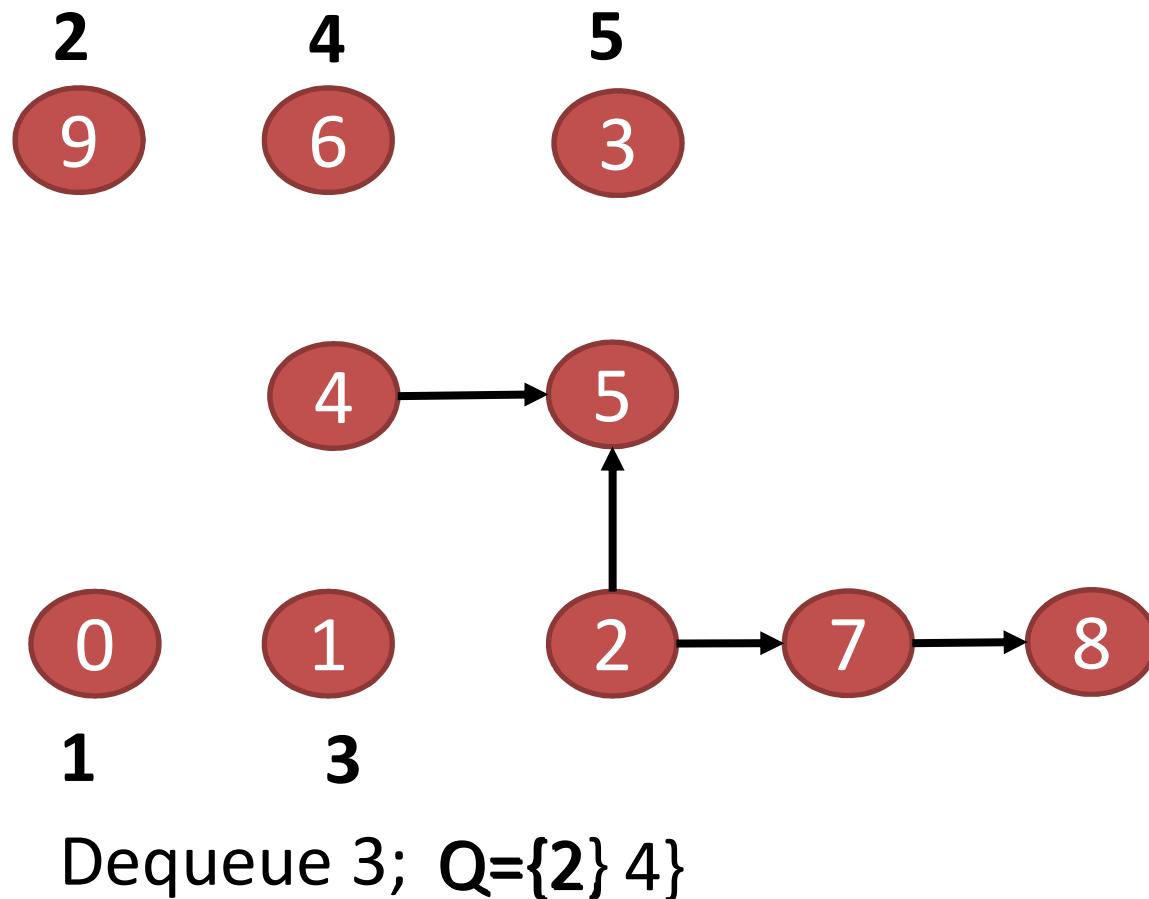
Sắp xếp topo: Ví dụ



| | Indegree | | |
|---|----------|---|----|
| 0 | 0 | 0 | -1 |
| 1 | 1 | 0 | -1 |
| 2 | 2 | 0 | -1 |
| 3 | 3 | 0 | -1 |
| 4 | 4 | 1 | -1 |
| 5 | 5 | 2 | -1 |
| 6 | 6 | 0 | -1 |
| 7 | 7 | 1 | -1 |
| 8 | 8 | 2 | -1 |
| 9 | 9 | 0 | -1 |

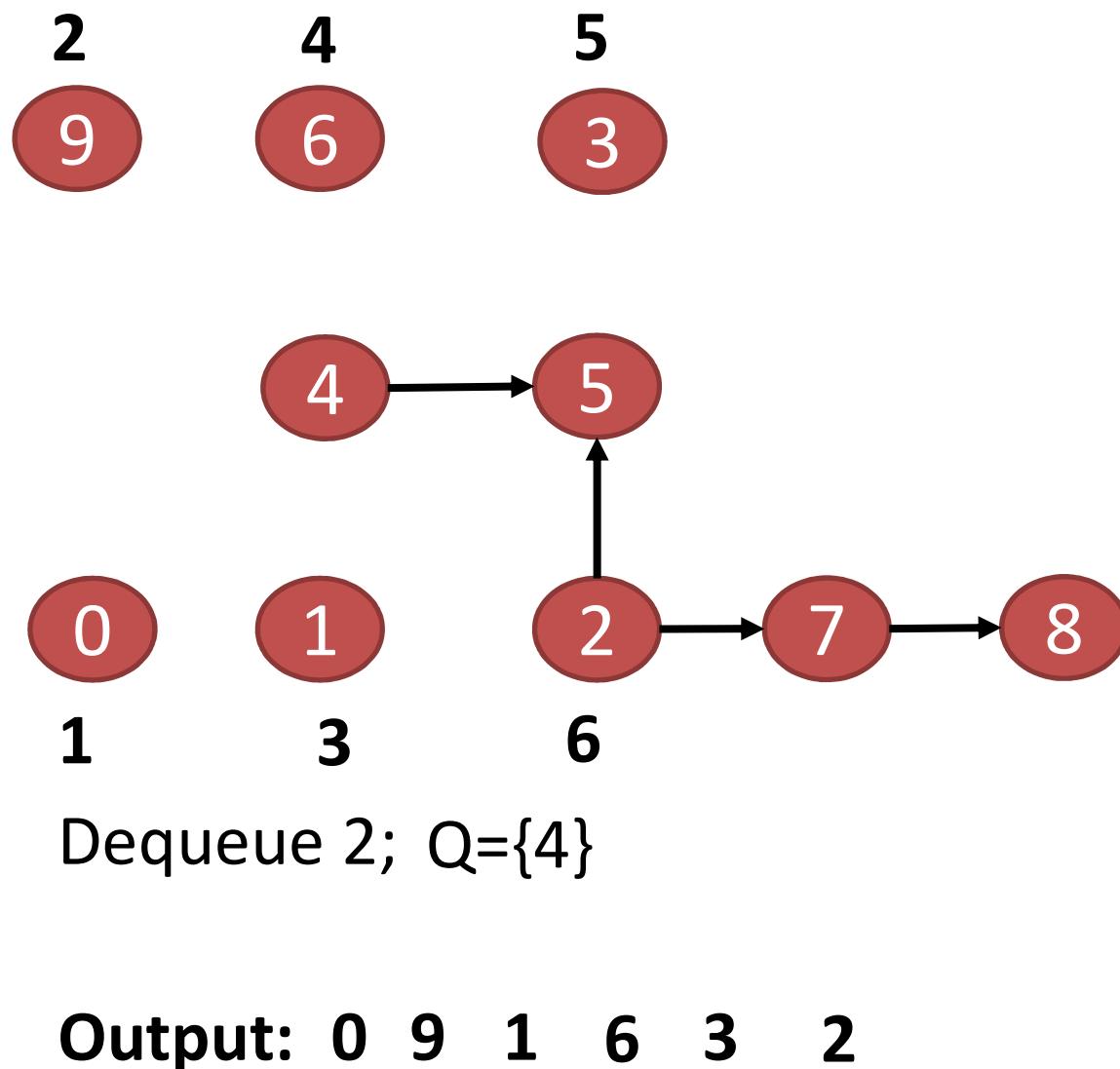
Output: 0 9 1 6 3

Sắp xếp topo: Ví dụ



Output: 0 9 1 6 3

Sắp xếp topo: Ví dụ



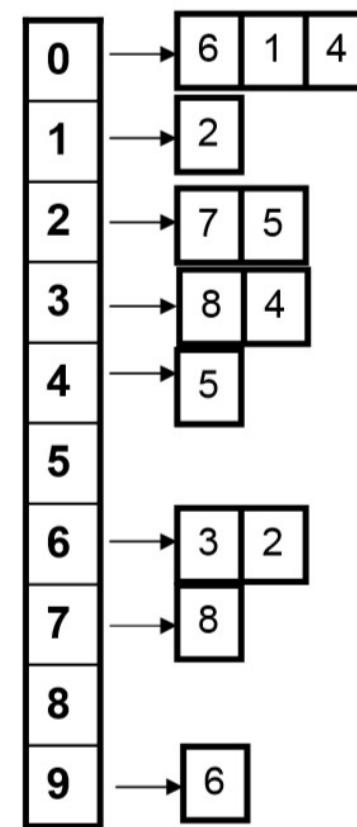
| | Indegree | | |
|---|----------|---|----|
| 0 | 0 | 0 | -1 |
| 1 | 1 | 0 | -1 |
| 2 | 2 | 0 | -1 |
| 3 | 3 | 0 | -1 |
| 4 | 4 | 0 | -1 |
| 5 | 5 | 2 | -1 |
| 6 | 6 | 0 | -1 |
| 7 | 7 | 1 | -1 |
| 8 | 8 | 1 | -1 |
| 9 | 9 | 0 | -1 |

Sắp xếp topo: Ví dụ



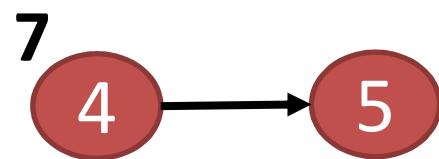
Dequeue 2; Q={4} 7}

Output: 0 9 1 6 3 2

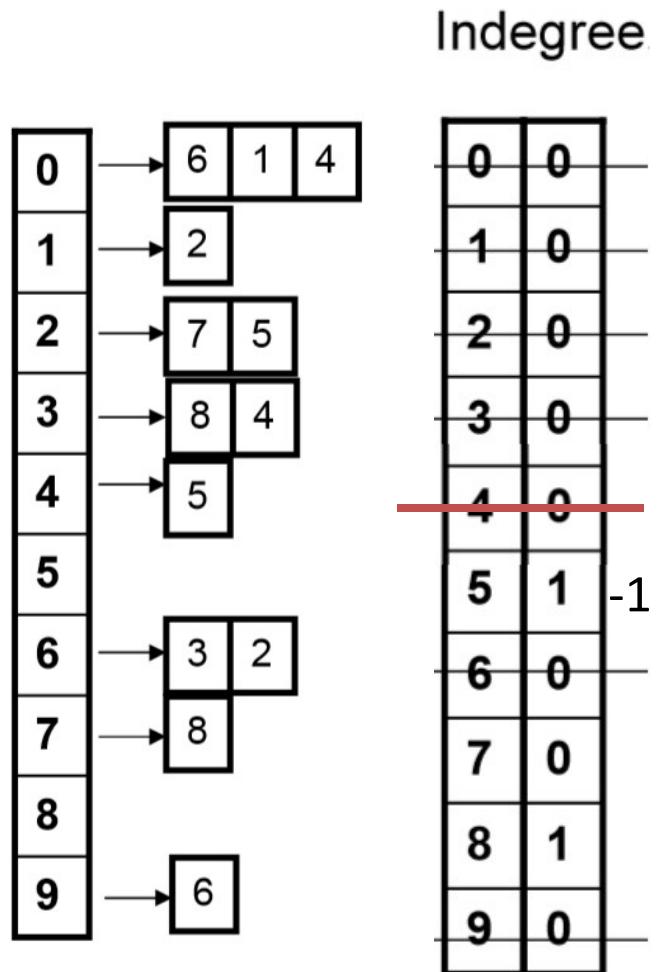


| | Indegree | |
|---|----------|----|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 2 | 0 | 0 |
| 3 | 0 | 0 |
| 4 | 0 | 0 |
| 5 | 2 | 0 |
| 6 | 0 | 0 |
| 7 | 1 | -1 |
| 8 | 1 | -1 |
| 9 | 0 | 0 |

Sắp xếp topo: Ví dụ

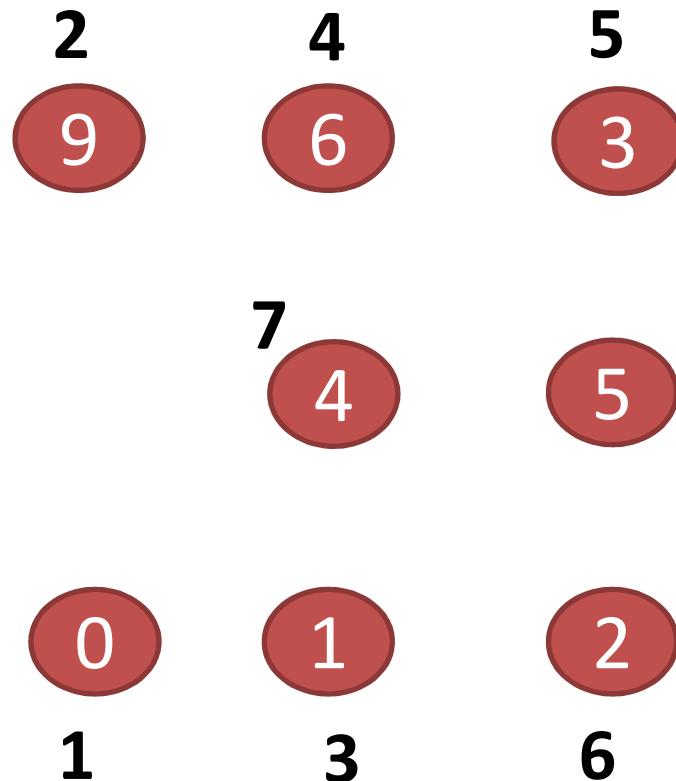


Dequeue 4; Q={7}



Output: 0 9 1 6 3 2 4

Sắp xếp topo: Ví dụ



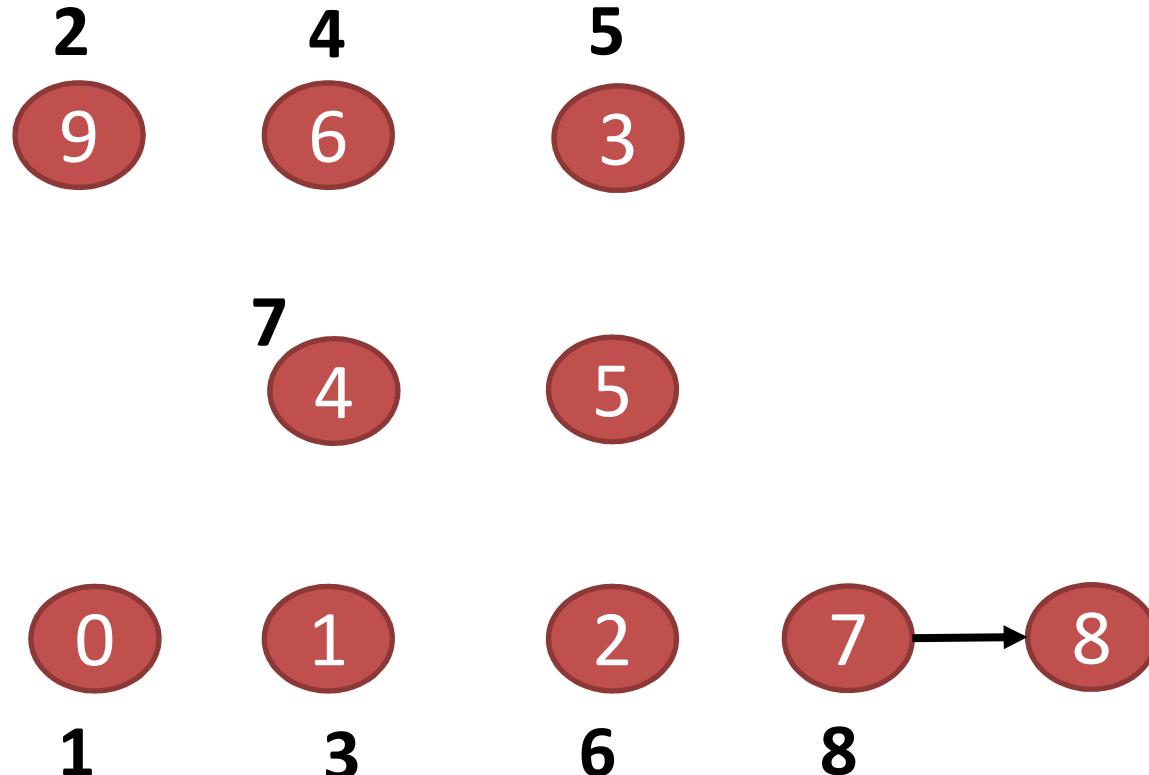
Dequeue 4; Q={7} 5}

| | Indegree | | |
|---|----------|---|----|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 2 | 2 | 0 | 0 |
| 3 | 3 | 0 | 0 |
| 4 | 4 | 0 | 0 |
| 5 | 5 | 1 | -1 |
| 6 | 6 | 0 | 0 |
| 7 | 7 | 0 | 0 |
| 8 | 8 | 1 | 0 |
| 9 | 9 | 0 | 0 |

The diagram shows a topological sort process. On the left, a vertical stack of boxes represents the current state of the sort. Arrows point from each box to its corresponding value in the stack. The stack contains the following sequence of values: 6, 1, 4, 2, 7, 5, 8, 4, 5, 6. To the right of the stack is a table showing the indegree for each node. The last row of the table, corresponding to node 4, is highlighted with a red line. This indicates that node 4 has an indegree of 0, meaning it has no incoming edges and can be safely dequeued.

