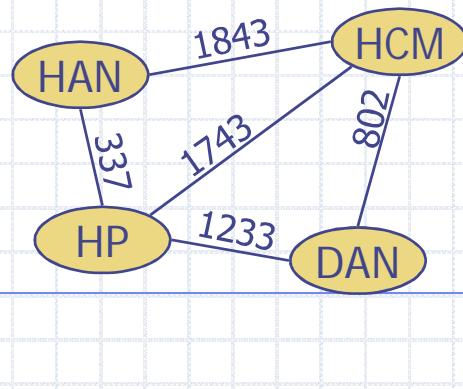


## CHƯƠNG 7

# Đồ thị và các thuật toán đồ thị



## NỘI DUNG

### 1. Đồ thị

Đồ thị vô hướng, Đồ thị có hướng, Tính liên thông của đồ thị

### 2. Biểu diễn đồ thị

Biểu diễn đồ thị bởi ma trận, Danh sách kè, Danh sách cạnh

### 3. Các thuật toán duyệt đồ thị

Thuật toán tìm kiếm theo chiều sâu, Thuật toán tìm kiếm theo chiều rộng

### 4. Một số ứng dụng của tìm kiếm trên đồ thị

Bài toán đường đi, Bài toán liên thông,

Đồ thị không chứa chu trình và bài toán sắp xếp tópô, Bài toán tô màu đỉnh đồ thị

### 5. Bài toán cây khung nhỏ nhất

Thuật toán Kruscal, Cấu trúc dữ liệu biểu diễn phân hoạch,

### 6. Bài toán đường đi ngắn nhất

Thuật toán Dijkstra, Cài đặt thuật toán với các cấu trúc dữ liệu

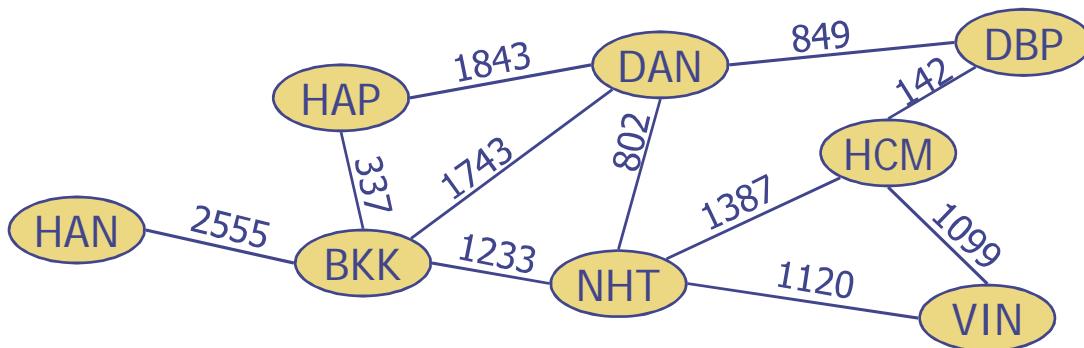
# 1. Đồ thị

◆ Đồ thị là cặp  $(V, E)$ , trong đó

- $V$  là tập đỉnh
- $E$  là họ các cặp đỉnh gọi là các cạnh

◆ Ví dụ:

- Các đỉnh là các sân bay
- Các cạnh thể hiện đường bay nối hai sân bay
- Các số trên cạnh có thể là chi phí (thời gian, khoảng cách)



3

Nguyễn Đức Nghĩa - Bộ môn KHMT ĐHBKHN

## Các kiểu cạnh

◆ Cạnh có hướng (Directed edge)

- Cặp có thứ tự gồm hai đỉnh  $(u,v)$
- Đỉnh  $u$  là đỉnh đầu
- Đỉnh  $v$  là đỉnh cuối
- Ví dụ, chuyến bay



◆ Cạnh vô hướng (Undirected edge)

- Cặp không có thứ tự gồm 2 đỉnh  $(u,v)$
- Ví dụ, tuyến bay



◆ Đồ thị có hướng (digraph)

- Các cạnh có hướng
- Ví dụ, mạng truyền tin

◆ Đồ thị vô hướng (Undirected graph/graph)

- Các cạnh không có hướng
- Ví dụ, mạng tuyến bay

Nguyễn Đức Nghĩa - Bộ môn KHMT ĐHBKHN

4

# Ứng dụng

## ◆ Mạch lôgic (Electronic circuits)

- Mạch in
- Mạch tích hợp

## ◆ Mạng giao thông (Transportation networks)

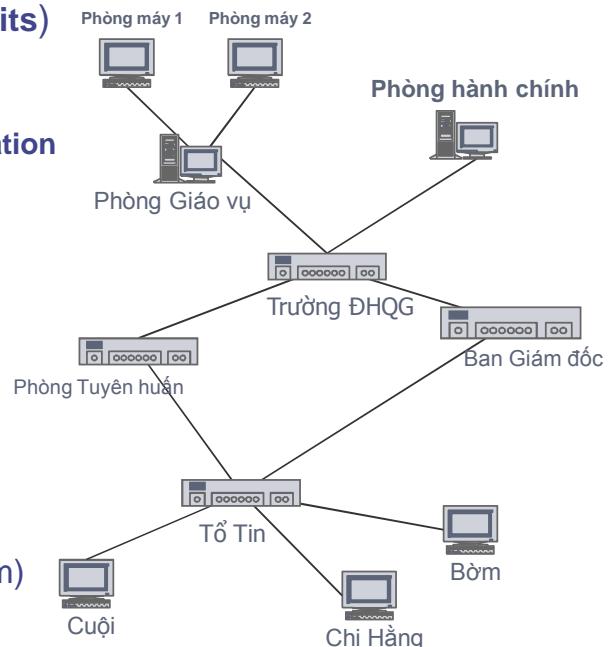
- Mạng xa lộ
- Mạng tuyến bay

## ◆ Mạng máy tính (Computer networks)

- Mạng cục bộ
- Internet
- Web

## ◆ Cơ sở dữ liệu (Databases)

- Sơ đồ quan hệ thực thể (Entity-relationship diagram)