Output: 0 9 1 6 3 2 4

Sắp xếp topo: Ví dụ

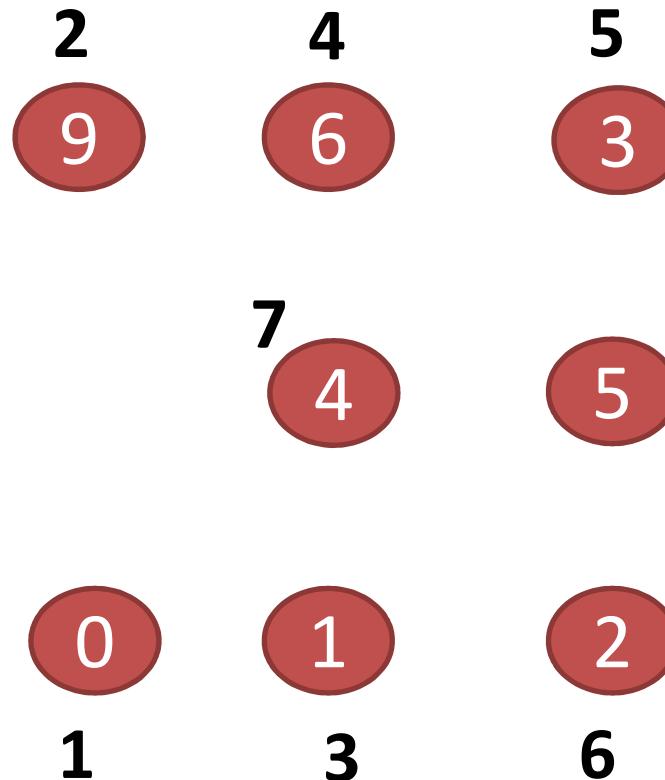


| | Indegree | | |
|---|----------|---|---|
| 0 | 6 | 1 | 4 |
| 1 | 2 | | |
| 2 | 7 | 5 | |
| 3 | 8 | 4 | |
| 4 | 5 | | |
| 5 | | | |
| 6 | 3 | 2 | |
| 7 | 8 | | |
| 8 | | | |
| 9 | 6 | | |

| | |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 1 |
| 9 | 0 |

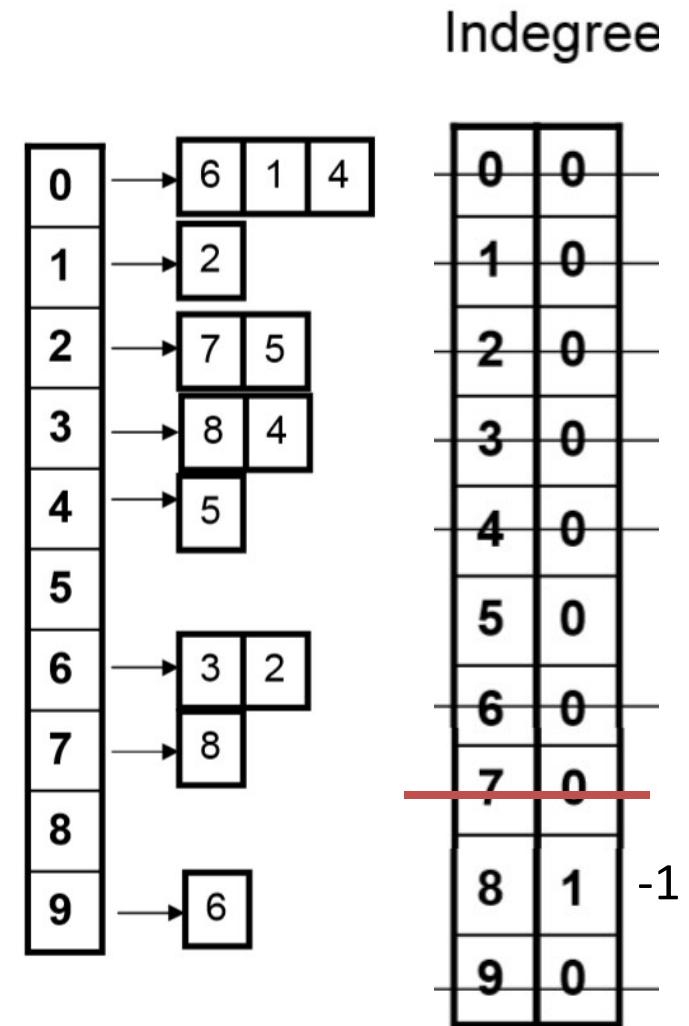
Output: 0 9 1 6 3 2 4 7

Sắp xếp topo: Ví dụ

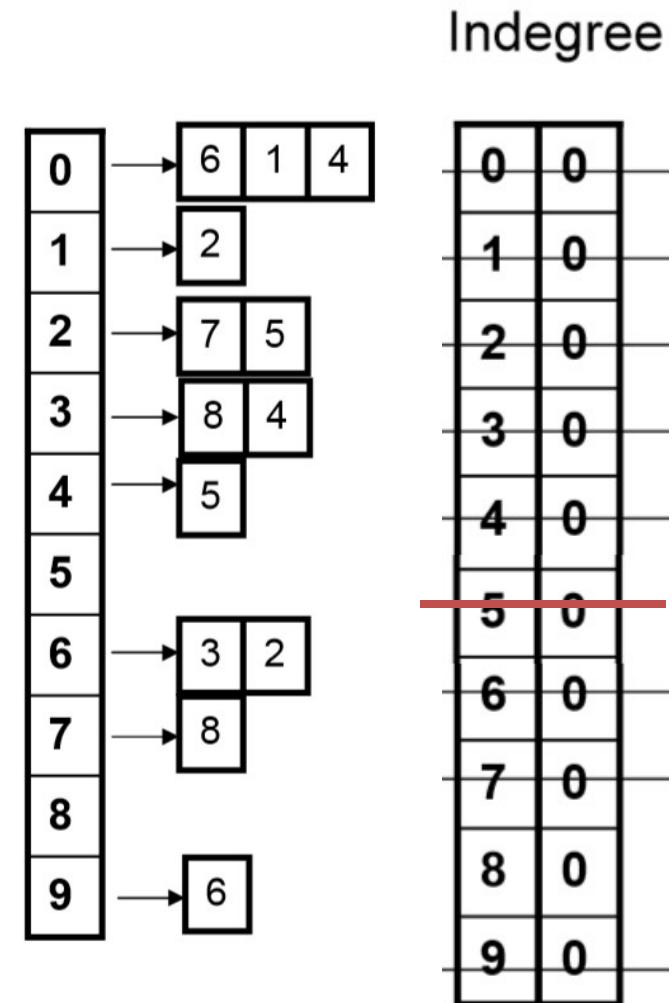
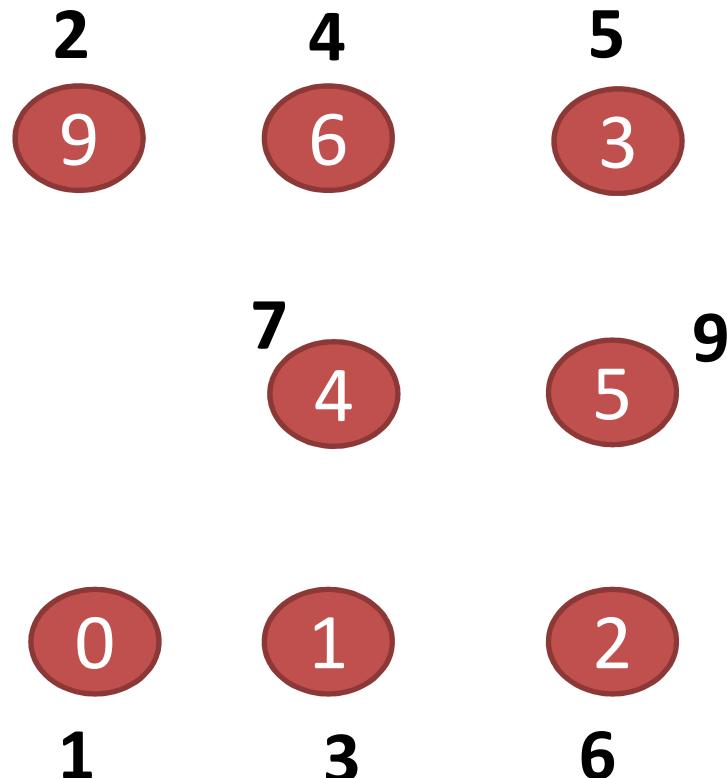


Dequeue 7; Q={5}8}

Output: 0 9 1 6 3 2 4 7

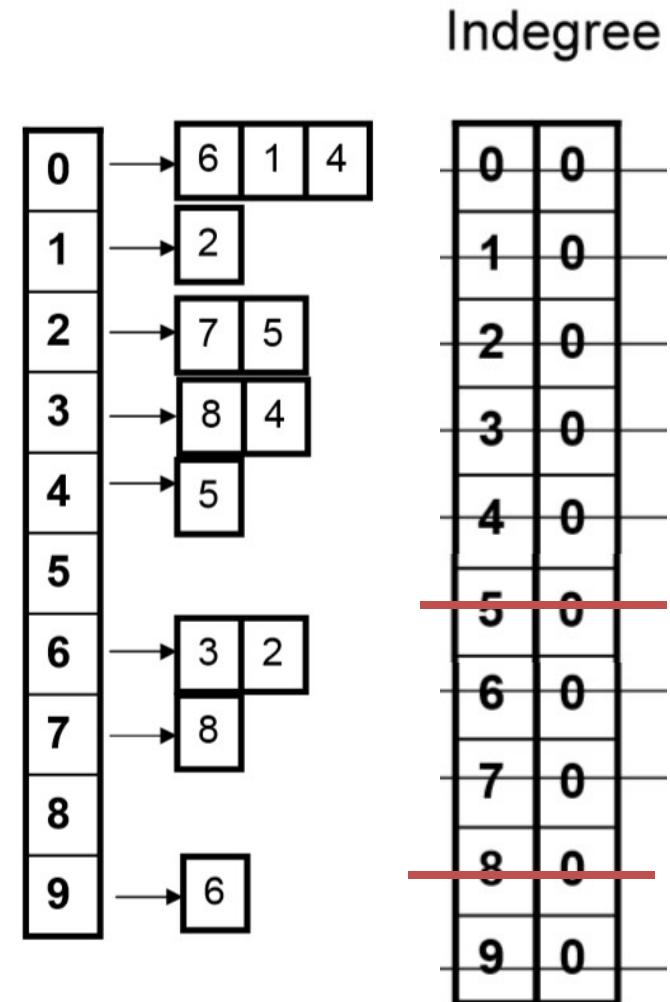
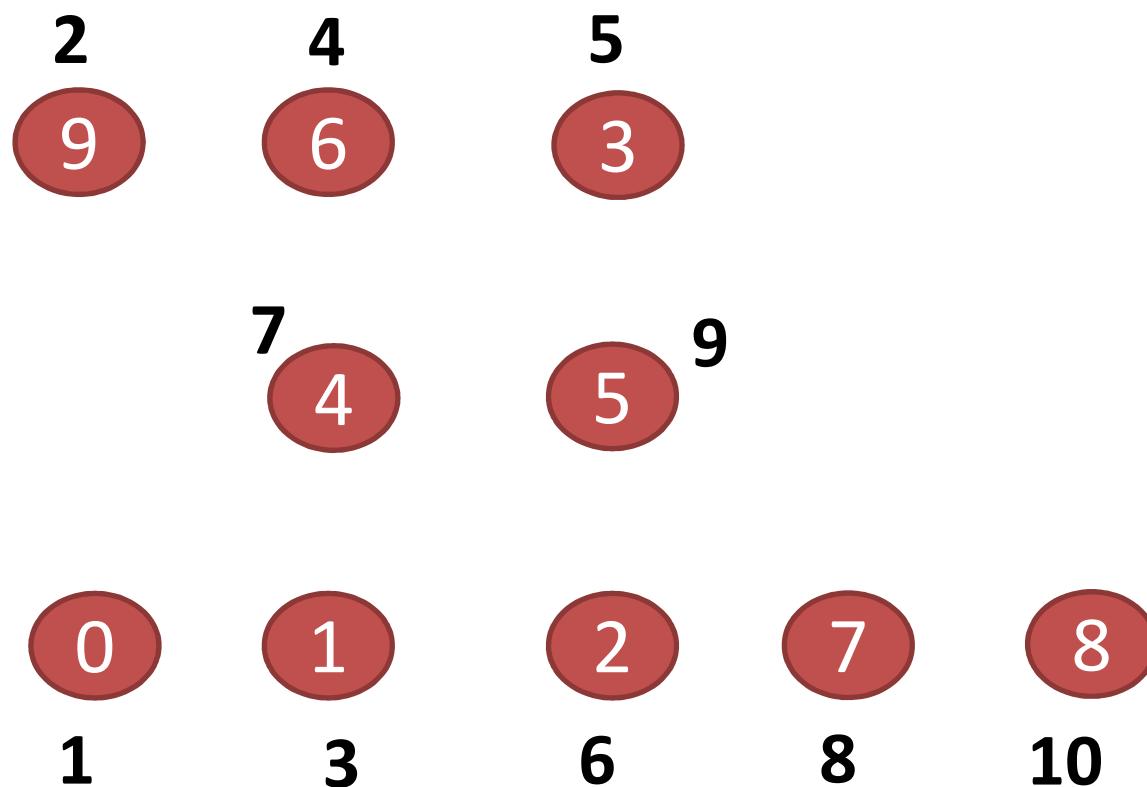


Sắp xếp topo: Ví dụ



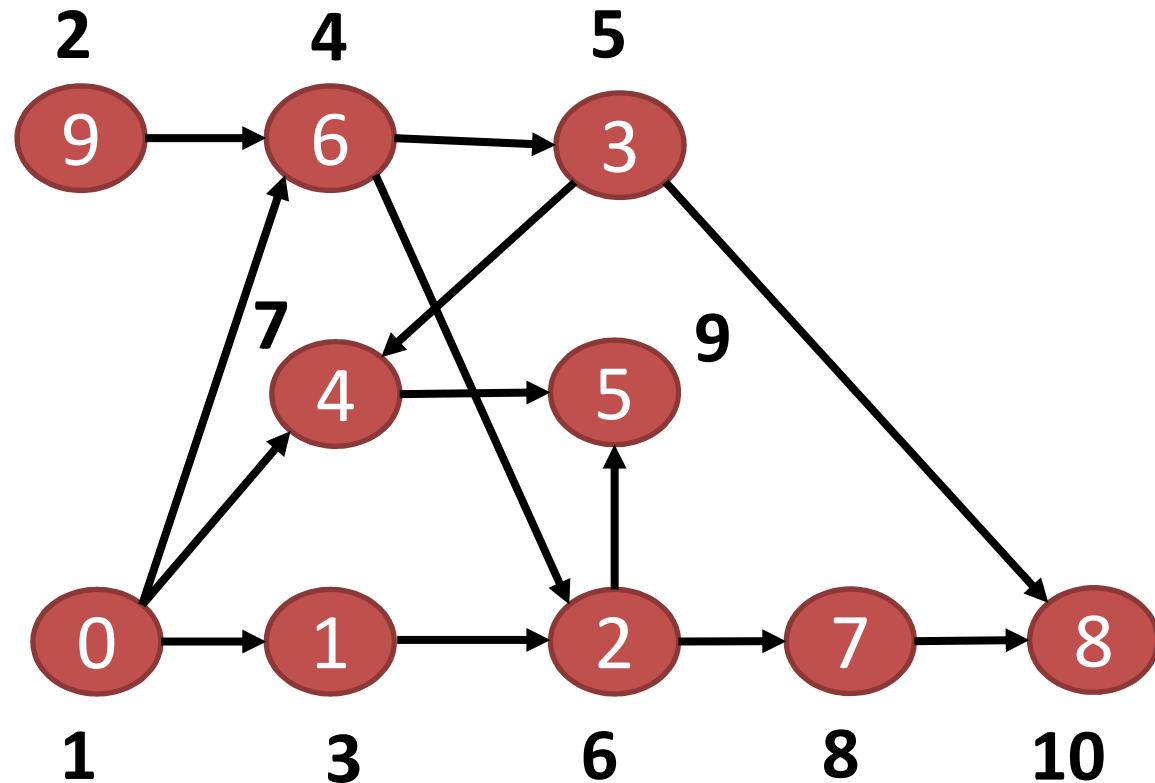
Output: 0 9 1 6 3 2 4 7 5

Sắp xếp topo: Ví dụ



Output: 0 9 1 6 3 2 4 7 5 8

Sắp xếp topo: Ví dụ



Output: 0 9 1 6 3 2 4 7 5 8

Sắp xếp tôtô: dùng thuật toán xoá dần đỉnh

```
void Graph::XoaDanDinh_topo()
{
    //Tạo vector chứa bán bắc vào của tất cả các đỉnh trên đồ thị: khởi tạo =0
    vector<int> in_degree(V, 0);
    for (int v=0; v<V; v++)
    {
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); i++) in_degree[*i]++;
    }

    //Hàng đợi q chứa tất cả các đỉnh có indegree=0
    queue<int> q;
    for (int v = 0; v < V; v++)
        if (in_degree[v] == 0) q.push(v);

    int num = 0;

    //Vecto chứa kết quả thu tu topo:
    vector <int> top_order;

    while (!q.empty())
    {
        int v = q.front();
        q.pop();
        num++;
        top_order.push_back(v);

        list<int>::iterator it;
        for (it = adj[v].begin(); it != adj[v].end(); it++)
        {
            int u = *it;
            in_degree[u]--;
            if (in_degree[u] == 0) q.push(u);
        }
    }
}
```

for $v \in V$ do

Tính Indegree[v] – bán bắc vào của đỉnh v ;
 Q = hàng đợi chứa tất cả các đỉnh có bán bắc vào = 0;
 $num=0$;

while $Q \neq \emptyset$ do

$v = \text{dequeue}(Q)$; $num=num+1$;