# Thuật ngữ

## ◆ Đầu mút của cạnh

- U và V là các đầu mút của cạnh a

## ◆ Cạnh kề với đỉnh

- a, d, và b kề với đỉnh V

## ◆ Đỉnh kề

- U và V là kề nhau

## ◆ Độ bậc của đỉnh

- X có độ bậc 5

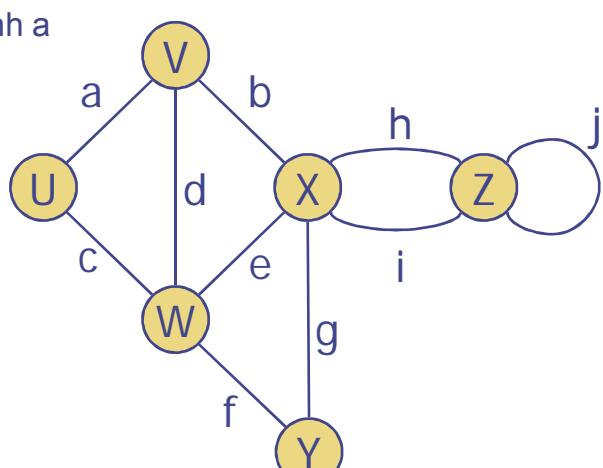
## ◆ Cạnh lặp

- h và i là các cạnh lặp

## ◆ Khuyên

- j là khuyên

## ◆ Đơn đồ thị: Không chứa cạnh lặp và khuyên



# Thuật ngữ (tiếp tục)

---

## ◆ Đường đi

- Dãy các đỉnh (hoặc dãy các cạnh), trong đó hai đỉnh liên tiếp là có cạnh nối:

$P: s = v_0, v_1, \dots, v_{k-1}, v_k = t,$

$(v_{i-1}, v_i)$  là cạnh của đồ thị,  $i=1, 2, \dots, k.$

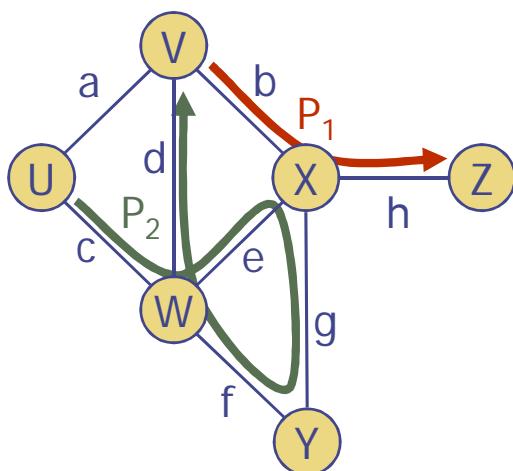
- Độ dài của đường đi là số cạnh trên đường đi ( $k$ ).
- $s$  - đỉnh đầu và  $t$  - đỉnh cuối của đường đi  $P$

## ◆ Đường đi đơn

- Các đỉnh trên đường đi là phân biệt

# Thuật ngữ (tiếp tục)

---



## ◆ Ví dụ

- $P_1 = V, X, Z$  (dãy cạnh:  $b, h$ ) là đường đi đơn
- $P_2 = U, W, X, Y, W, V$  ( $P_2 = c, e, g, f, d$ ) là đường đi nhưng không là đơn

# Thuật ngữ (tiếp)

## ◆ Chu trình

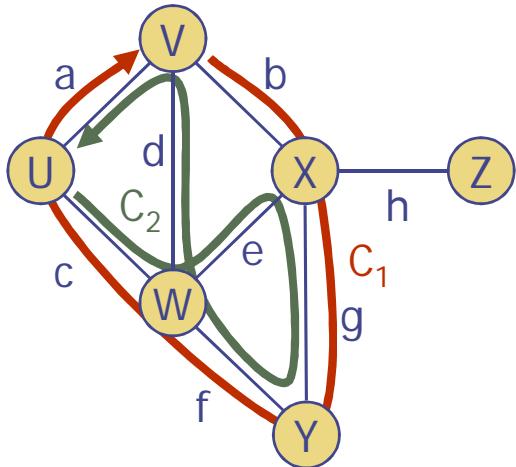
- Đường đi gồm các cạnh phân biệt có đỉnh đầu trùng đỉnh cuối

## ◆ Chu trình đơn

- Ngoại trừ đầu trùng cuối, không còn hai đỉnh nào giống nhau

## ◆ Ví dụ

- $C_1 = V, X, Y, W, U$  là CT đơn
- $C_2 = U, W, X, Y, W, V$  là chu trình không là đơn



Nguyễn Đức Nghĩa - Bộ môn KHMT ĐHBKHN

9

# Tính chất

## Tính chất 1

$$\sum_v \deg(v) = 2m$$

CM: mỗi cạnh được đếm 2 lần

Ký hiệu

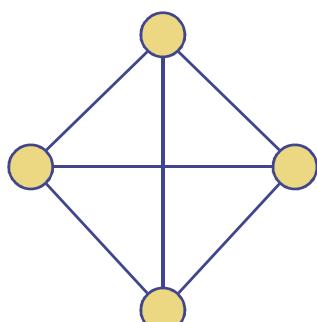
$n$	số đỉnh
$m$	số cạnh
$\deg(v)$	bậc của đỉnh $v$

## Tính chất 2

Trong đơn đồ thị vô hướng (đồ thị không có cạnh lặp và khuyên)

$$m \leq n(n-1)/2$$

CM: mỗi đỉnh có bậc không quá  $(n-1)$



Ví dụ

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Tương tự có những cận cho đồ thị có hướng

# Graph ADT

---

## ◆ Các phép toán cơ bản (Basic Graph operations)

- khởi tạo/create (số đỉnh, isDirected)
- huỷ/destroy
- nhận số cạnh / get number of edges
- nhận số đỉnh / get number of vertices
- cho biết đồ thị là có hướng hay vô hướng / tell whether graph is directed or undirected
- bổ sung cạnh / insert an edge
- loại bỏ cạnh / remove an edge
- có cạnh nối giữa hai đỉnh / tell whether an edge exists between two vertices
- duyệt các đỉnh kề của một đỉnh cho trước / An iterator that processes all vertices adjacent to a given vertex

# Các bài toán xử lý đồ thị

---

- ◆ Tính giá trị của một số đặc trưng số của đồ thị (số liên thông, sắc số, ...)
- ◆ Tìm một số tập con cạnh đặc biệt (chẳng hạn, cặp ghép, bè, chu trình, cây khung, ...)
- ◆ Tìm một số tập con đỉnh đặc biệt (chẳng hạn, phủ đỉnh, phủ cạnh, tập độc lập,...)
- ◆ Trả lời truy vấn về một số tính chất của đồ thị (liên thông, phẳng, ...)
- ◆ Các bài toán tối ưu trên đồ thị: Cây khung nhỏ nhất, đường đi ngắn nhất, luồng cực đại trong mạng, ...
- ◆ ...

---

---

## 2 Biểu diễn đồ thị

Nguyễn Đức Nghĩa - Bộ môn KHMT ĐHBKHN

13

---

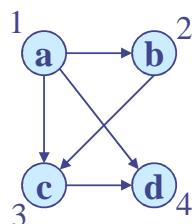
## 2. Biểu diễn đồ thị

---

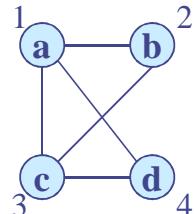
- ◆ Có nhiều cách biểu diễn,
- ◆ Việc lựa chọn cách biểu diễn phụ thuộc vào từng bài toán cụ thể cần xét, từng thuật toán cụ thể cần cài đặt.
- ◆ Có hai vấn đề chính cần quan tâm khi lựa chọn cách biểu diễn:
  - Bộ nhớ mà cách biểu diễn đó đòi hỏi
  - Thời gian cần thiết để trả lời các truy vấn thường 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  $v$  ?

# Ma trận kè (Adjacency Matrix)

- ◆  $n \times n$  ma trận  $A$ .
- ◆ Các đỉnh được đánh số từ 1 đến  $|V|$  theo 1 thứ tự nào đó.
- ◆  $A$  xác định bởi:  $A[i, j] = a_{ij} = \begin{cases} 1 & \text{nếu } (i, j) \in E \\ 0 & \text{nếu } i = j \end{cases}$



	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

$$A = A^T \text{ đối với đồ thị vô hướng.}$$

## Ma trận kè

- Chú ý về sử dụng ma trận kè:
  - ♦ Dòng toàn không ~đỉnh cô lập.
  - ♦  $M[i, i] = 1 \leftrightarrow$  khuyên (self-loop)
- Bộ nhớ (Space)
  - ♦  $|V|^2$  bits
  - ♦ Các thông tin bổ sung, chẳng hạn chi phí trên cạnh, cần được cất giữ dưới dạng ma trận.
- Thời gian trả lời các truy vấn
  - ♦ Hai đỉnh  $i$  và  $j$  có kè nhau?  $O(1)$
  - ♦ Bổ sung hoặc loại bỏ cạnh  $O(1)$
  - ♦ Bổ sung đỉnh: tăng kích thước ma trận
  - ♦ Liệt kê các đỉnh kè của  $v$ :  $O(|V|)$  (ngay cả khi  $v$  là đỉnh cô lập).

# 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

$$C = c[i, j], i, j = 1, 2, \dots, n,$$

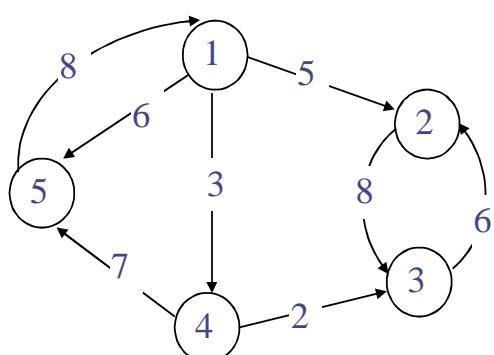
ví i

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

trong  $\theta$  là giá trị  $c(i, j)$  khi  $(i, j) \in E$ ,  $\theta$  là giá trị của  $c(i, j)$  khi  $(i, j) \notin E$ .  $\theta$  có thể là  $0, +\infty, -\infty$ .

# Ma trận trọng số

- ♦ Ví dụ



$$A = \begin{bmatrix} \theta & 5 & \theta & 3 & 6 \\ \theta & \theta & 8 & \theta & \theta \\ \theta & 6 & \theta & \theta & \theta \\ \theta & \theta & 2 & \theta & 7 \\ 8 & \theta & \theta & \theta & \theta \end{bmatrix}$$

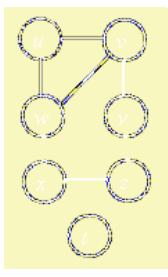
# Danh sách kè (Adjacency List)

◆ **Danh sách kè:** Với mỗi đỉnh  $v$  cất giữ danh sách các đỉnh kè của nó.

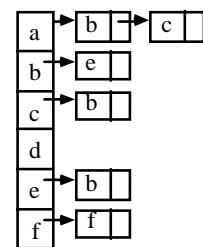
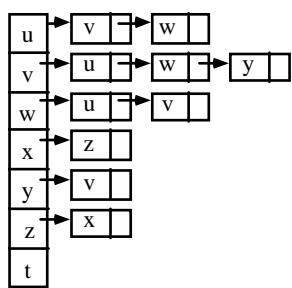
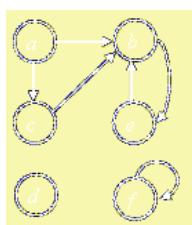
- Là mảng  $Adj$  gồm  $|V|$  danh sách.
- Mỗi đỉnh có một danh sách.
- Với mỗi  $u \in V$ ,  $Adj[u]$  bao gồm tất cả các đỉnh kè của  $u$ .

◆ **Ví dụ**

Đồ thị vô hướng



Đồ thị có hướng



## Biểu diễn đồ thị bởi danh sách kè

### Yêu cầu bộ nhớ:

◆ **Đối với đồ thị có hướng:**

- Tổng số phần tử trong tất cả các danh sách kè là
$$\sum_{v \in V} \text{out-degree}(v) = |E|$$
(out-degree( $v$ ) – số cung đi ra khỏi  $v$ )

- **Tổng cộng bộ nhớ:**  $\Theta(|V| + |E|)$

◆ **Đối với đồ thị vô hướng:**

- Tổng số phần tử trong tất cả các danh sách kè là
$$\sum_{v \in V} \text{degree}(v) = 2|E|$$
(degree( $v$ ) – số cạnh kè với  $v$ )

- **Tổng cộng bộ nhớ:**  $\Theta(|V| + |E|)$

# Biểu diễn đồ thị bởi danh sách kè

---

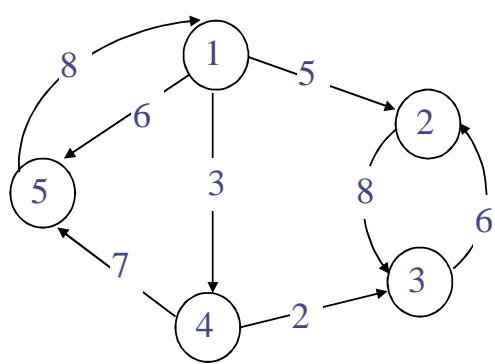
- Bộ nhớ (Space):
  - ♦  $O(|V| + |E|)$
  - ♦ Thường là nhỏ hơn nhiều so với  $|V|^2$ , nhất là đối với đồ thị thưa (sparse graph) – là đồ thị mà  $|E| = k|V|$  với  $k < 10$ .
- Thời gian trả lời các truy vấn:
  - ♦ Thêm cạnh  $O(1)$
  - ♦ Xoá cạnh Duyệt qua danh sách kè của mỗi đầu mút.
  - ♦ Thêm đỉnh Phụ thuộc vào cài đặt.
  - ♦ Liệt kê các đỉnh kè của v:  $O(<\text{số đỉnh kè}>)$  (tốt hơn ma trận kè)
  - ♦ Hai đỉnh  $i, j$  có kè nhau? Tìm kiếm trên danh sách:  $\Theta(\text{degree}(u))$ . Đánh giá trong tình huống tồi nhất là  $O(|V|) \Rightarrow$  không hiệu quả (tồi hơn ma trận kè)