Đánh số đỉnh v bởi num ;

for $u \in \text{Adj}(v)$ do

Degree[u]=Degree[u] -1;

if Degree[u]==0

Enqueue(Q, u);

if ($num \neq V$)

{ cout << "Đo thị có chu trình\n"; return; }

//In thu tu topo

for (int i=0; i < top_order.size(); i++) cout << top_order[i] << " "; cout << endl;

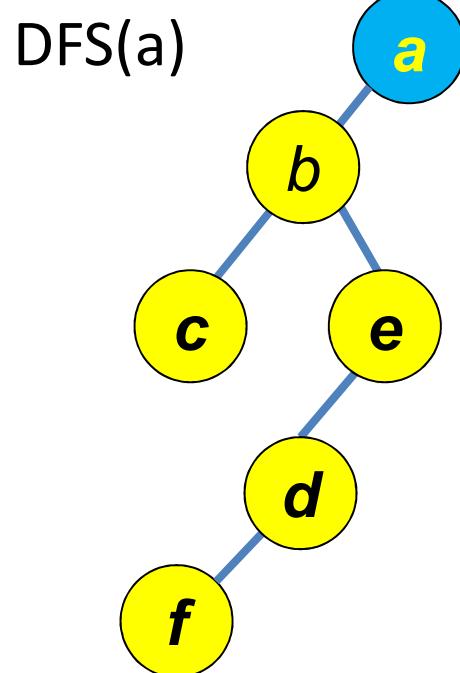
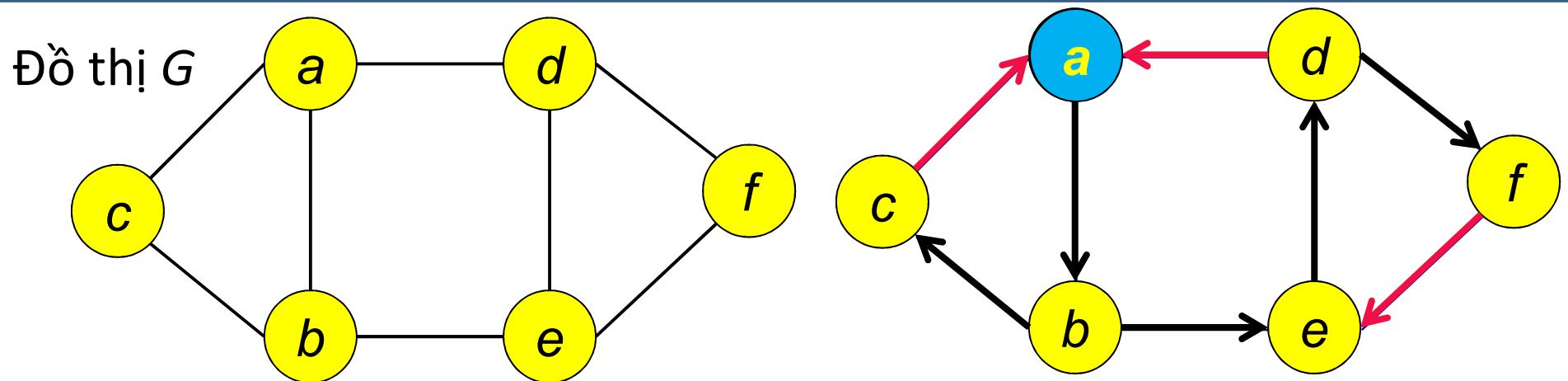
Các ứng dụng của BFS/DFS

1. Sắp xếp topo
- 2. Định hướng đồ thị**
3. Đường đi từ s đến t
4. Đồ thị hai phía
5. Tìm đường đi dài nhất trên cây
6. Bài toán tìm bao đóng truyền ứng
7. Phát hiện chu trình
8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông
9. Tính liên thông của đồ thị
10. Kiểm tra tính liên thông mạnh

Định hướng đồ thị

- **Bài toán:** Cho đồ thị vô hướng liên thông $G = (V, E)$. Hãy tìm cách định hướng các cạnh của nó để thu được đồ thị có hướng liên thông mạnh hoặc trả lời G là không định hướng được.
- **Thuật toán định hướng δ :** Trong quá trình thực hiện DFS(G) định hướng: (1) các cạnh của cây DFS theo chiều từ tổ tiên đến con cháu, (2) các cạnh ngược theo hướng từ con cháu đến tổ tiên. Ký hiệu đồ thị thu được là $G(\delta)$
- **Bổ đề.** G là định hướng được khi và chỉ khi $G(\delta)$ là liên thông mạnh.

Ví dụ: Định hướng đồ thị



Thuật toán định hướng δ : Trong quá trình thực hiện DFS(G) định hướng: (1) các cạnh của cây DFS theo chiều từ tổ tiên đến con cháu, (2) các cạnh ngược theo hướng từ con cháu đến tổ tiên. Ký hiệu đồ thị thu được là $G(\delta)$

Các ứng dụng của BFS/DFS

1. Sắp xếp topo
2. Định hướng đồ thị
- 3. Đường đi từ s đến t**
4. Đồ thị hai phía
5. Tìm đường đi dài nhất trên cây
6. Bài toán tìm bao đóng truyền ứng
7. Phát hiện chu trình
8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông
9. Tính liên thông của đồ thị
10. Kiểm tra tính liên thông mạnh

Tìm đường đi

Bài toán tìm đường đi

- **Input:** Đồ thị $G = (V, E)$ xác định bởi danh sách kè và hai đỉnh s, t .
- **Output:** Đường đi từ đỉnh s đến đỉnh t , hoặc khẳng định không tồn tại đường đi từ s đến t .

Thuật toán: Thực hiện DFS(s) (hoặc BFS(s)).

- Nếu $\text{truoc}[t] == \text{NULL}$ thì không có đường đi, trái lại ta có đường đi
$$t \leftarrow \text{truoc}[t] \leftarrow \text{truoc}[\text{truoc}[t]] \leftarrow \dots \leftarrow s$$

Chú ý: BFS tìm được đường đi ngắn nhất theo số cạnh.

DFS giải bài toán đường đi

(* Main Program*)

```
1. for  $u \in V$  {  
2.     visited[ $u$ ]  $\leftarrow$  false  
3.     truoc[ $u$ ]  $\leftarrow$  NULL }  
4. DFS( $s$ )
```

DFS(s)

```
1.     visited[ $s$ ]  $\leftarrow$  true //Thăm đỉnh  $s$   
2.     for each  $v \in Adj[s]$   
3.         if ( $visited[v] ==$  false) {  
4.             truoc[ $v$ ]  $\leftarrow s$   
5.             DFS( $v$ )  
6.         }
```

BFS giải bài toán đường đi

(* Main Program*)

```
1. for  $u \in V$  {  
2.   visited[ $u$ ]  $\leftarrow$  false  
3.   truoc[ $u$ ]  $\leftarrow$  NULL }  
4. BFS( $s$ )
```

BFS(s)

```
1.   visited[ $s$ ]  $\leftarrow$  true; //Thăm đỉnh  $s$   
2.   truoc[ $s$ ]  $\leftarrow$  null;  
3.   Q  $\leftarrow$   $\emptyset$ ; enqueue(Q, $s$ ); // Nạp  $s$  vào Q  
4.   while ( $Q \neq \emptyset$ )  
5.   {  
6.      $u \leftarrow$  dequeue(Q); // Lấy  $u$  khỏi Q  
7.     for  $v \in \text{Adj}[u]$   
8.       if (visited[ $v$ ] == false) //chưa thăm  
9.       {  
10.          visited[ $v$ ]  $\leftarrow$  true; //đã thăm  
11.          truoc[ $v$ ]  $\leftarrow u$ ;  
12.          enqueue(Q, $v$ ) //Nạp  $v$  vào Q  
13.       }  
14.   }
```

Các ứng dụng của BFS/DFS

1. Sắp xếp topo
2. Định hướng đồ thị
3. Đường đi từ s đến t

4. Đồ thị hai phía

5. Tìm đường đi dài nhất trên cây
6. Bài toán tìm bao đóng truyền ứng
7. Phát hiện chu trình
8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông
9. Tính liên thông của đồ thị
10. Kiểm tra tính liên thông mạnh

4. Đồ thị hai phía

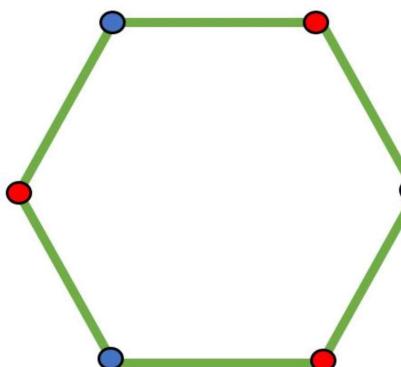
Cho đồ thị vô hướng $G = (V, E)$. Hỏi đồ thị G có phải là đồ thị hai phía hay không?

Trả lời: Tiến hành tô màu mỗi đỉnh của đồ thị bởi 1 trong hai màu: đen hoặc đỏ.

Thực hiện thuật toán BFS trên đồ thị G để tô màu các đỉnh trên đồ thị.

- Đỉnh nguồn: tô màu đen “1” (cho vào tập đỉnh V_1)
- Tất cả các đỉnh kề với nó: tô màu đỏ “0” (cho vào tập đỉnh V_2)
- Tổng quát:
 - Những đỉnh đến được từ đỉnh nguồn bởi số cạnh là lẻ: tô màu đỏ “0”
 - Những đỉnh đến được từ đỉnh nguồn bởi số cạnh là chẵn: tô màu đen “1”.

Trong quá trình thực hiện tô màu, nếu ta phát hiện ra hai đỉnh kề nhau có cùng màu
➔ đồ thị đã cho không phải là hai phía



4. Đồ thị hai phía

```
//return true neu do thi la do thi hai phia
bool isBipartite(int A[][]V], int src)
{
    int colorArr[V];
    //Khoi tao cac dinh deu chua duoc to mau:
    for (int i = 0; i < V; ++i) colorArr[i] = -1;
    colorArr[src] = 1; //To mau den "1" cho dinh nguon

    queue <int> q;
    q.push(src);

    while (!q.empty())
    {
        //Lay u khỏi q
        int u = q.front();
        q.pop();

        //Return false vi do thi co canh khuyen
        if (A[u][u] == 1) return false;

        //Duyet qua tat ca cac dinh v ke voi dinh u:
        for (int v = 0; v < V; ++v)
        {
            if (A[u][v] && colorArr[v] == -1) //v ke voi u va v chua duoc to mau
            {
                // To mau v nguoc voi mau cua u
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }
            // v ke voi u, nhung v co cung mau voi u:
            else if (A[u][v] && colorArr[v] == colorArr[u]) return false;
        }
    }

    //Den day khi: tat ca cac dinh ke nhau da duoc to mau khac nhau
    return true;
}
```

```
void BFS(s) {
    // Tìm kiếm theo chiều rộng bắt đầu từ đỉnh s
    visited[s] ← 1; //đã thăm
    Q ← ∅; enqueue(Q,s); // Nạp s vào Q
    while (Q ≠ ∅)
    {
        u ← dequeue(Q); // Lấy u khỏi Q
        for v ∈ Adj[u]
            if (visited[v] == 0) //chưa thăm
            {
                visited[v] ← 1; //đã thăm
                enqueue(Q,v) // Nạp v vào Q
            }
    }
}
```

4. Đồ thị hai phía

```
int main()
{
    //cho do thi boi ma tran ke
    int A[V][V] = {{0, 1, 0, 1}, {1, 0, 1, 0}, {0, 1, 0, 1}, {1, 0, 1, 0} };

    cout<<"Do thi da cho la hai phia ?: ";
    isBipartite(A, 0) ? cout << "Yes" : cout << "No";
    return 0;
}
```

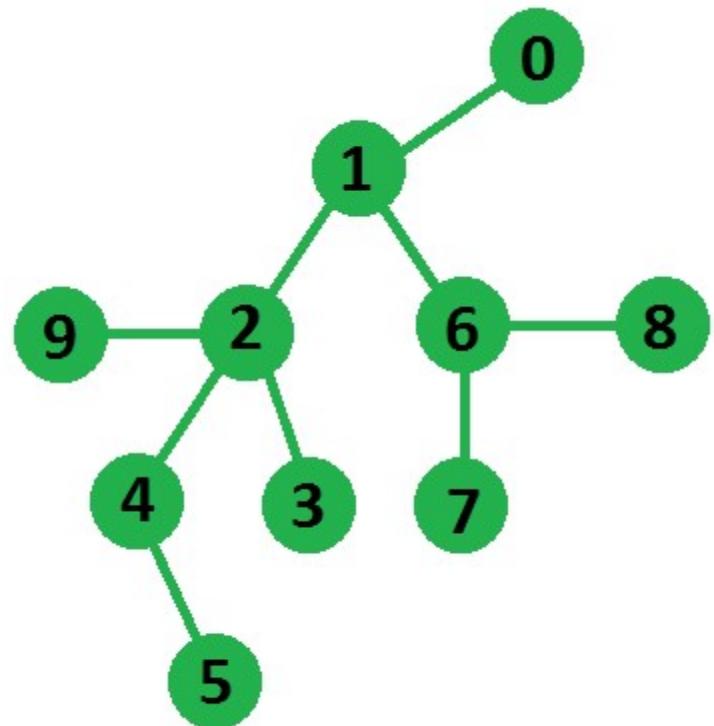
Các ứng dụng của BFS/DFS

1. Sắp xếp topo
2. Định hướng đồ thị
3. Đường đi từ s đến t
4. Đồ thị hai phía
- 5. Tìm đường đi dài nhất trên cây**
6. Bài toán tìm bao đóng truyền ứng
7. Phát hiện chu trình
8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông
9. Tính liên thông của đồ thị
10. Kiểm tra tính liên thông mạnh

5. Tìm đường đi dài nhất trên cây

Yêu cầu: Cho cây $T = (V, E)$. Tìm đường đi dài nhất trên cây

Ví dụ: Đường đi dài nhất có độ dài = 5 trên
cây là $8 - 6 - 1 - 2 - 4 - 5$



Bổ đề: Thực hiện thuật toán BFS (DFS) từ
một đỉnh x bất kỳ của cây, tìm đỉnh mà độ dài của đường đi từ x đến nó là
dài nhất. Kí hiệu đó là u . Khi đó u là đỉnh đầu/cuối của đường đi có độ dài
dài nhất trên cây đã cho.

5. Tìm đường đi dài nhất trên cây

Yêu cầu: Cho cây $T = (V, E)$. Tìm đường đi dài nhất trên cây

Thuật toán: thực hiện 2 lần BFS (DFS)

- Thực hiện thuật toán BFS(DFS) từ một đỉnh bất kỳ x của cây, tìm đỉnh mà độ dài của đường đi từ x đến nó là dài nhất. Kí hiệu đỉnh đó là u .
- Thực hiện BFS (DFS) từ đỉnh u , tìm đỉnh mà độ dài của đường đi từ u đến nó là dài nhất. Kí hiệu nó là v .

Đường đi từ u đến v là đường đi có độ dài lớn nhất trên cây đã cho.