## Danh sách cạnh (Edge List)

---

- ◆ Với mỗi cạnh  $e = (u, v)$  cất giữ  
 $dau[e] = u$  ,  $cuoi[e] = v$
- ◆ Nếu đồ thị có trọng số trên cạnh, thì cần có thêm một biến cất giữ  $c[e]$
- ◆ **Đây là cách chuẩn bị dữ liệu cho các đồ thị thực tế**

# Danh sách cạnh



e	dau[e]	cuoi[e]	c[e]
1	1	5	6
2	5	1	8
3	4	5	7
4	1	4	3
5	1	2	5
6	4	3	2
7	2	3	8
8	3	2	6

## Đánh giá thời gian thực hiện các thao tác

◆ $n$ đỉnh, $m$ cạnh ◆ đơn đồ thị vô hướng	Edge List	Adjacency List	Adjacency Matrix
Bộ nhớ	$n + m$	$n + m$	$n^2$
incidentEdges( $v$ )	$m$	$\deg(v)$	$n$
areAdjacent ( $v, w$ )	$m$	$\min(\deg(v), \deg(w))$	1
insertVertex( $o$ )	1	1	$n^2$
insertEdge( $v, w, o$ )	1	1	1
removeVertex( $v$ )	$m$	$\deg(v)$	$n^2$
removeEdge( $e$ )	1	1	1

---

---

### 3. Các thuật toán duyệt đồ thị

#### Graph Searching

## Duyệt đồ thị

---

---

- ◆ Ta gọi *duyệt đồ thị* (Graph Searching hoặc Graph Traversal) là việc duyệt qua mỗi đỉnh và mỗi cạnh của đồ thị.
- ◆ Ứng dụng:
  - Xây dựng các thuật toán khảo sát các tính chất của đồ thị;
  - Là thành phần cơ bản của nhiều thuật toán.
- ◆ Cần xây dựng thuật toán hiệu quả để thực hiện việc duyệt đồ thị. Ta xét hai thuật toán duyệt cơ bản:
  - *Tìm kiếm theo chiều rộng* (Breadth First Search – BFS)
  - *Tìm kiếm theo chiều sâu* (Depth First Search – DFS)

## Ý tưởng chung

---

- ◆ Trong quá trình thực hiện thuật toán, mỗi đỉnh ở một trong ba trạng thái:
  - Chưa thăm (thể hiện bởi màu trắng),
  - Đã thăm nhưng chưa duyệt xong (thể hiện bởi màu xám)
  - Đã duyệt xong (thể hiện bởi màu đen).
- ◆ Quá trình duyệt được bắt đầu từ một đỉnh  $v$  nào đó. Ta sẽ khảo sát các đỉnh đạt tới được từ  $v$ :
  - Thoạt đầu mỗi đỉnh đều có màu trắng (chưa thăm - not visited).
  - Đỉnh đã được thăm sẽ chuyển thành màu xám (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 có màu đen (đã duyệt xong).

---

## Thuật toán tìm kiếm theo chiều rộng (BFS algorithm)

# BFS

---

◆ **Input:** Đồ thị  $G = (V, E)$ , có hướng hoặc vô hướng, và đỉnh xuất phát  $s \in V$ .

◆ **Output:**

Với mọi  $v \in V$

- $d[v]$  = khoảng cách từ  $s$  đến  $v$ .
- $\pi[v]$  – đỉnh đi trước  $v$  trong đường đi ngắn nhất từ  $s$  đến  $v$ .
- Xây dựng cây BFS gốc tại  $s$  chứa tất cả các đỉnh đạt đến được từ  $s$ .

◆ Ta sẽ sử dụng màu để ghi nhận trạng thái của đỉnh trong quá trình duyệt:

- *Trắng (White)* – chưa thăm.
- *Xám (Gray)* – đã thăm nhưng chưa duyệt xong.
- *Đen (Black)* – đã duyệt xong

## Tìm kiếm theo chiều rộng từ đỉnh s

---

### BFS\_Visit(s)

```
1. for  $u \in V - \{s\}$                                 10 while  $Q \neq \emptyset$ 
2      do  $color[u] \leftarrow \text{white}$                 11   do  $u \leftarrow \text{dequeue}(Q)$ 
3       $d[u] \leftarrow \infty$                             12     for  $v \in \text{Adj}[u]$ 
4       $\pi[u] \leftarrow \text{NULL}$                          13       do if  $color[v] = \text{white}$ 
5   $color[s] \leftarrow \text{gray}$                            14         then  $color[v] \leftarrow \text{gray}$ 
6   $d[s] \leftarrow 0$                                  15            $d[v] \leftarrow d[u] + 1$ 
7   $\pi[s] \leftarrow \text{NULL}$                           16            $\pi[v] \leftarrow u$ 
8   $Q \leftarrow \emptyset$                                 17            $\text{enqueue}(Q, v)$ 
9   $\text{enqueue}(Q, s)$                                 18            $color[u] \leftarrow \text{black}$ 
```

## Thuật toán tìm kiếm theo chiều rộng trên đồ thị G

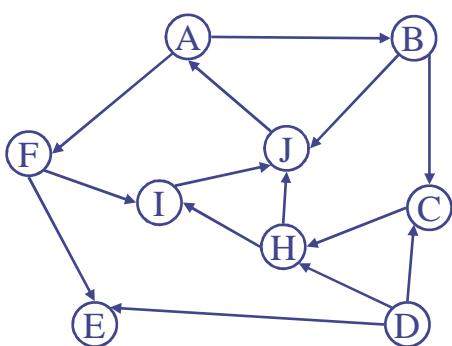
---

### BFS(G)

1. **for**  $u \in V$
2.     **do**  $\text{color}[u] \leftarrow \text{white}$
3.          $\pi[u] \leftarrow \text{NULL}$
5. **for**  $u \in V$
6.     **do if**  $\text{color}[u] = \text{white}$
7.         **then BFS-Visit( $u$ )**

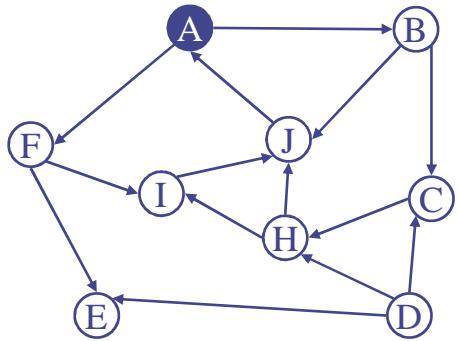
## Ví dụ: Thực hiện BFS(A)