5. Tìm đường đi dài nhất trên cây

```
#include <bits/stdc++.h>
using namespace std;

class Graph
{
    int V;                      // No. of vertices
    list<int> *adj;             // Pointer to an array containing adjacency lists
public:
    Graph(int V);               // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void longestPathLength();   // prints longest path of the tree
    pair<int, int> bfs(int u); /* ham tra ve cap 2 so (v, maxDis)
    voi v la dinh cho duong di tu u den v co do dai dai nhat
    trong tat ca cac duong di tu u den cac dinh khac */
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}
```

```
// method returns farthest node and its distance from node u
pair<int, int> Graph::bfs(int u)
{
    //Khoi tao:
    int dis[V];
    memset(dis, -1, sizeof(dis)); // mark all distance with -1

    queue<int> q;
    q.push(u);
    dis[u] = 0; // distance of u from u will be 0
    while (!q.empty())
    {
        int t = q.front();
        q.pop();
        list<int>::iterator it;
        // Duyet qua tat ca cac dinh ke voi t:
        for (it = adj[t].begin(); it != adj[t].end(); it++)
        {
            int v = *it;
            if (dis[v] == -1) //chua tham v
            {
                q.push(v); //day v vao queue
                // make distance of v, one more than distance of t
                dis[v] = dis[t] + 1;
            }
        }
    }
    int maxDis = 0;
    int nodeIdx;
    // get farthest node distance and its index
    for (int i = 0; i < V; i++)
        if (dis[i] > maxDis)
    {
        maxDis = dis[i];
        nodeIdx = i;
    }
    return make_pair(nodeIdx, maxDis);
}
```

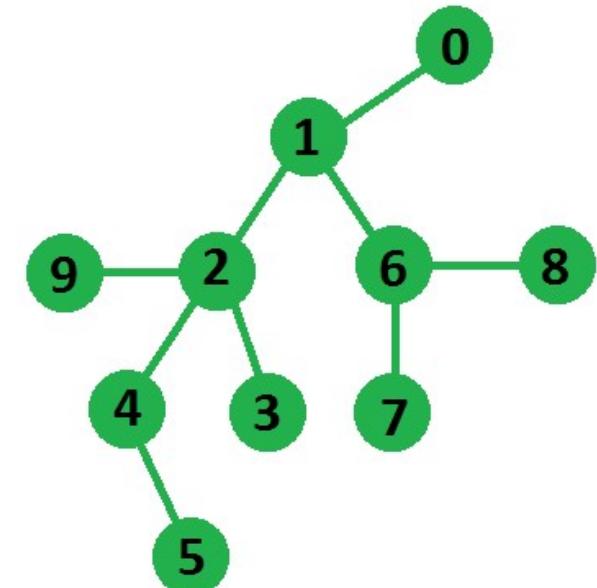
```

// method prints longest path of given tree
void Graph::longestPathLength()
{
    pair<int, int> t1, t2;
    // first bfs to find one end point of longest path
    t1 = bfs(0);
    // second bfs to find actual longest path
    t2 = bfs(t1.first);
    cout << "Longest path is from " << t1.first << " to "
        << t2.first << " of length " << t2.second;
}

int main()
{
    // Tao cay co 10 dinh va 9 canh:
    Graph g(10);
    g.addEdge(0, 1); g.addEdge(1, 2);
    g.addEdge(2, 3); g.addEdge(2, 9);
    g.addEdge(2, 4); g.addEdge(4, 5);
    g.addEdge(1, 6); g.addEdge(6, 7);
    g.addEdge(6, 8);

    g.longestPathLength();
    return 0;
}

```



Các ứng dụng của BFS/DFS

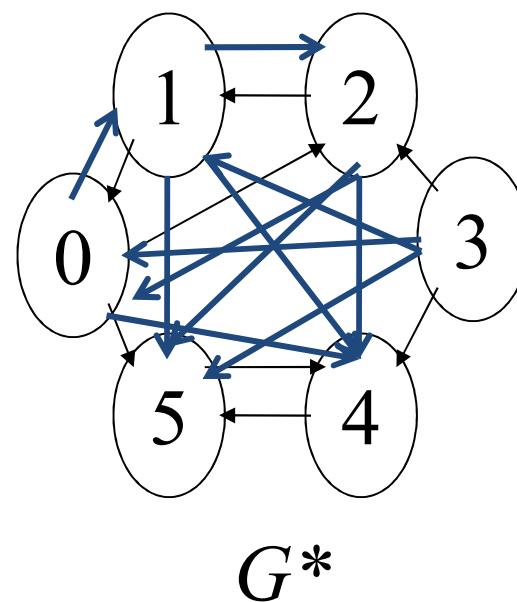
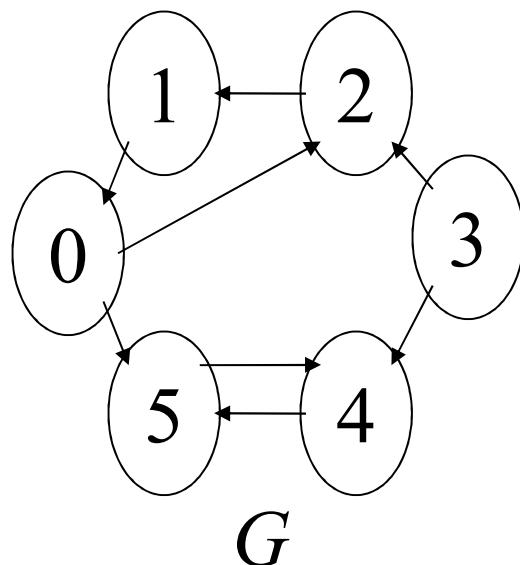
1. Sắp xếp topo
2. Định hướng đồ thị
3. Đường đi từ s đến t
4. Đồ thị hai phía
5. Tìm đường đi dài nhất trên cây
- 6. Bài toán tìm bao đóng truyền ứng**
7. Phát hiện chu trình
8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông
9. Tính liên thông của đồ thị
10. Kiểm tra tính liên thông mạnh

6. Transitive Closure

Định nghĩa. Bao đóng truyền ứng của đồ thị có hướng $G=(V,E)$ là đồ thị có hướng $G^*=(V,E^*)$ với tập đỉnh là tập đỉnh của đồ thị G và tập cạnh

$$E^* = \{(u,v) \mid \text{có đường đi từ } u \text{ đến } v \text{ trên } G\}$$

Bài toán: Cho đồ thị có hướng G , tìm bao đóng truyền ứng G^*



Nhân ma trận Bun

Boolean Matrix multiplication

- Ma trận Bun – là ma trận có tất cả phần tử là 0 hoặc 1
- Để tính tích của hai ma trận Bun A và B ta thay thế
 - phép toán logic *AND* vào chỗ phép toán số học *
 - phép toán logic *OR* vào chỗ phép toán số học +
- Cho 2 ma trận kích thước $|V|.|V|$
 - Với mỗi s và t : tính tích vô hướng của dòng s của ma trận thứ nhất với cột t của ma trận thứ hai.

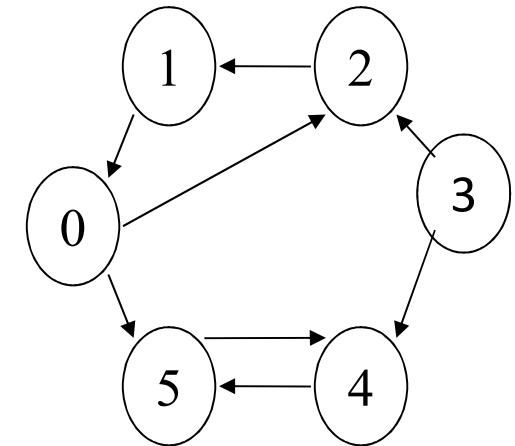
Matrix Multiplication Implementation

- $C = A * B$ (sô)

```
for (s = 0; s < V; s++)
    for (t = 0; t < V; t++)
        for (i = 0, C[s][t] = 0; i < V; i++)
            C[s][t] += A[s][i] * B[i][t];
```

- $C = A * B$ (bun)

```
for (s = 0; s < V; s++)
    for (t = 0; t < V; t++)
        for (i = 0, C[s][t] = 0; i < V; i++)
            if (A[s][i] && B[i][t])
                C[s][t] = 1
```



to t

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | | 1 | | | 1 |
| 1 | 1 | 1 | | | | |
| 2 | | 1 | 1 | | | |
| 3 | | | 1 | 1 | 1 | |
| 4 | | | | | 1 | 1 |
| 5 | | | | | 1 | 1 |

from s

Sử dụng tích ma trận để tìm bao đóng truyền ứng

- Ta có thể tính bao đóng truyền ứng cho đồ thị có hướng bằng cách xây dựng ma trận kề A cho nó, bổ sung các số 1 vào đường chéo rồi tính luỹ thừa $A^{|V|}$

Chứng minh:

- A^2 – với mỗi cặp s và t , ta ghi 1 vào ma trận tích C khi và chỉ khi có đường đi từ s đến i và đường đi từ i đến t trong A
 - Xét đến các đường đi độ dài 2
- A^3 – xét đến các đường đi độ dài 3
- Không cần xét đến các đường đi độ dài lớn hơn $|V|$ bởi vì đường đi đơn trên đồ thị có độ dài lớn nhất là $|V|$

Thời gian tính

- Question: Thuật toán tìm bao đóng truyền ứng vừa mô tả đòi hỏi thời gian bao nhiêu ?
- Answer: $|V|^4$
- Question: Liệu có thể tính A^2 tiếp đến $A^4 \dots A^t$ với $t \geq |V|$ để cải thiện thời gian tính?
- Ans: $|V|^3 \log |V|$

Thuật toán Warshall

```
// n = |V|, Các đỉnh đánh số từ 0 đến n-1
for (i = 0; i < n; i++)
    for (s = 0; s < n; s++)
        for (t=0; t < n; t++)
            if (A[s][i] && A[i][t])
                A[s][t] = 1;
```

- Time: $O(|V|^3)$
- Space: $O(|V|^2)$

Mệnh đề. Thuật toán tìm được bao đóng truyền ứng với thời gian $O(|V|^3)$

- **Chứng minh:** Ta chứng minh thuật toán tìm được bao đóng truyền ứng bằng qui nạp.
 - Lần lặp 1: Ma trận có 1 ở vị trí (s,t) iff có đường đi $s-t$ hoặc $s-0-t$
 - Lần lặp thứ i : Gán phần tử ở vị trí (s,t) giá trị 1 iff có đường đi từ s đến t trong đồ thị không chứa đỉnh với chỉ số lớn hơn i (ngoại trừ hai mút)
 - Lần lặp thứ $i+1$
 - Nếu có đường đi từ s đến t không chứa đỉnh có chỉ số lớn hơn i – $A[s,t]$ đã có giá trị 1
 - Nếu có đường đi từ s đến $i+1$ và đường đi từ $i+1$ đến t , và cả hai đều không chứa đỉnh với chỉ số lớn hơn i (ngoại trừ hai mút) thì $A[s,t]$ được gán giá trị 1

Thuật toán Warshall cải tiến

Ta có thể cải tiến thuật toán bằng cách dịch chuyển câu lệnh if $A[s][i]$ lên trước vòng lặp for trong cùng:

```
// n = |V|, Các đỉnh đánh số từ 0 đến n-1
for (i = 0; i < n; i++)
    for (s = 0; s < n; s++)
        if A[s][i]
            for (t=0; t < n; t++)
                if A[i][t]
                    A[s][t] = 1;
```

Cải tiến này chỉ có tác dụng tăng hiệu quả thực tế của thuật toán, mà không thay đổi được đánh giá thời gian tính trong tình huống tồi nhất của thuật toán

Áp dụng DFS tìm bao đóng truyền ứng

Mệnh đề. Sử dụng DFS ta có thể xác định bao đóng truyền ứng sau thời gian $O(|V|^*(|E|+|V|))$

- **Chứng minh**

- DFS cho phép xác định tất cả các đỉnh đạt đến được từ một đỉnh cho trước v sau thời gian $O(|E|+|V|)$ nếu ta sử dụng biểu diễn đồ thị bởi danh sách kè
- Do đó để xác định bao đóng truyền ứng ta thực hiện DFS với mỗi $v \in V \rightarrow$ tổng cộng thực hiện DFS $|V|$ lần.
 \rightarrow Thời gian tính: $O(|V|^*(|E|+|V|))$.

Kinh nghiệm tính toán

| đồ thị thưa ($ E =10 V $) | | | | | đồ thị dày (250 đỉnh) | | | | |
|-----------------------------|------|------|------|----|-----------------------|-----|-----|-----|-----|
| V | W | W* | A | L | E | W | W* | A | L |
| 25 | 0 | 0 | 1 | 0 | 5000 | 289 | 203 | 177 | 23 |
| 50 | 3 | 1 | 2 | 1 | 10000 | 300 | 214 | 184 | 38 |
| 125 | 35 | 24 | 23 | 4 | 25000 | 309 | 226 | 200 | 97 |
| 250 | 275 | 181 | 178 | 13 | 50000 | 315 | 232 | 218 | 337 |
| 500 | 2222 | 1438 | 1481 | 54 | 100000 | 326 | 246 | 235 | 784 |

W Thuật toán Warshall

W* Thuật toán Warshall cải tiến

A DFS sử dụng ma trận kề

L DFS sử dụng danh sách kề

Các ứng dụng của BFS/DFS

1. Sắp xếp topo
2. Định hướng đồ thị
3. Đường đi từ s đến t
4. Đồ thị hai phía
5. Tìm đường đi dài nhất trên cây
6. Bài toán tìm bao đóng truyền ứng
- 7. Phát hiện chu trình**
8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông
9. Tính liên thông của đồ thị
10. Kiểm tra tính liên thông mạnh

7. Bài toán chu trình

Bài toán: Cho đồ thị $G=(V,E)$. Hỏi G có chứa chu trình hay không?

- **Cách 1: Sử dụng DFS**
- **Cách 2: Sử dụng BFS**

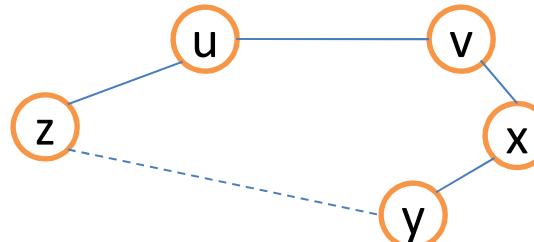
7. Bài toán chu trình: sử dụng DFS

Bài toán: Cho đồ thị $G=(V,E)$. Hỏi G có chứa chu trình hay không?

- **Định lý:** *Đồ thị G là không chứa chu trình khi và chỉ khi trong quá trình thực hiện DFS ta không phát hiện ra cạnh ngược.*

Chứng minh:

- \Rightarrow) Nếu G không chứa chu trình thì không thể có cạnh ngược. Hiển nhiên: bởi vì sự tồn tại cạnh ngược kéo theo sự tồn tại chu trình.
- \Leftarrow) Ta phải chứng minh: Nếu không có cạnh ngược thì G không chứa chu trình. Ta chứng minh bằng lập luận phản đòn: G có chu trình $\Rightarrow \exists$ cạnh ngược. Gọi v là đỉnh trên chu trình được thăm đầu tiên trong quá trình thực hiện DFS, và u là đỉnh đi trước v trên chu trình. Khi v được thăm, các đỉnh khác trên chu trình đều chưa được thăm. Ta phải thăm được tất cả các đỉnh đạt được từ v trước khi quay trở lại từ DFS(v). Vì thế cạnh $u \rightarrow v$ được duyệt từ đỉnh u về tổ tiên v của nó, vì thế (u, v) là cạnh ngược.



Như vậy DFS có thể áp dụng để giải bài toán đặt ra.

7. Bài toán chu trình: sử dụng DFS

Bài toán: Cho đồ thị $G=(V,E)$. Hỏi G có chứa chu trình hay không?

- **Định lý:** *Đồ thị G là không chứa chu trình khi và chỉ khi trong quá trình thực hiện DFS ta không phát hiện ra cạnh ngược.*

– **Cách phát hiện ra có cạnh ngược hay không?**

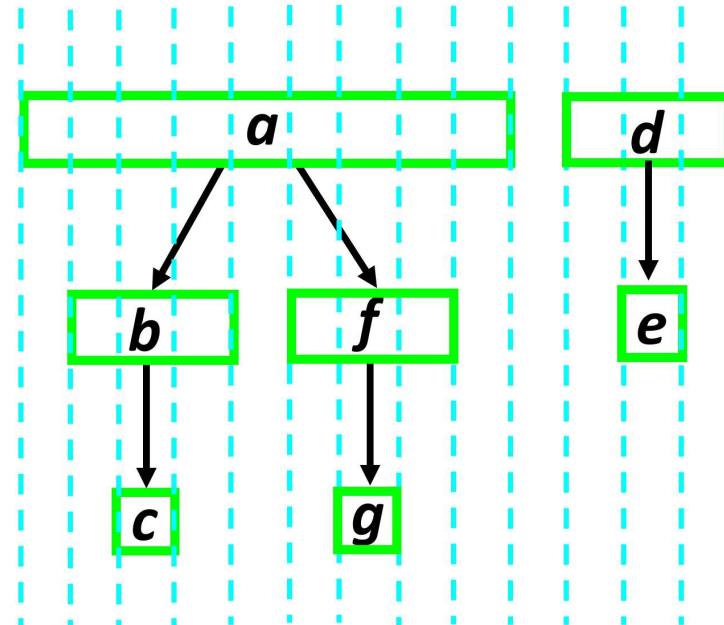
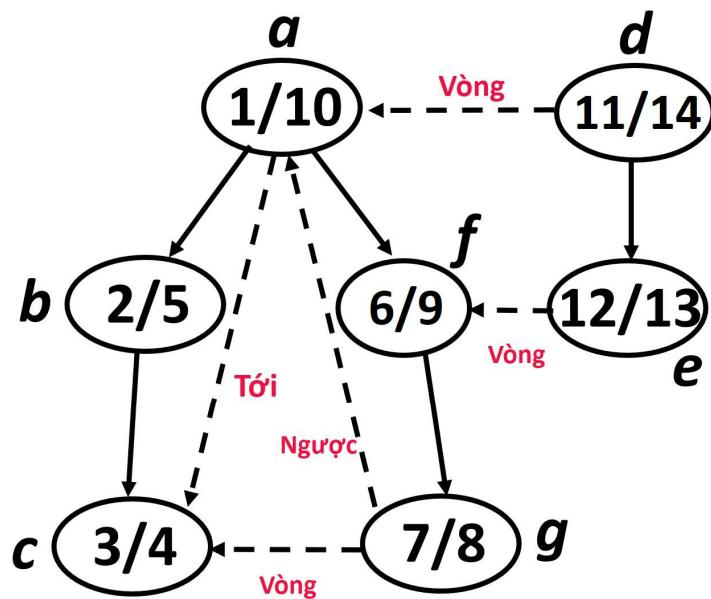
- Cách 1: Sử dụng bờ đề về các khoảng lồng nhau
- Cách 2: Đánh trạng thái cho các đỉnh

DFS: Bỏ đề về các khoảng lồng nhau (Cách 1)

Cho đồ thị có hướng $G = (V, E)$, và cây DFS bất kỳ của G và hai đỉnh u, v tùy ý của nó. Khi đó

- u là con cháu của v khi và chỉ khi $[d[u], f[u]] \subseteq [d[v], f[v]]$
- u là tổ tiên của v khi và chỉ khi $[d[u], f[u]] \supseteq [d[v], f[v]]$
- u và v không có quan hệ họ hàng khi và chỉ khi $[d[u], f[u]]$ và $[d[v], f[v]]$ là không giao nhau.