---



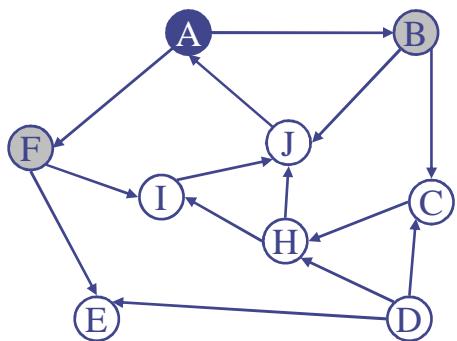
**Q = {A}**

---



**Q = {B,F}**

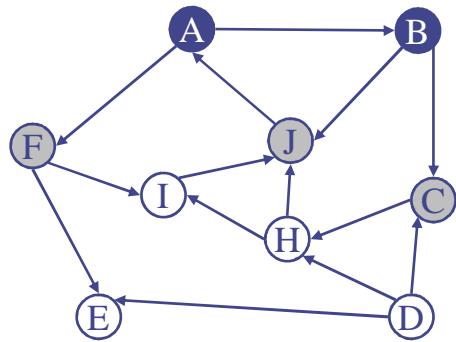
---



$$Q = \{F, C, J\}$$

---

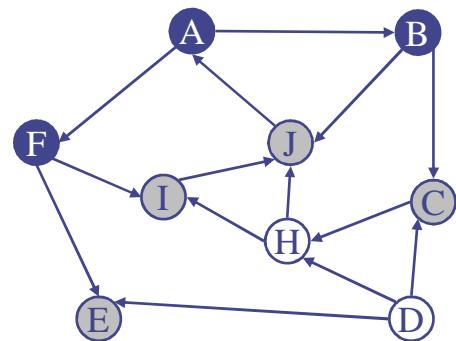
---



$$Q = \{C, J, E, I\}$$

---

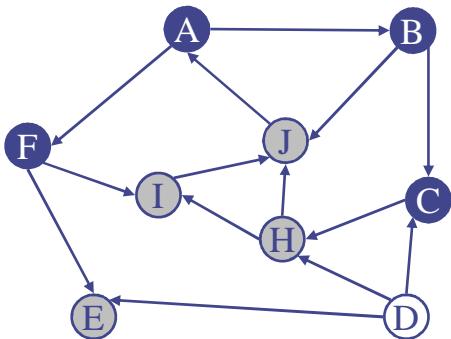
---



$$Q = \{J, E, I, H\}$$

---

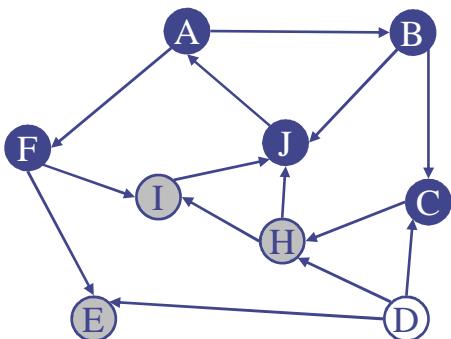
---



$$Q = \{E, I, H\}$$

---

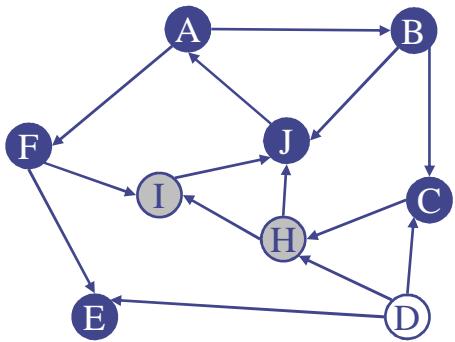
---



$$Q = \{I, H\}$$

---

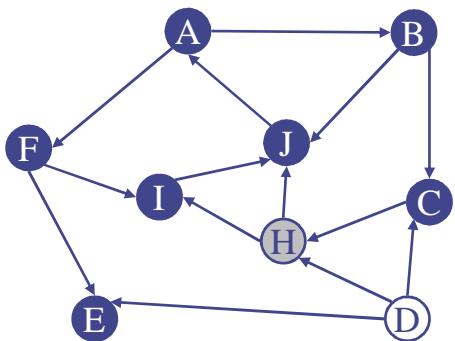
---



$$Q = \{H\}$$

---

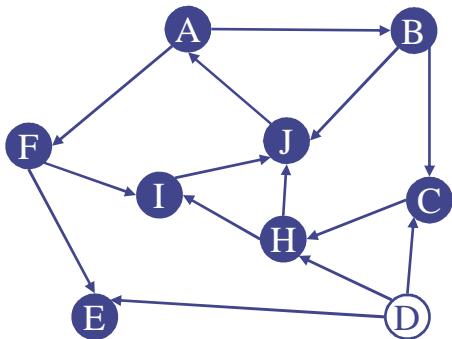
---



**Q = {}**

**Kết thúc BFS(A)**

---



## Tính đúng đắn của BFS

---

**Định lý:**

- BFS\_Visit( $s$ ) cho phép đến thăm tất cả các đỉnh  $v \in V$  đạt đến được từ  $s$ .
- Khi thuật toán kết thúc  $d[v]$  cho ta độ dài đường đi ngắn nhất (theo số cạnh) từ  $s$  đến  $v$ .
- Với mỗi đỉnh  $v$  đạt đến được từ  $s$ ,  $\pi[v]$  cho ta đỉnh đi trước đỉnh  $v$  trong đường đi ngắn nhất từ  $s$  đến  $v$ .

## Cây tìm kiếm theo chiều rộng (Breadth-first Tree)

---

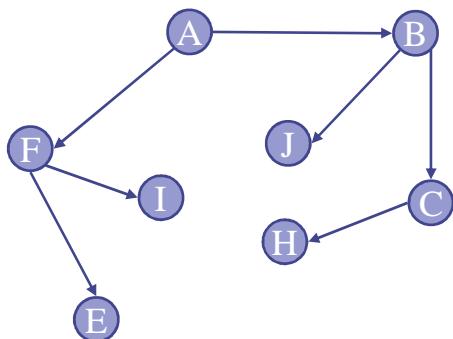
---

- ◆ Đối với đồ thị  $G = (V, E)$  với đỉnh xuất phát  $s$ , ký hiệu  $G_\pi = (V_\pi, E_\pi)$  là đồ thị với
  - $V_\pi = \{v \in V : \pi[v] \neq \text{NULL}\} \cup \{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$
- ◆ Đồ thị  $G_\pi$  được gọi là cây BFS( $s$ ):
  - $V_\pi$  chứa tất cả các đỉnh đạt đến được từ  $s$  và
  - với mọi  $v \in V_\pi$ , đường đi từ  $s$  đến  $v$  trên  $G_\pi$  là đường đi ngắn nhất từ  $s$  đến  $v$  trên  $G$ .
- ◆ Các cạnh trong  $E_\pi$  được gọi là các *cạnh của cây*.  
 $|E_\pi| = |V_\pi| - 1$ .

## Ví dụ: Cây BFS(A)

---

---



## Độ phức tạp của BFS

---

- ◆ Thuật toán loại bỏ mỗi đỉnh khỏi hàng đợi đúng 1 lần, do đó thao tác DeQueue thực hiện đúng  $|V|$  lần.
- ◆ Với mỗi đỉnh, thuật toán duyệt qua tất cả các đỉnh kề của nó và thời gian xử lý mỗi đỉnh kề như vậy là hằng số. Như vậy thời gian thực hiện câu lệnh **if** trong vòng lặp **while** là bằng hằng số nhân với số cạnh kề với đỉnh đang xét.
- ◆ Do đó tổng thời gian thực hiện việc duyệt qua tất cả các đỉnh là bằng một hằng số nhân với số cạnh  $|E|$ .
- ◆ **Thời gian tổng cộng:**  $O(|V|) + O(|E|) = O(|V|+|E|)$ , hay  $O(|V|^2)$

---

## Thuật toán tìm kiếm theo chiều sâu (DFS)

# Tìm kiếm theo chiều sâu

---

- ◆ **Input:**  $G = (V, E)$  - đồ thị vô hướng hoặc có hướng.
- ◆ **Output:** Với mỗi  $v \in V$ .
  - $d[v] =$  thời điểm bắt đầu thăm ( $v$  chuyển từ màu trắng sang xám)
  - $f[v] =$  thời điểm kết thúc thăm ( $v$  chuyển từ màu xám sang đen)
  - $\pi[v]$  : đỉnh từ đó ta đến thăm đỉnh  $v$ .
- ◆ **Rừng tìm kiếm theo chiều sâu** (gọi tắt là rừng DFS - Forest of depth-first trees):
  - $G_\pi = (V, E_\pi)$ ,
  - $E_\pi = \{(\pi[v], v) : v \in V \text{ và } \pi[v] \neq \text{null}\}$ .

## Thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh $u$

---

### DFS-Visit( $u$ )

```
color[u] ← GRAY
time ← time + 1
d[u] ← time
for  $v \in Adj[u]$ 
    do if color[v] = WHITE
        then  $\pi[v] \leftarrow u$ 
              DFS-Visit(v)
    color[u] ← BLACK
    f[u] ← time ← time + 1
```

## Thuật toán tìm kiếm theo chiều sâu trên đồ thị G

---

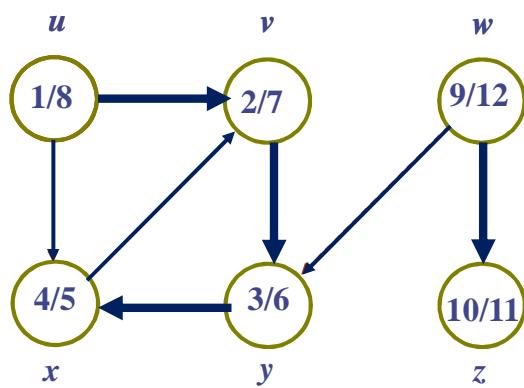
### DFS(G)

1. **for**  $u \in V[G]$
2.     **do**  $color[u] \leftarrow \text{white}$
3.          $\pi[u] \leftarrow \text{NULL}$
4.      $time \leftarrow 0$
5. **for**  $u \in V[G]$
6.     **do if**  $color[u] = \text{white}$
7.         **then** DFS-Visit( $u$ )

## Ví dụ

---

- ◆ Thực hiện DFS trên đồ thị sau:

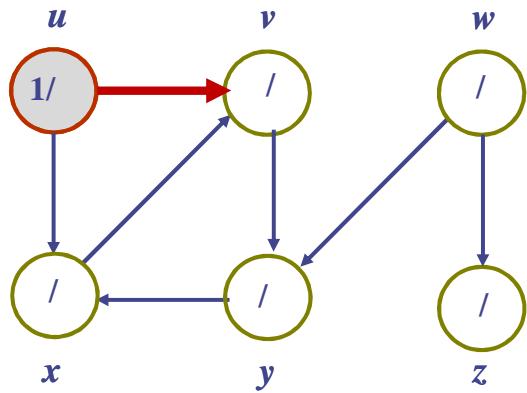


- ◆ DFS(G) sẽ gọi thực hiện DFS(u) và DFS(w).
- ◆ Cặp số viết trong các đỉnh v là  $d[v]/f[v]$ .
- ◆ Các cạnh đậm là các cạnh của rừng tìm kiếm.