- *Tree edge (cạnh cây)*: cạnh theo đó từ một đỉnh đến thăm đỉnh mới
- *Back edge (cạnh ngược)*: đi từ con cháu đến tổ tiên
- *Forward edge (cạnh tới)*: đi từ tổ tiên đến con cháu
- *Cross edge (cạnh vòng)*: giữa hai đỉnh không có họ hàng



DFS và chu trình: Cách 2

- *Cần phải điều chỉnh như thế nào để phát hiện chu trình?*

(* Main Program*)

1. **for** $u \in V$
2. $\text{visited}[u] \leftarrow \text{false}$
3. **for** $u \in V$
4. **if** ($\text{visted}[u] == \text{false}$)
5. $\text{DFS}(u)$

DFS(u)

1. $\text{visited}[u] \leftarrow \text{true}$ //Thăm đỉnh u
2. **for each** $v \in \text{Adj}[u]$
3. **if** ($\text{visited}[v] == \text{false}$)
4. $\text{DFS}(v)$

Hiện mỗi đỉnh có 2 trạng thái: $\text{visited} = \text{false}$ hoặc true

DFS và chu trình: Cách 2

- Thay vì chỉ sử dụng **hai** trạng thái: chưa thăm/đã thăm (`visited = 0 / 1`) cho mỗi đỉnh của đồ thị → Cần sửa lại: mỗi đỉnh ở một **trong ba** trạng thái:
 - Chưa thăm: `visited = 0`
 - Đã thăm (nhưng chưa duyệt xong): `visited = 1`
 - Đã duyệt xong: `visited = 2`
- Trạng thái của đỉnh sẽ biến đổi theo qui tắc sau:
 - Khởi tạo: mỗi đỉnh v đều chưa thăm `visited[v] = 0`
 - Thăm đỉnh v: `visited[v] = 1` (trở thành đã thăm nhưng chưa duyệt xong).
 - Khi tất cả các đỉnh kề của một đỉnh v là đã được thăm, đỉnh v sẽ gọi là đã duyệt xong `visited[v] = 2`

DFS: Các loại cạnh

DFS tạo ra một cách phân loại các cạnh của đồ thị đã cho:

- Khi ta duyệt cạnh $e = (u, v)$ từ đỉnh u , căn cứ vào $\text{visited}[v]$, ta có thể biết cạnh này thuộc loại cạnh nào:
 - 1) $\text{visited}[v] = 0$: cho biết e là cạnh của cây
 - 2) $\text{visited}[v] = 1$: cho biết e là cạnh ngược
 - 3) $\text{visited}[v] = 2$: cho biết e là cạnh tói hoặc vòng
 - *Tree edge (cạnh cây)*: cạnh theo đó từ một đỉnh đến đỉnh mới
 - *Back edge (cạnh ngược)*: đi từ con cháu đến tổ tiên
 - *Forward edge (cạnh tói)*: đi từ tổ tiên đến con cháu
 - *Cross edge (cạnh vòng)*: giữa hai đỉnh không có họ hàng

DFS và chu trình: Cách 2

- Cần phải điều chỉnh như thế nào để phát hiện chu trình?

(* Main Program*)

1. **for** $u \in V$
2. $\text{visited}[u] \leftarrow \text{false}$
3. **for** $u \in V$
4. **if** ($\text{visted}[u] == \text{false}$)
5. $\text{DFS}(u)$

DFS(u)

1. $\text{visited}[u] \leftarrow \text{true}$ //Thăm đỉnh u
2. **for each** $v \in \text{Adj}[u]$
3. **if** ($\text{visited}[v] == \text{false}$)
4. $\text{DFS}(v)$



DFS(u)

1. $\text{visited}[u] \leftarrow 1$ //Thăm đỉnh u
2. **for each** $v \in \text{Adj}[u]$
3. **if** ($\text{visited}[v] == 1$)
4. {
5. cout<<“Đồ thị có chu trình”;
6. exit();
7. }
8. **else if**($\text{visited}[v]==0$) $\text{DFS}(v)$
9. $\text{visited}[u] \leftarrow 2$ //Đỉnh u đã duyệt xong

(* Main Program*)

1. **for** $u \in V$
2. $\text{visited}[u] \leftarrow 0$
3. **for** $u \in V$
4. **if** ($\text{visted}[u] == 0$)
5. $\text{DFS}(u)$

DFS và chu trình

- **Câu hỏi:** Thời gian tính là bao nhiêu?

Trả lời: Chính là thời gian thực hiện *DFS*: $O(|V| + |E|)$.

- **Câu hỏi:** Nếu G là đồ thị vô hướng thì có thể đánh giá thời gian tính sát hơn nữa được không?

Trả lời: Thuật toán có thời gian tính $O(|V|)$, bởi vì:

- Trong một rừng (đồ thị không chứa chu trình) $|E| \leq |V| - 1$
- Vì vậy nếu đồ thị có $|V|$ cạnh thì chắc chắn nó chứa chu trình, và thuật toán kết thúc.

Mệnh đề. Đơn đồ thị vô hướng với n đỉnh và n cạnh chắc chắn chứa chu trình.

7. Bài toán chu trình

Bài toán: Cho đồ thị $G=(V,E)$. Hỏi G có chứa chu trình hay không?

- Cách 1: Sử dụng DFS
- **Cách 2: Sử dụng BFS**
 - Đồ thị vô hướng
 - Đồ thị có hướng

BFS và chu trình trên đồ thị vô hướng

- Bài toán:** Cho **đơn** đồ thị vô hướng $G=(V,E)$. Hỏi G có chứa chu trình hay không?

Trả lời: Thực hiện thuật toán BFS trên đồ thị G :

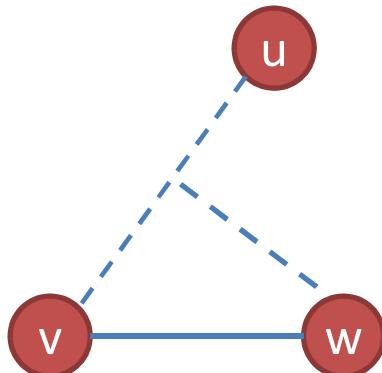
Phát hiện chu trình: trong quá trình thực hiện BFS(u)

Với mỗi đỉnh w , nếu tồn tại đỉnh kề v sao cho:

- v đã thăm ($\text{visited}[v] = 1$)
- $\text{truoc}[w] \neq v$

thì đồ thị G chứa chu trình

→ Sửa lại chương trình như thế nào?



BFS(u)

```
1. visited[u] ← 1 //Thăm đỉnh u
2. truoc[u] ← NULL;
3. Q ← ∅; enqueue(Q,u); // Nạp u vào Q
4. while (Q ≠ ∅)
5. {
6.     w ← dequeue(Q); // Lấy w khỏi Q
7.     for v ∈ Adj[w]
8.         if (visited[v] == 0) //chưa thăm
9.         {
10.             visited[v] ← 1; //đã thăm
11.             truoc[v] ← w;
12.             enqueue(Q,v); //Nạp v vào Q
13.         }
14. }
```

BFS và chu trình trên đồ thị vô hướng

- **Bài toán:** Cho **đơn** đồ thị vô hướng $G=(V,E)$. Hỏi G có chứa chu trình hay không?

Trả lời: Thực hiện thuật toán BFS trên đồ thị G :

(* Main Program*)

```
1. for  $u \in V$  {  
2.   visited[ $u$ ]  $\leftarrow 0$ ;  
3.   truoc[ $u$ ]  $\leftarrow \text{NULL}$ ; }  
4. for  $u \in V$   
5.   if (visited[ $u$ ] == 0)  
6.     BFS( $u$ );
```

BFS(u)

```
1. visited[ $u$ ]  $\leftarrow 1$  //Thăm đỉnh  $u$   
2. truoc[ $u$ ]  $\leftarrow \text{NULL}$ ;  
3.  $Q \leftarrow \emptyset$ ; enqueue( $Q,u$ ); // Nạp  $u$  vào  $Q$   
4. while ( $Q \neq \emptyset$ )  
5. {  
6.    $w \leftarrow \text{dequeue}(Q)$ ; // Lấy  $w$  khỏi  $Q$   
7.   for  $v \in \text{Adj}[w]$   
8.     if (visited[ $v$ ] == 0) //chưa thăm  
9.     {  
10.        visited[ $v$ ]  $\leftarrow 1$ ; //đã thăm  
11.        truoc[ $v$ ]  $\leftarrow w$ ;  
12.        enqueue( $Q,v$ ); //Nạp  $v$  vào  $Q$   
13.     }  
14. }
```

Phát hiện chu trình:

Với mỗi đỉnh w , nếu tồn tại đỉnh kề v sao cho:

- v đã thăm ($\text{visited}[v] = 1$)

- $\text{truoc}[w] \neq v$

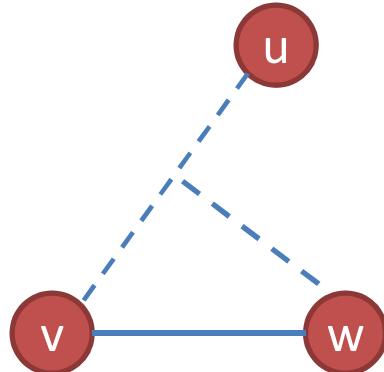
thì đồ thị G chứa chu trình

➔ Sửa lại chương trình như thế nào?

BFS và chu trình trên đồ thị vô hướng

(* Main Program*)

```
1. for  $u \in V$  {  
2.     visited[ $u$ ]  $\leftarrow 0$ ;  
3.     truoc[ $u$ ]  $\leftarrow \text{NULL}$ ; }  
4. Cycle = false;  
5. for  $u \in V$   
6.     if (visited[ $u$ ] == 0)  
7.         { Cycle = BFS( $u$ );  
8.             if (Cycle) {cout<<“YES”; exit();}  
9.             cout<<“NO”;
```



Phát hiện chu trình:

Với mỗi đỉnh w, nếu tồn tại đỉnh kề v sao cho:

- v đã thăm (visited[v] = 1)
 - truoc[w] != v
- thì đồ thị G chứa chu trình

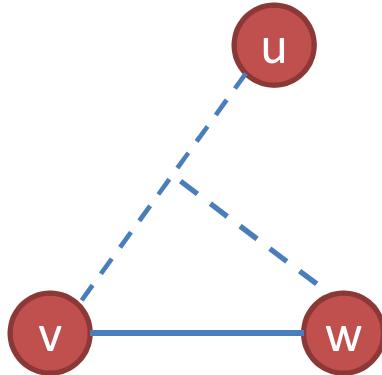
BFS(u)

```
1.     visited[ $u$ ]  $\leftarrow 1$  //Thăm đỉnh  $u$   
2.     truoc[ $u$ ]  $\leftarrow \text{NULL}$ ;  
3.     Q  $\leftarrow \emptyset$ ; enqueue(Q, $u$ ); // Nạp  $u$  vào Q  
4.     while ( $Q \neq \emptyset$ )  
5.     {  
6.         w  $\leftarrow \text{dequeue}(Q)$ ; // Lấy  $w$  khỏi Q  
7.         for  $v \in \text{Adj}[w]$   
8.             if (visited[ $v$ ] == 0) //chưa thăm  
9.             {  
10.                 visited[ $v$ ]  $\leftarrow 1$ ; //đã thăm  
11.                 truoc[ $v$ ]  $\leftarrow w$ ;  
12.                 enqueue(Q, $v$ ); //Nạp  $v$  vào Q  
13.             }  
14.             else if (truoc[w] != v) return true;  
15.         }  
16.     return false;
```

BFS và chu trình trên đồ thị vô hướng

(* Main Program*)

```
1. for  $u \in V$  {  
2.     visited[ $u$ ]  $\leftarrow 0$ ;  
3.     truoc[ $u$ ]  $\leftarrow \text{NULL}$ ; }  
4. Cycle = false;  
5. for  $u \in V$   
6.     if (visited[ $u$ ] == 0)  
7.         { Cycle = BFS( $u$ );  
8.             if (Cycle) {cout<<“YES”; exit();}  
9.             cout<<“NO”;
```



Phát hiện chu trình:

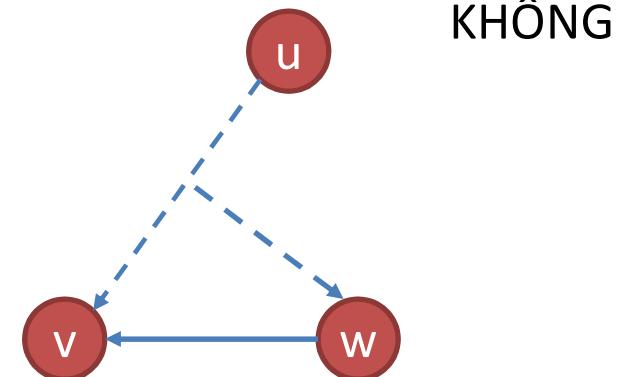
Với mỗi đỉnh w , nếu tồn tại đỉnh kề v sao cho:

- v đã thăm (visited [v] = 1)
- truoc [w] != v

thì đồ thị G chứa chu trình



Có áp dụng cho đồ thị có hướng
được không ?



KHÔNG

7. Bài toán chu trình

Bài toán: Cho đồ thị $G=(V,E)$. Hỏi G có chứa chu trình hay không?

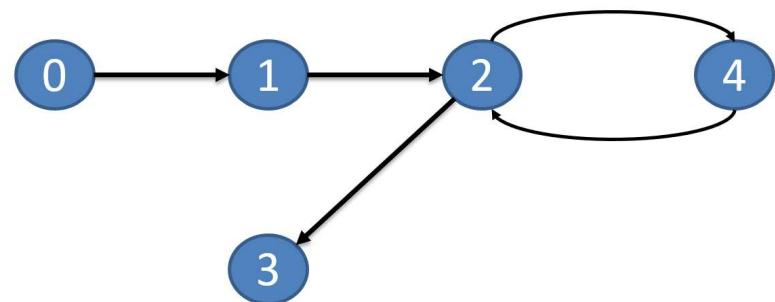
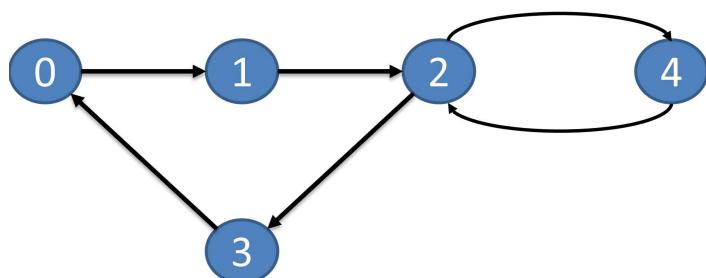
- Cách 1: Sử dụng DFS
- Cách 2: Sử dụng BFS
 - Đồ thị vô hướng
 - **Đồ thị có hướng**

BFS và chu trình trên đồ thị có hướng

Bài toán: Cho đồ thị có hướng $G=(V,E)$. Hỏi G có chứa chu trình hay không?

Trả lời:

- Thực hiện thuật toán xoá dần đỉnh (BFS chỉnh sửa) ở sắp xếp topo.
- Kết thúc thuật toán, nếu số đỉnh được thăm \neq số đỉnh của đồ thị \rightarrow đồ thị có chu trình; ngược lại thì đồ thị không có chu trình.



Không có đỉnh nào có indegree = 0

BFS và chu trình trên đồ thị có hướng

for $v \in V$ **do**

 Tính Indegree[v] – bán bậc vào của đỉnh v ;

Q = hàng đợi chứa tất cả các đỉnh có bán bậc vào = 0;

$num=0$;

while $Q \neq \emptyset$ **do**

{

$v = \text{dequeue}(Q)$; $num=num+1$;

for $u \in \text{Adj}(v)$ **do**

 {

 Degree[u]=Degree[u] -1;

if (Degree[u]==0) Enqueue(Q, u);

 }

}

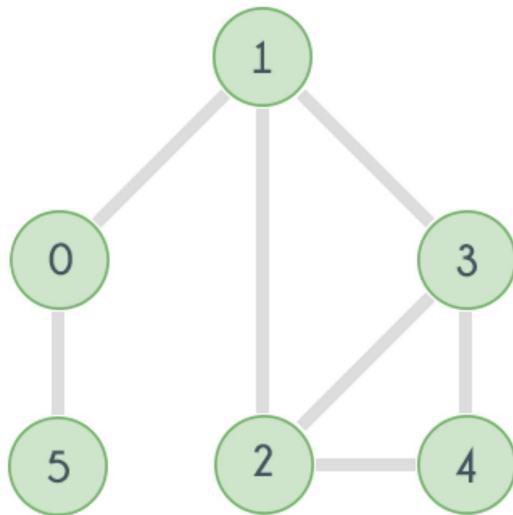
if ($num \neq |V|$) cout<<“YES”;

else cout<<“NO”;

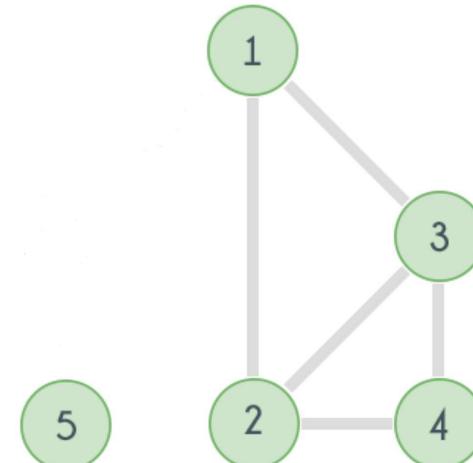
Các ứng dụng của BFS/DFS

1. Sắp xếp topo
2. Định hướng đồ thị
3. Đường đi từ s đến t
4. Đồ thị hai phía
5. Tìm đường đi dài nhất trên cây
6. Bài toán tìm bao đóng truyền ứng
7. Phát hiện chu trình
- 8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông**
9. Tính liên thông của đồ thị
10. Kiểm tra tính liên thông mạnh

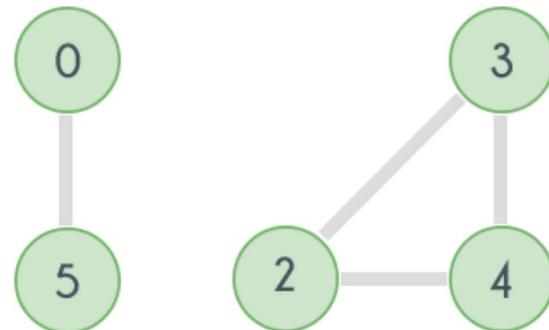
Đỉnh rẽ nhánh



Xóa đỉnh 0



Xóa đỉnh 1



Xóa đỉnh 0 hoặc 1 sẽ
làm tăng số thành phần
liên thông của đồ thị
→ Đỉnh 0 và 1 được gọi
là đỉnh rẽ nhánh

Tìm đỉnh rẽ nhánh

Yêu cầu: Tìm đỉnh rẽ nhánh (cut vertex / articulation point) của **đơn** đồ thị vô hướng $G = (V, E)$ (không nhất thiết phải liên thông).

Trả lời:

- Brute force: thời gian tính $O(V(V + E))$
- DFS sửa đổi: thời gian tính $O(E + V)$

Đỉnh v được gọi là đỉnh rẽ nhánh trên đồ thị G nếu xóa đỉnh v khỏi đồ thị cùng các cạnh nối chúng sẽ làm tăng số thành phần liên thông của đồ thị G

Tìm đỉnh rẽ nhánh: thuật toán brute force

- Brute force: với mỗi đỉnh v của đồ thị
 - Xóa đỉnh v khỏi đồ thị : $O(E)$
 - Kiểm tra tính liên thông của đồ thị bằng BFS/DFS : $O(V+E)$
(nếu đồ thị không liên thông \rightarrow đỉnh v là đỉnh rẽ nhánh)
 - Thêm lại đỉnh v vào đồ thị : $O(E)$

➔ Thời gian tính là $O(V(V + E))$

Tìm đỉnh rẽ nhánh: thuật toán brute force

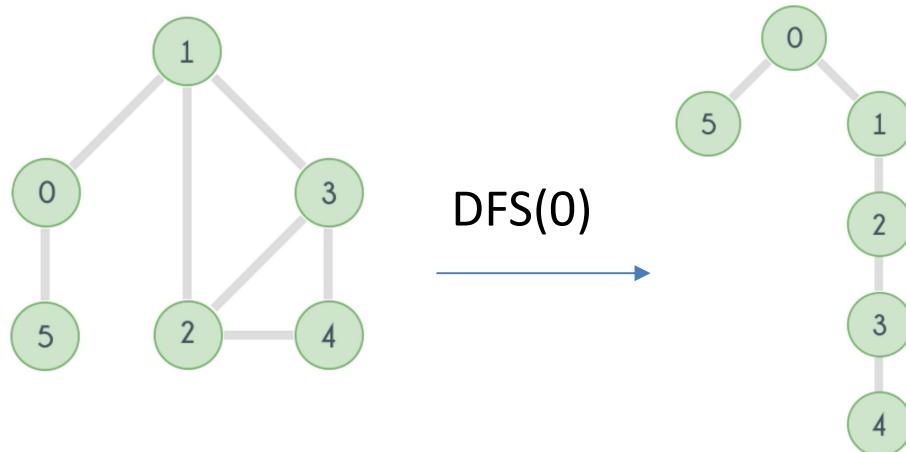
- adj [] []: ma trận kề kích thước VxV ($\text{adj}[i][j]=1$ nếu trên đồ thị có cạnh (i, j) ; trái lại = 0)
- count: trả về số đỉnh rẽ nhánh trên đồ thị
- cutVertex[i] = true nếu đỉnh i là đỉnh rẽ nhánh ($i \in V$)

```
function find_cutVertex(adj[][], V)
{
    //Khoi tao:
    count = 0; //dem so dinh re nhanh cua do thi G
    for (each v in V)
    {
        visited[v] = false;
        cutVertex[v] = false; // cutVertex[v]= true neu dinh v la dinh re nhanh cua do thi G
    }
    //Dem so thanh phan Lien thong (luu o bien initial_val) cua do thi G:
    initial_val = 0
    for (each v in V)
        if (visited[v] == false)
        {
            DFS(adj, V, visited, v);
            initial_val = initial_val+1;
        }
}
```

Tìm đỉnh rẽ nhánh: thuật toán brute force

```
//Với mỗi đỉnh i của đồ thị: xóa đỉnh i, rồi thực hiện DFS(G\i)
for (each i in V)
{
    // (1) Xóa đỉnh i: bằng cách sửa lại mảng adj
    for (each j in V)
    {
        visited[j] = false;
        copy[j] = adj[i][j];
        adj[i][j]=adj[j][i]=0;
    }
    // (2) Thực hiện DFS(G\i): biến nval trả về số thành phần liên thông của đồ thị G\i
    nval = 0
    for (each j in V)
        if (visited[j] == false AND j != i)
        {
            nval = nval + 1;
            DFS(adj, n, visited, j);
        }
    if (nval > initial_val) // Số thành phần liên thông của G\i tăng so với số thành phần liên thông của G
    {
        count++;
        CutVertex[i] = true; // đỉnh i là đỉnh rẽ nhánh vì xóa i làm tăng số thành phần liên thông của đồ thị
    }
    // (3) Thêm lại đỉnh i vào đồ thị:
    for (each j in V)
        adj[i][j] = adj[j][i] = copy[j];
}
}
```

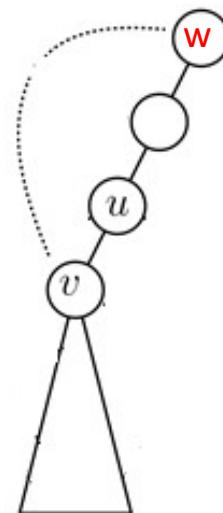
Tìm đỉnh rẽ nhánh: thuật toán DFS sửa đổi



Cạnh ngược $(4, 2)$ nối đỉnh 4 với tổ tiên của nó là đỉnh 2 → nếu xóa đỉnh 3 (cha của 4 trên cây DFS) khỏi đồ thị, vẫn tồn tại đường đi giữa 2 và 4 thông qua cạnh ngược $(4, 2)$

Nếu tồn tại cạnh ngược (v, w) trên cây DFS: thì dù xóa đỉnh u là cha của v trên cây DFS, ta vẫn có thể đi từ v đến w thông qua cạnh ngược (v, w) → xóa u không làm tăng số thành phần liên thông của đồ thị → u không phải là đỉnh rẽ nhánh

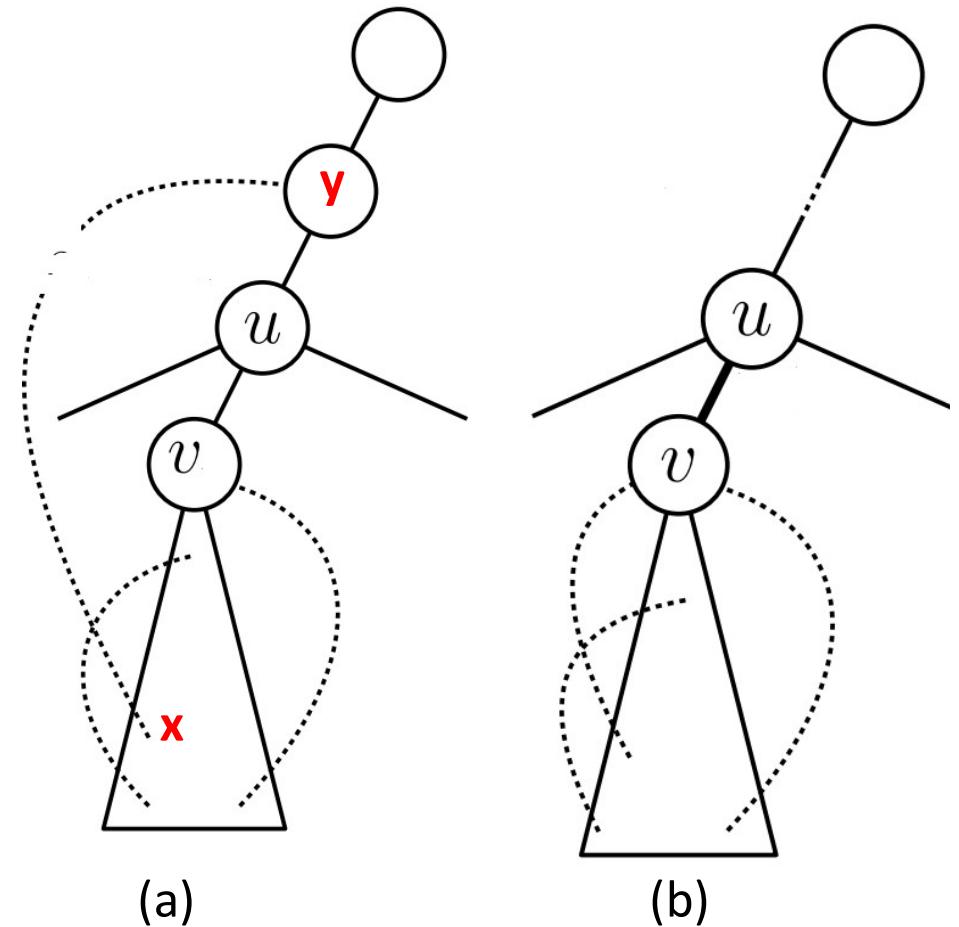
→ Đỉnh u là đỉnh rẽ nhánh khi nào ??



Đỉnh u là đỉnh rẽ nhánh khi nào ??

Giả sử trên cây DFS, đỉnh u có con là đỉnh v sao cho trên cây con $T(v)$ có gốc tại v :

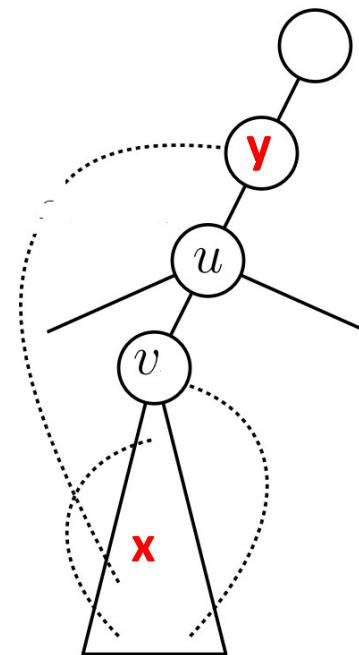
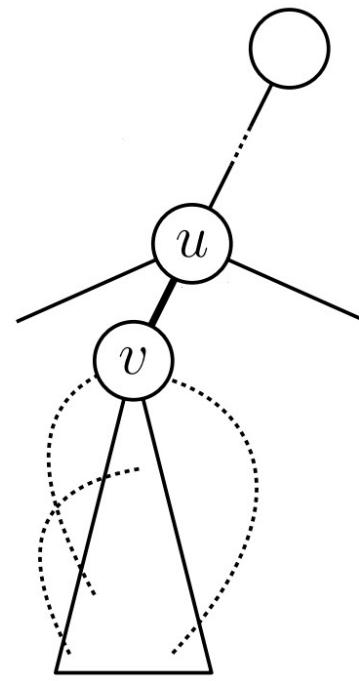
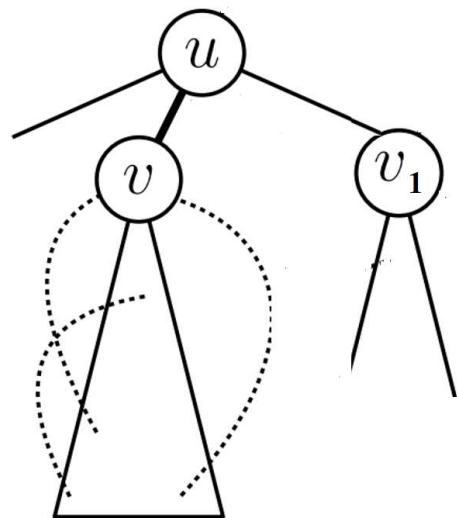
- (a) Có đỉnh x kề với đỉnh y (y được thăm trước đỉnh u), tức là (x, y) là cạnh ngược \rightarrow nếu u bị xóa khỏi đồ thị, vẫn tồn tại đường đi từ một đỉnh bất kỳ trên $T(v)$ đến $y \rightarrow$ nếu điều này đúng với tất cả các con của u thì đỉnh u không phải là đỉnh rẽ nhánh
- (b) không có đỉnh nào (gọi chung là x) kề với đỉnh y là đỉnh được thăm trước đỉnh u trên cây DFS (tức là không tồn tại cạnh ngược (x, y)) \rightarrow khi đó, nếu xóa đỉnh u khỏi đồ thị, sẽ không tồn tại đường đi giữa 1 đỉnh bất kỳ trên $T(v)$ và một đỉnh được thăm trước đỉnh $u \rightarrow$ cây $T(v)$ sẽ bị ngắt kết nối khỏi đồ thị nếu đỉnh u bị xóa khỏi đồ thị \rightarrow đỉnh u là đỉnh rẽ nhánh



Đỉnh u là đỉnh rẽ nhánh khi nào ??

Đỉnh u là đỉnh rẽ nhánh nếu một trong hai trường hợp sau xảy ra:

- u là gốc của cây DFS và u có nhiều hơn 1 con
- u không là gốc cây DFS và **u có 1 con v sao cho không có đỉnh nào trên cây $T(v)$ kè với tổ tiên của u trên cây DFS (tức là không tồn tại cạnh ngược nối một đỉnh thuộc $T(v)$ với một đỉnh tổ tiên của u)**



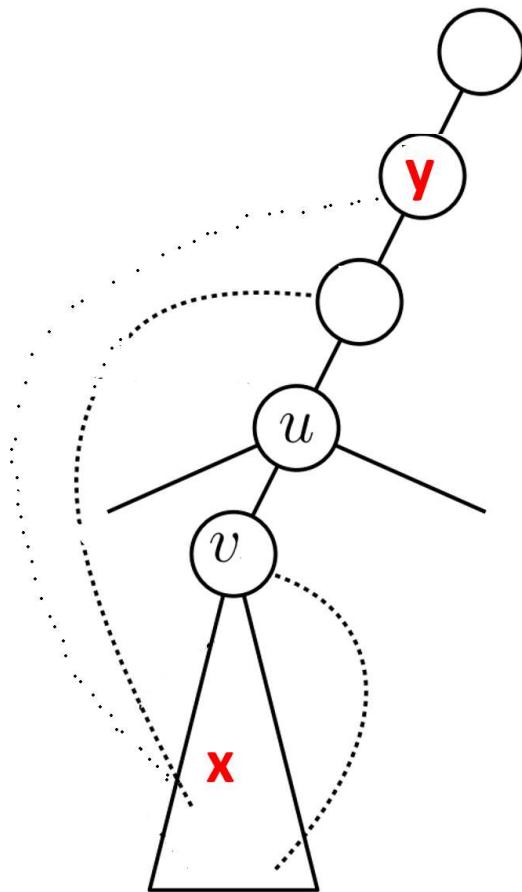
Xác định trường hợp này thế nào???

Đỉnh u là đỉnh rẽ nhánh khi nào ??

Vậy với mỗi đỉnh x của cây DFS, ta sẽ xác định xem nút nào là tổ tiên ở mức cao nhất trên cây của x (kí hiệu là y) mà x có thể tới được:

có cạnh ngược (x, y)

tức là: y là nút được thăm sớm nhất trong số tất cả các tổ tiên của x ở trên cây và có cạnh ngược (x, y)



Vậy: với mỗi đỉnh x ta sẽ lưu thời điểm bắt đầu thăm của đỉnh y .