## DFS( $u$ )

---

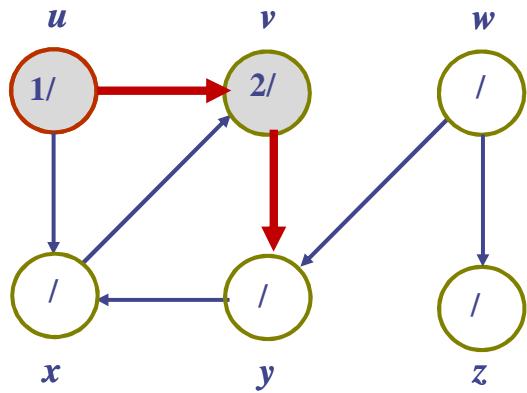
- ◆ Thăm đỉnh  $u$



## DFS( $v$ )

---

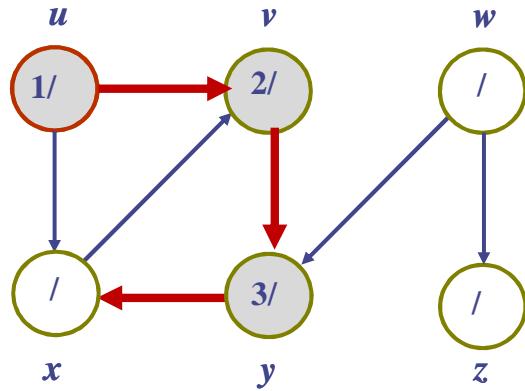
- ◆ Thăm đỉnh  $v$



## DFS(y)

---

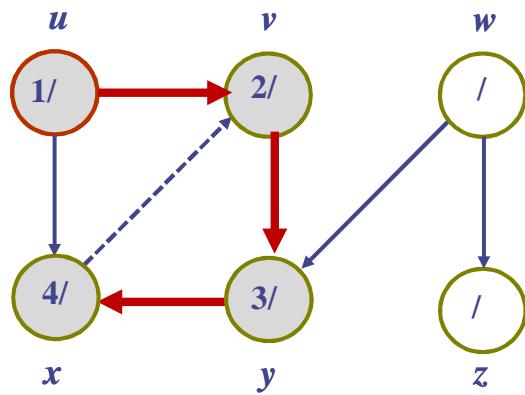
- ◆ Thăm đỉnh y



## DFS(x)

---

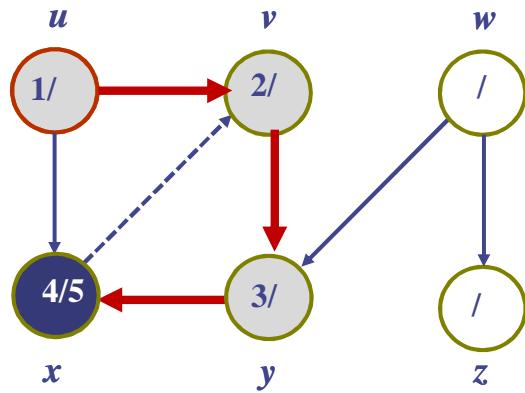
- ◆ Thăm đỉnh x



## DFS(x)

---

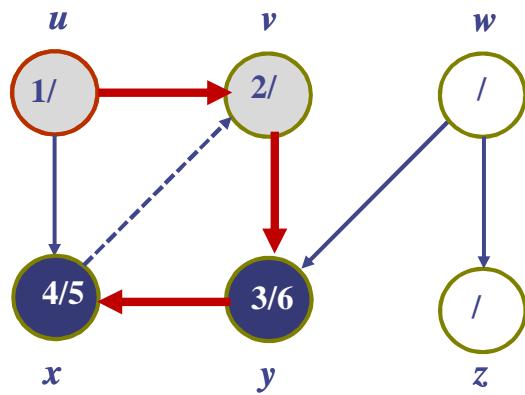
- ◆ Kết thúc thăm đǐnh x



## DFS(y)

---

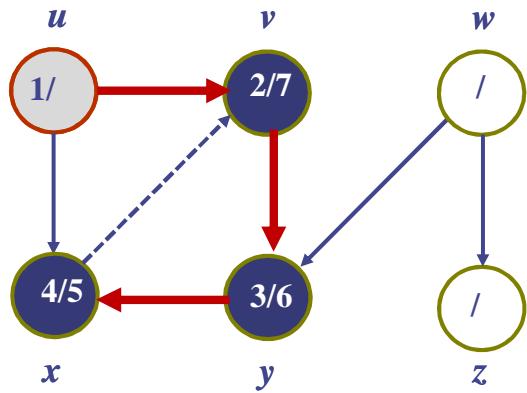
- ◆ Kết thúc thăm đǐnh y



## DFS(v)

---

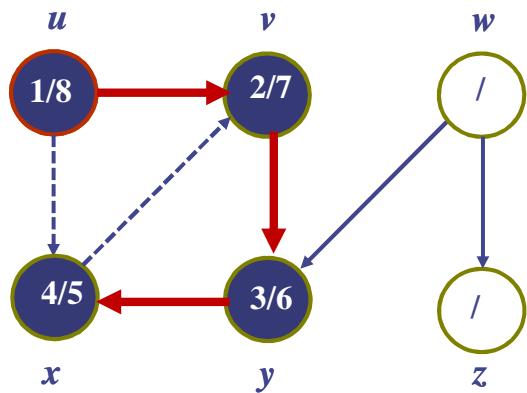
- ◆ Kết thúc thăm đǐnh v



## DFS(u)

---

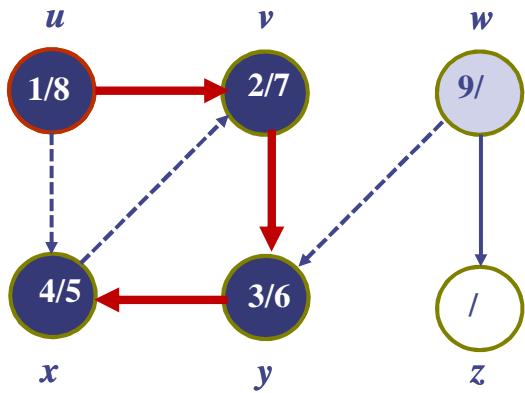
- ◆ Kết thúc thăm đǐnh u



## DFS(w)

---

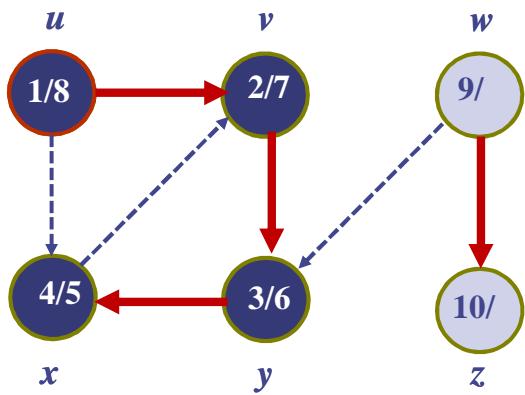
- ◆ Thăm đỉnh w



## DFS(z)

---

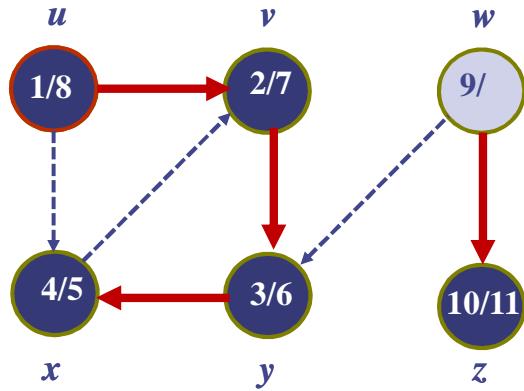
- ◆ Thăm đỉnh z



## DFS(z)

---

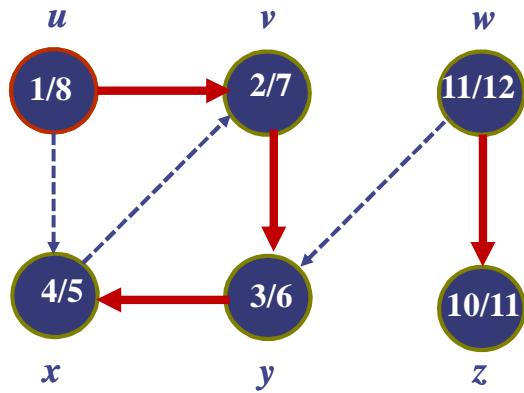
- ◆ Kết thúc thăm đǐnh z



## DFS(w)

---

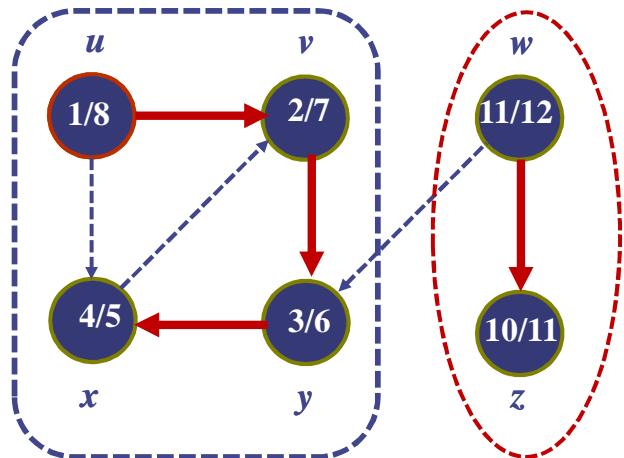
- ◆ Kết thúc thăm đǐnh w



## DFS(G): Kết thúc

---

- ◆ Rừng tìm kiếm gồm 2 cây: Cây DFS(u) và cây DFS(w):