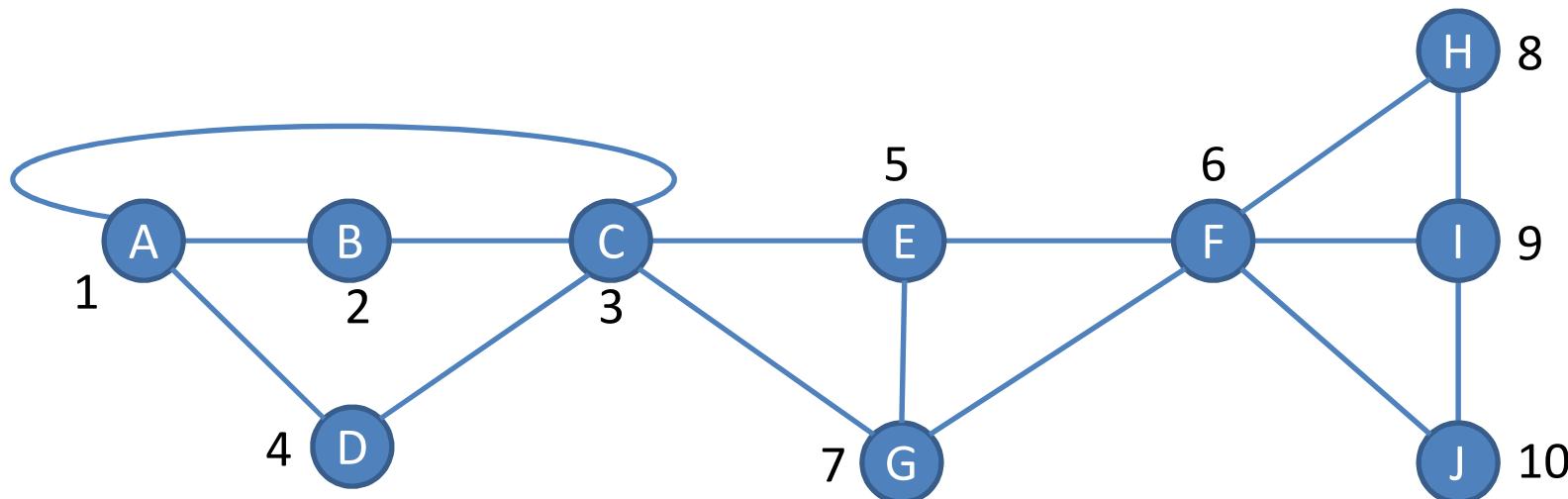
và kí hiệu: $\text{low}[x] = \text{thời điểm bắt đầu thăm } y$

Đỉnh u là đỉnh rẽ nhánh khi nào ??

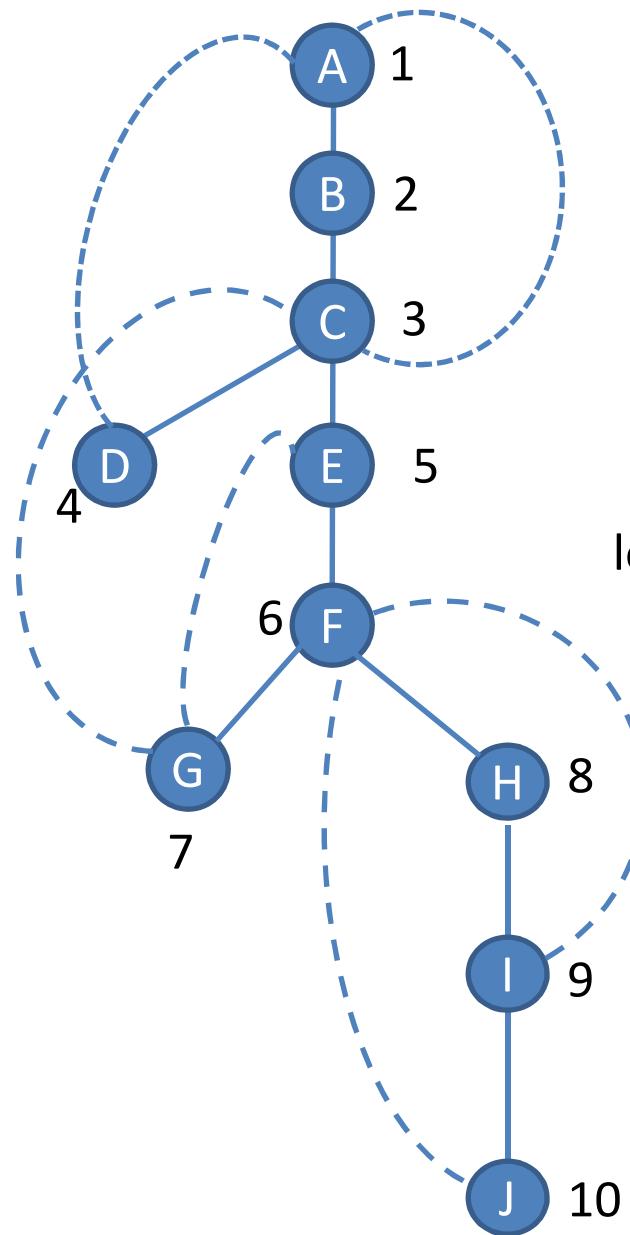
Với mỗi đỉnh u : lưu

- $d[u]$ là thời điểm bắt đầu thăm đỉnh u
- $low[u] = \min \{d[u],$
 $d[y]:$ tồn tại cạnh ngược (x, y) với x là con cháu của u , y là tổ tiên của $u\}$

DFS(A):



Đỉnh u là đỉnh rẽ nhánh khi nào ??



Với mỗi đỉnh u : lưu

- $d[u]$ là thời điểm bắt đầu thăm đỉnh u
- $low[u] = \min \{d[u],$

$d[y]:$ tồn tại cạnh ngược (x, y) với x là con cháu của u , y là tổ tiên của $u\}$

| A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 1 | 1 | 1 | 3 | 3 | 3 | 6 | 6 | 6 |

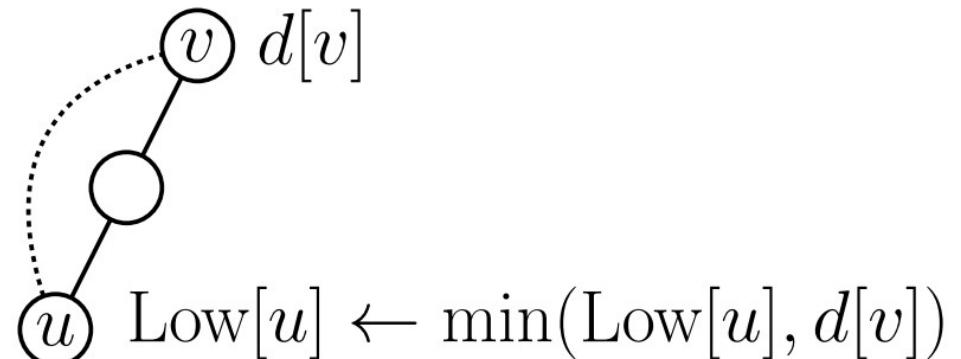
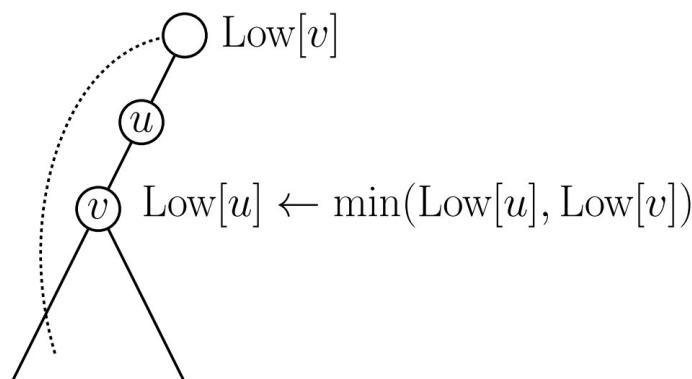
Đỉnh u là đỉnh rẽ nhánh khi nào ??

Với mỗi đỉnh u : lưu

- $d[u]$ là thời điểm bắt đầu thăm đỉnh u
- $\text{low}[u] = \min \{d[u],$
 $d[y]: \text{tồn tại cạnh ngược } (x, y) \text{ với } x \text{ là con cháu của } u, y \text{ là tổ tiên của } u\}$

Cách tính $\text{low}[u]$: giả sử đang gọi $\text{DFS}(u)$

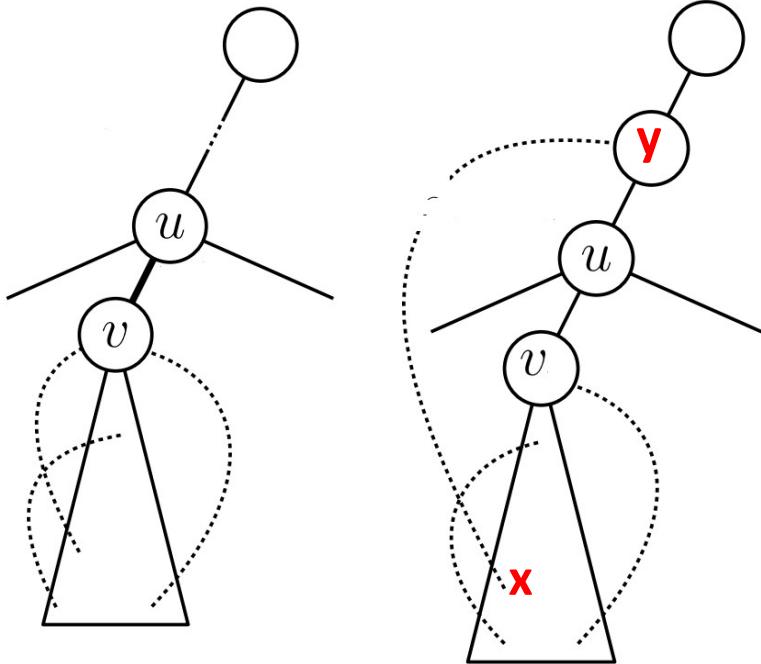
- Khởi tạo: $\text{low}[u] = d[u]$
- For each $v \in \text{Adj}[u]$:
 - Cạnh cây (u, v): $\text{low}[u] = \min(\text{low}[u], \text{low}[v])$ $\rightarrow v$ chưa thăm: $\text{visited}[v]=\text{false}$
 - Cạnh ngược (u, v): $\text{low}[u] = \min(\text{low}[u], d[v])$
 - v đã thăm: $\text{visited}[v]=\text{true}$
 - $\text{parent}[u] \neq v$



Đỉnh u là đỉnh rẽ nhánh khi nào ??

Đỉnh u là đỉnh rẽ nhánh nếu một trong hai trường hợp sau xảy ra:

- u là gốc của cây DFS và u có nhiều hơn 1 con
- u không là gốc cây DFS và u có 1 con v sao cho không có đỉnh nào trên cây $T(v)$ kè với tổ tiên của u trên cây DFS (tức là không tồn tại cạnh ngược nối một đỉnh thuộc $T(v)$ với một đỉnh tổ tiên của u)



Xác định trường hợp này thế nào???



```
if (parent[u] != NULL && low[v] >= d[u])  
    cutVertex[u] = true;
```

Tìm đỉnh rẽ nhánh

DFS(u)

```
1.   visited[u] = true; //Thăm đỉnh  $u$ 
2.   d[u] = low[u] = ++time;
3.   numChild = 0;
4.   for each  $v \in Adj[u]$ 
5.       if (visited[v] == false) {
6.           numChild++;
7.           parent[v] ←  $u$ ;
8.           DFS(v);
9.           low[u] = min(low[u], low[v]);
10.          //TH1:  $u$  là gốc của DFS và có nhiều hơn 1 con:
11.          if (parent[u] == NULL && numChild > 1) cutVertex[u] = true;
12.          //TH2:  $u$  không là gốc của cây DFS và giá trị low của đỉnh con cháu  $\geq d[u]$ 
13.          if (parent[u] != NULL && low[v] >= d[u]) cutVertex[u] = true;
14.      }
15.      else if (parent[u] != v) low[u] = min(low[u], d[v]);
```

```
void main()
1.   for each  $u \in V$ 
2.       parent[u] = NULL;
3.       visited[u] = false;
4.       cutVertex[u] = false;
5.   time = 0;
6.   for each  $u \in V$ 
7.       if (visited[u] == false) DFS(u);
8.   for each  $u \in V$  //In danh sách đỉnh rẽ nhánh
9.       if (cutVertex[u])
10.           cout << u << endl;
```

Tìm cạnh cầu

Yêu cầu: Tìm tất cả cạnh cầu (bridge) của đơn đồ thị vô hướng $G = (V, E)$

Trả lời:

- Brute force: thời gian tính $O(E(V + E))$
- DFS sửa đổi: thời gian tính $O(E + V)$

Cạnh (u, v) được gọi là cạnh cầu trên đồ thị G nếu xóa cạnh (u, v) sẽ không còn đường đi giữa đỉnh u và v trên đồ thị G

Tìm cạnh cầu: thuật toán brute force

- Brute force: với mỗi cạnh (u, v) của đồ thị
 - Xóa cạnh (u, v) khỏi đồ thị : $O(1)$
 - Kiểm tra tính liên thông của đồ thị bằng BFS/DFS : $O(V+E)$
(nếu đồ thị không liên thông \rightarrow cạnh (u, v) là cạnh cầu)
 - Thêm lại cạnh (u, v) vào đồ thị : $O(1)$

➔ Thời gian tính là $O(E(V + E))$

(đồ thị biểu diễn bởi ma trận kề)

Tìm cạnh cầu: thuật toán brute force $O(E(V+E))$

- adj [] []: ma trận kè kích thước VxV
- bridge [e] = true nếu cạnh e là cạnh cầu ($e \in E$)

```
function find_bridges(adj[()][], E)
{
    for (each e in E)
    {
        u = edge[e].dau; v = edge[e].cuoi;
        for (each i in V) visited[i] = false;
        //1. Xoa canh (u, v)
        adj[u][v]=adj[v][u]=0;
        //2. Kiem tra co con duong di khac tu u den v hay khong bang cach thuc hien BFS(u):
        Queue.push(u);
        visited[u]=true;
        existPath = false;
        while (!Queue.empty())
        {
            x = Queue.front();
            if (x==v) {existPath = true; break;}
            Queue.pop();
            for (j in V)
                if (adj[x][j] == 1 && !visited[j])
                {
                    Queue.push(j);
                    visited[j]= true;
                }
        }//end while
        if (existPath == false) //khong ton tai duong di tu u den v sau khi xoa canh e(u,v) -> canh e La canh cau
            bridge[e]=true;
        //3. Them lai canh e(u, v):
        adj[u][v]=adj[v][u]=1;
    }
}
```

Tìm cạnh cầu: thuật toán DFS sửa đổi

(u, v) là cạnh cầu nếu: sau khi xóa cạnh (u, v) khỏi đồ thị, thì trên đồ thị sẽ không còn tồn tại đường đi giữa u và v

Với mỗi cạnh (u, v) (với $d[u] < d[v]$ trên cây DFS):

- Nếu có cạnh ngược từ v tới tổ tiên của u tức là $low[v] \leq d[u]$ thì tồn tại một đường đi khác giữa u và v bằng cách đi qua cạnh ngược đó
- (u, v) không phải là cạnh cầu
- Nếu $low[v] > d[u]$ thì (u, v) là cạnh cầu

Tìm cạnh cầu

```
void main()
1.   for each  $u \in V$ 
2.       parent[u] = NULL;
3.       visited[u] = false;
4.   time = 0;
5.   for each  $u \in V$ 
6.       if (visited[u] == false) DFS(u);
```

DFS(u)

```
1.   visited[u] = true; //Thăm đỉnh  $u$ 
2.   d[u] = low[u] = ++time;
3.   for each  $v \in Adj[u]$ 
4.       if (visited[v] == false) {
5.           parent[v]  $\leftarrow u$ ;
6.           DFS(v);
7.           low[u] = min(low[u], low[v]);
8.           if (low[v] > d[u]) print(cạnh cầu (u, v));
9.       }
10.      else if (parent[u] != v) low[u] = min(low[u], d[v]);
```

Các ứng dụng của BFS/DFS

1. Sắp xếp topo
2. Định hướng đồ thị
3. Đường đi từ s đến t
4. Đồ thị hai phía
5. Tìm đường đi dài nhất trên cây
6. Bài toán tìm bao đóng truyền ứng
7. Phát hiện chu trình
8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông
- 9. Tính liên thông của đồ thị vô hướng**
10. Kiểm tra tính liên thông mạnh

Bài toán về tính liên thông

Bài toán: Cho đồ thị vô hướng $G = (V,E)$. Hỏi đồ thị gồm bao nhiêu thành phần liên thông, và từng thành phần liên thông gồm các đỉnh nào?

- Giải: Sử dụng DFS (BFS) :

- Mỗi lần gọi đến DFS (BFS) ở trong chương trình chính sẽ sinh ra một thành phần liên thông

DFS giải bài toán liên thông

(* Main Program*)

1. **for** $s \in V$
2. $\text{visited}[s] \leftarrow \text{false}$
3. **for** $s \in V$
4. **if** ($\text{visited}[s] == \text{false}$)
5. $\text{DFS}(s)$

$\text{DFS}(s)$

1. $\text{visited}[s] \leftarrow \text{true}$ // Thăm đỉnh s
2. **for each** $v \in \text{Adj}[s]$
3. **if** ($\text{visited}[v] == \text{false}$)
4. $\text{DFS}(v)$



(* Main Program*)

1. **for** $u \in V$
2. $\text{id}[u] \leftarrow 0;$
3. $\text{cnt} \leftarrow 0$; // cnt – số lượng tplt
4. **for** $u \in V$
5. **if** ($\text{id}[u] == 0$) {
6. $\text{cnt} \leftarrow \text{cnt} + 1$;
7. $\text{DFS}(u)$;
8. }

$\text{DFS}(u)$

1. $\text{id}[u] \leftarrow \text{cnt};$
2. **for each** $v \in \text{Adj}[u]$
3. **if** ($\text{id}[v] == 0$)
4. $\text{DFS}(v);$

BFS giải bài toán liên thông

```
void BFS(s) {  
    // Tìm kiếm theo chiều rộng bắt đầu từ đỉnh s  
    visited[s] ← 1; //đã thăm  
    Q ← ∅; enqueue(Q,s); // Nạp s vào Q  
    while (Q ≠ ∅)  
    {  
        u ← dequeue(Q); // Lấy u khỏi Q  
        for v ∈ Adj[u]  
            if (visited[v] == 0) //chưa thăm  
            {  
                visited[v] ← 1; //đã thăm  
                enqueue(Q,v) // Nạp v vào Q  
            }  
    }  
}  
void main ()  
{  
    for s ∈ V // Khởi tạo  
        visited[s] ← 0;  
  
    for s ∈ V  
        if (visited[s]==0) BFS(s);  
}
```



BFS(s)

1. $id[s] \leftarrow cnt$
2. $Q \leftarrow \emptyset$; enqueue(Q, s); // Nạp s vào Q
3. **while** ($Q \neq \emptyset$)
4. {
5. $u \leftarrow dequeue(Q)$; // Lấy u khỏi Q
6. **for** $v \in Adj[u]$
7. **if** ($id[v] == 0$) //v chưa thăm
8. {
9. $id[v] \leftarrow cnt$;
10. enqueue(Q, v) // Nạp v vào Q
11. }
12. }

(* Main Program*)

1. **for** $s \in V$
2. $id[s] \leftarrow 0$
3. $cnt \leftarrow 0$ //cnt – số lượng tplt
4. **for** $s \in V$
5. **if** ($id[s] == 0$){
6. $cnt \leftarrow cnt + 1$
7. BFS(s)
8. }

Các ứng dụng của BFS/DFS

1. Sắp xếp topo
2. Định hướng đồ thị
3. Đường đi từ s đến t
4. Đồ thị hai phía
5. Tìm đường đi dài nhất trên cây
6. Bài toán tìm bao đóng truyền ứng
7. Phát hiện chu trình
8. Tìm cầu, rẽ nhánh của đồ thị vô hướng liên thông
9. Tính liên thông của đồ thị
- 10. Kiểm tra tính liên thông mạnh**

10. Kiểm tra tính liên thông mạnh

Bài toán 1: Cho đồ thị có hướng $G=(V, E)$. Hãy kiểm tra xem đồ thị G có phải liên thông mạnh hay không?

Bài toán 2: Cho đồ thị có hướng $G=(V, E)$. Hãy kiểm tra xem đồ thị G có bao nhiêu thành phần liên thông mạnh.

10. Kiểm tra tính liên thông mạnh

Bài toán 1: Cho đồ thị có hướng $G=(V, E)$. Hãy kiểm tra xem đồ thị G có phải liên thông mạnh hay không?

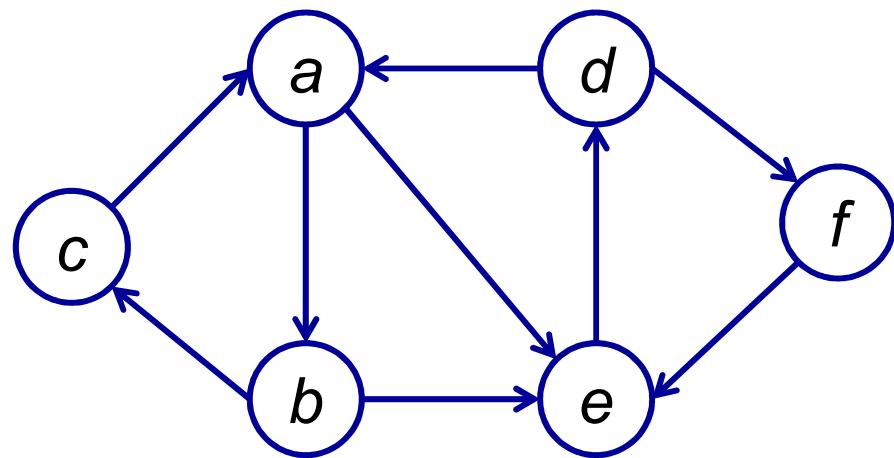
Mệnh đề: *Đồ thị có hướng $G=(V, E)$ là liên thông mạnh khi và chỉ khi luôn tìm được đường đi từ một đỉnh v đến tất cả các đỉnh còn lại và luôn tìm được đường đi từ tất cả các đỉnh thuộc $V \setminus \{v\}$ đến v .*

Thuật toán kiểm tra tính liên thông mạnh

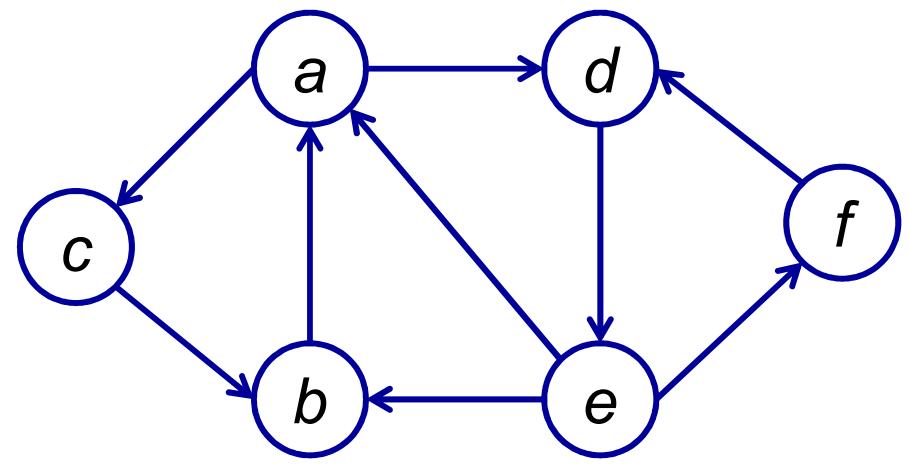
- Chọn $v \in V$ là một đỉnh tùy ý.
- Thực hiện DFS(v) trên G . Nếu tồn tại đỉnh u không được thăm thì G không liên thông mạnh và thuật toán kết thúc. Trái lại thực hiện tiếp
- Thực hiện DFS(v) trên $G^T = (V, E^T)$, với E^T thu được từ E bởi việc đảo ngược hướng các cung. Nếu tồn tại đỉnh u không được thăm thì G không liên thông mạnh, nếu trái lại G là liên thông mạnh.

Thời gian tính = thời gian DFS = $O(|V|+|E|)$

Thuật toán kiểm tra tính liên thông mạnh



Đồ thị G



Đồ thị G^T

Thuật toán kiểm tra tính liên thông mạnh

- Chọn $v \in V$ là một đỉnh tùy ý.
- Thực hiện DFS(v) trên G . Nếu tồn tại đỉnh u không được thăm thì G không liên thông mạnh và thuật toán kết thúc. Trái lại thực hiện tiếp
- Thực hiện DFS(v) trên $G^T = (V, E^T)$, với E^T thu được từ E bởi việc đảo ngược hướng các cung. Nếu tồn tại đỉnh u không được thăm thì G không liên thông mạnh, nếu trái lại G là liên thông mạnh.

Câu hỏi: Nếu đồ thị G được biểu diễn bởi ma trận kề A

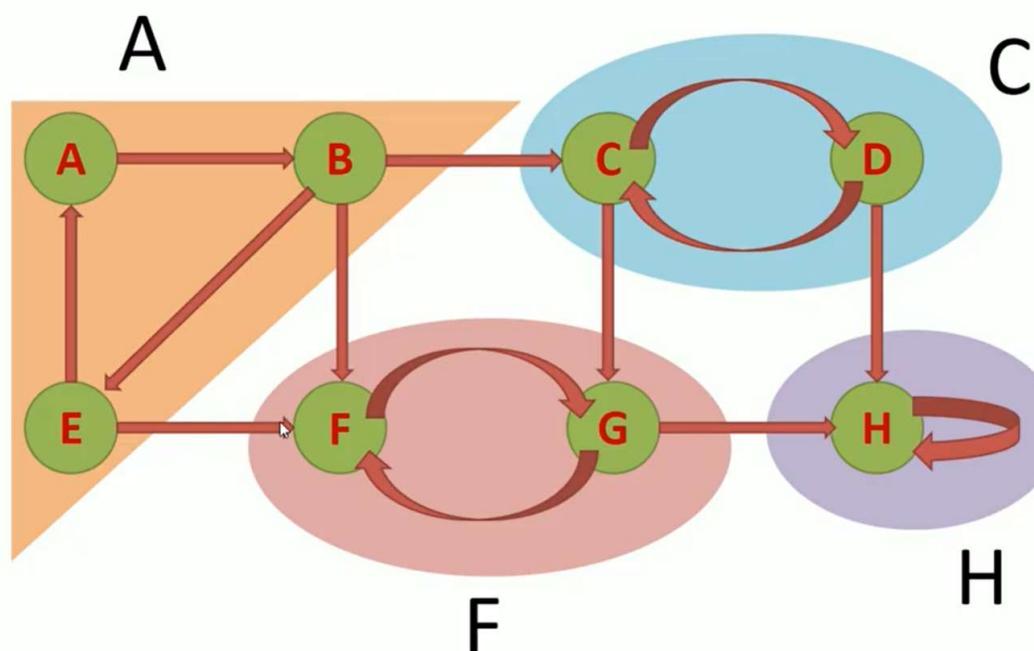
→ Làm thế nào để thu được đồ thị G^T từ ma trận A ?

Đồ thị đảo hướng (đồ thị chuyển vị)

- Cho đồ thị có hướng $G=(V,E)$. Ta gọi **đồ thị đảo hướng** (đồ thị chuyển vị) của đồ thị G là đồ thị có hướng $G^T = (V, E^T)$, với $E^T = \{(u, v): (v, u) \in E\}$, nghĩa là tập cung E^T thu được từ E bởi việc đảo ngược hướng của tất cả các cung.
- Để thấy nếu A là ma trận kè của G thì ma trận chuyển vị A^T là ma trận kè của G^T (điều này giải thích tên gọi đồ thị chuyển vị).

Liệt kê các SCC

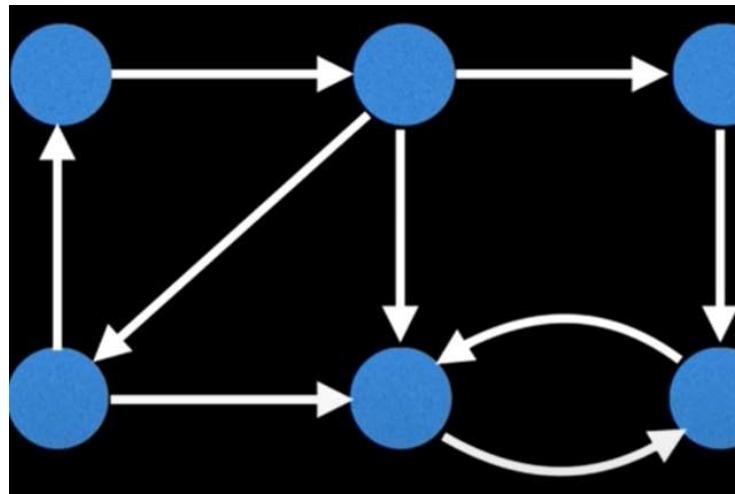
Bài toán 2: Cho đồ thị có hướng $G=(V, E)$. Hãy kiểm tra xem đồ thị G có bao nhiêu thành phần liên thông mạnh(Strongly Connected Component - SCC). Liệt kê tất cả các đỉnh trong mỗi thành phần liên thông mạnh



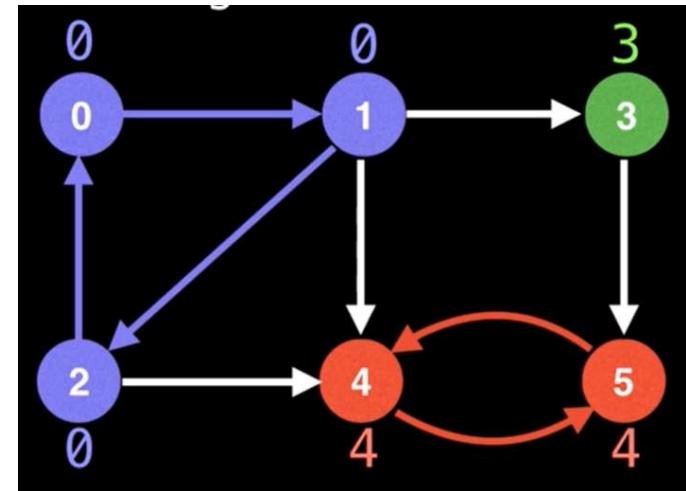
Liệt kê các SCC

Thuật toán Tarjan:

1. Thực hiện thuật toán DFS trên đồ thị, sẽ tạo ra cây/rừng DFS
2. Mỗi thành phần liên thông mạnh sẽ là một cây con trong 1 cây DFS:
 - Nếu ta có thể tìm được **gốc của cây con** đó, thì toàn bộ các nút trong cây này chính là 1 thành phần liên thông mạnh của đồ thị



DFS



Liệt kê các SCC

Thuật toán Tarjan:

1. Thực hiện thuật toán DFS trên đồ thị, sẽ tạo ra cây/rừng DFS
2. Mỗi thành phần liên thông mạnh sẽ là một cây con trong 1 cây DFS:
 - Nếu ta có thể tìm được **gốc của cây con** đó, thì toàn bộ các nút trong cây này chính là 1 thành phần liên thông mạnh của đồ thị

Kết thúc DFS(u) nếu $d[u] = low[u]$ thì u chính là gốc

Vậy kết thúc DFS(u) nếu $d[u] = low[u] \rightarrow$ ta tìm được 1 thành phần liên thông mạnh mới:

- Gồm đỉnh u
- Và các đỉnh còn lại là những đỉnh nào??

Sử dụng stack:

- Trong quá trình thực hiện DFS (u): mỗi lần thăm 1 đỉnh mới, ta đẩy đỉnh đó vào stack; update giá trị low cho các đỉnh
- Khi kết thúc DFS(u): nếu $d[u] = low[u]$: ta bắt đầu lần lượt đẩy từng đỉnh ra khỏi stack cho đến khi lấy ra được đỉnh $u \rightarrow$ tất cả các đỉnh lấy ra này sẽ thuộc vào 1 SCC

Liệt kê các SCC

Với mỗi đỉnh u: lưu

- $d[u]$ là thời điểm bắt đầu thăm đỉnh u
- $low[u] = \min \{d[u],$
 $d[y]:$ tồn tại cạnh ngược (x, y) với x là con cháu của u , y là tổ tiên của $u\}$

Cách tính $low[u]$: giả sử đang gọi $DFS(u)$

- Khởi tạo: $low[u] = d[u]$
- For each $v \in \text{Adj}[u]$:
 - **Cạnh cây (u, v) :** $low[u] = \min (low[u], low[v])$ → v chưa thăm: $\text{visited}[v]=\text{false}$
 - **Cạnh ngược (u, v) :** $low[u] = \min(\text{low}[u], d[v])$
 - v đã thăm: $\text{visited}[v]=\text{true}$
 - v đang trong stack

Liệt kê các SCC: FULL CODE

```
int time;
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;     //adjacency lists

    int *d = new int[V];
    int *low = new int[V];
    bool *onStack = new bool[V];
    stack<int> *STACK = new stack<int>();
    void DFS(int u);

public:
    Graph(int V);
    void addEdge(int v, int w);    // them canh (v, w) vao do thi
    void findSCC();    // in tat ca cac SCC cua do thi
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); //do thi co huong
}
```

```

void Graph::DFS(int u)
{
    d[u] = low[u] = ++time;
    STACK->push(u);
    onStack[u] = true;

    //Duyet qua ds ke cua dinh u:
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;
        if (d[v] == -1) //v chua tham
        {
            DFS(v);
            // Case 1: (u, v) canh cay
            low[u] = min(low[u], low[v]);
        }
        // Case 2: (u, v) canh vong
        else if (onStack[v] == true)
            low[u] = min(low[u], d[v]);
    }

    // head node found, pop the stack and print an SCC
    int w = 0;
    if (low[u] == d[u])
    {
        while (STACK->top() != u)
        {
            w = (int) STACK->top();
            cout << w << " ";
            onStack[w] = false;
            STACK->pop();
        }
        w = (int) STACK->top();
        cout << w << "\n";
        onStack[w] = false;
        STACK->pop();
    }
}

```

```

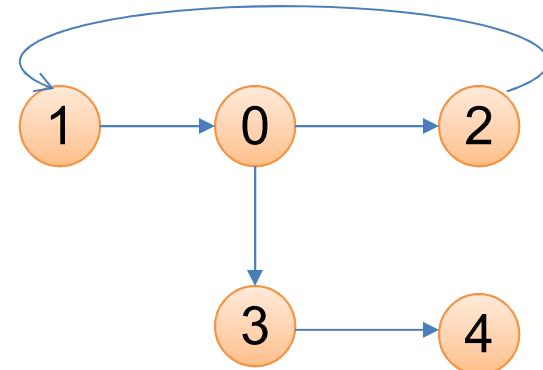
void Graph::findSCC()
{
    for (int i = 0; i < v; i++)
    {
        d[i] = -1;
        low[i] = -1;
        onStack[i] = false;
    }

    for (int i = 0; i < v; i++)
        if (d[i] == -1) //i chua duoc tham
            DFS(i);
}

int main()
{
    Graph g1(5);
    g1.addEdge(1, 0); g1.addEdge(0, 2);
    g1.addEdge(2, 1); g1.addEdge(0, 3);
    g1.addEdge(3, 4);

    time = 0;
    cout << "\nCac SCCs cua do thi: \n";
    g1.findSCC();
    return 0;
}

```



Cac SCCs cua do thi:
4
3
1 2 0

Nội dung

1. Đồ thị và cách biểu diễn đồ thị
2. Duyệt đồ thị
- 3. Một số bài toán cơ bản trên đồ thị**