## Các tính chất của DFS

---

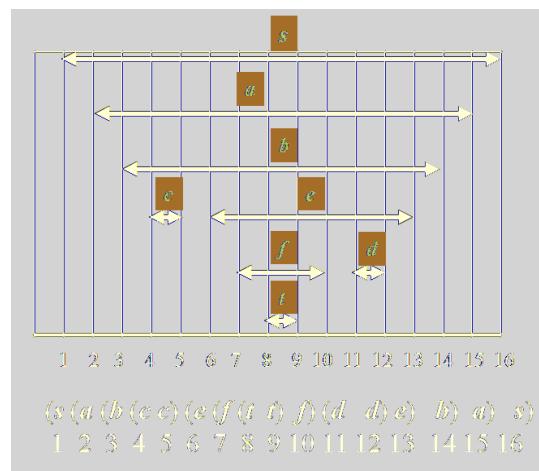
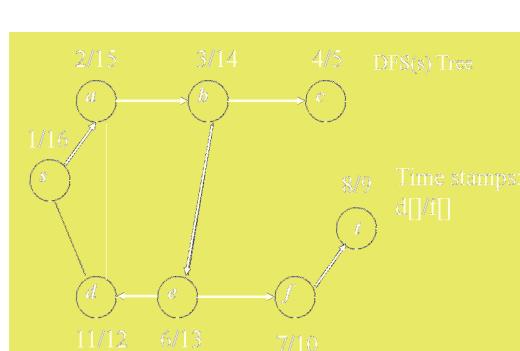
- ◆ Rừng DFS là phụ thuộc vào thứ tự các đỉnh được duyệt trong các vòng lặp **for** duyệt đỉnh trong  $\text{DFS}(G)$  và  $\text{DFS\_Visit}(u)$ .
- ◆ Để gõ đệ qui có thể sử dụng ngăn xếp. Có thể nói, điểm khác biệt cơ bản của DFS với BFS là các đỉnh đang được thăm trong DFS được cất giữ vào ngăn xếp thay vì hàng đợi trong BFS.
- ◆ Các khoảng thời gian thăm  $[d[v], f[v]]$  của các đỉnh có cấu trúc lồng nhau (*parenthesis structure*).

## Cấu trúc lồng nhau (parenthesis structure)

◆ **Định lý.** Với mọi  $u, v$ , chỉ có thể xảy ra một trong các tình huống sau:

1.  $d[u] < f[u] < d[v] < f[v]$  hoặc  $d[v] < f[v] < d[u] < f[u]$  (nghĩa là hai khoảng thời gian thăm của  $u$  và  $v$  là dời nhau) và khi đó  $u$  và  $v$  là không có quan hệ tổ tiên – hậu duệ.
2.  $d[u] < d[v] < f[v] < f[u]$  (nghĩa là khoảng thời gian thăm của  $v$  là lồng trong khoảng thời gian thăm của  $u$ ) và khi đó  $v$  là hậu duệ của  $u$ .
3.  $d[v] < d[u] < f[u] < f[v]$  (nghĩa là khoảng thời gian thăm của  $u$  là lồng trong khoảng thời gian thăm của  $v$ ) và khi đó  $u$  là hậu duệ của  $v$ .

## Ví dụ



# Độ phức tạp của DFS

---

- ◆ Thuật toán thăm mỗi đỉnh  $v \in V$  đúng một lần, việc thăm đỉnh đòi hỏi thời gian  $\Theta(|V|)$
- ◆ Với mỗi đỉnh  $v$  duyệt qua tất cả các đỉnh kề, với mỗi đỉnh kề thực hiện thao tác với thời gian hằng số. Do đó việc duyệt qua tất cả các đỉnh mất thời gian:

$$\sum_{v \in V} |\text{neighbors}[v]| = \Theta(|E|)$$

- ◆ Tổng cộng:  $\Theta(|V|) + \Theta(|E|) = \Theta(|V|+|E|)$ , hay  $\Theta(|V|^2)$
- ◆ Như vậy, DFS có cùng độ phức tạp như BFS.

## Phân loại cạnh

---

- ◆ DFS tạo ra một cách phân loại các cạnh của đồ thị đã cho:
  - **Cạnh của cây (Tree edge)**: là cạnh mà theo đó từ một đỉnh ta đến thăm một đỉnh mới
  - **Cạnh ngược (Back edge)**: đi từ con cháu (descendent) đến tổ tiên (ancestor)
  - **Cạnh tới (Forward edge)**: đi từ tổ tiên đến hậu duệ
  - **Cạnh vòng (Cross edge)**: cạnh nối hai đỉnh không có quan hệ họ hàng.

# Nhận biết các loại cạnh

---

- ◆ Để nhận biết cạnh  $(u, v)$  thuộc loại cạnh nào, ta dựa vào màu của đỉnh  $v$  khi lần đầu tiên cạnh  $(u, v)$  được khảo sát. Cụ thể, nếu màu của đỉnh  $v$  là
- Trắng, thì  $(u, v)$  là cạnh của cây;
  - Xám, thì  $(u, v)$  là cạnh ngược;
  - Đen, thì  $(u, v)$  là cạnh tới hoặc vòng. Trong trường hợp này để phân biệt cạnh tới và cạnh vòng ta cần xét xem hai đỉnh  $u$  và  $v$  có quan hệ họ hàng hay không nhờ sử dụng kết quả của định lý về cấu trúc lồng nhau.

# DFS trên đồ thị vô hướng

---

- ◆ **Định lý.** Nếu  $G$  là đồ thị vô hướng, thì DFS chỉ sản sinh ra cạnh của cây và cạnh ngược.
- ◆ **Chứng minh.**
- Giả sử  $(u, v) \in E$ . Không giả thiết giả sử  $d[u] < d[v]$ . Khi đó  $v$  phải trở thành đã duyệt xong trước khi  $u$  trở thành đã duyệt xong.
  - Nếu  $(u, v)$  được khảo sát lần đầu tiên theo hướng  $u \rightarrow v$ , thì trước thời điểm khảo sát  $v$  phải có màu trắng, và do đó  $(u, v)$  là cạnh của cây.
  - Nếu  $(u, v)$  được khảo sát lần đầu tiên theo hướng  $v \rightarrow u$ ,  $u$  phải có màu xám tại thời điểm khảo sát cạnh này và do đó nó là cạnh ngược.

---

---

## 4. Một số ứng dụng của tìm kiếm trên đồ thị

---

---

### Các ứng dụng của DFS và BFS

- 
- 
- ◆ Các thuật toán tìm kiếm trên đồ thị BFS và DFS được ứng dụng để giải nhiều bài toán trên đồ thị, chẳng hạn như
    - Tìm đường đi giữa hai đỉnh s và t của đồ thị;
    - Kiểm tra tính liên thông, liên thông mạnh của đồ thị;
    - Xác định các thành phần liên thông, song liên thông, liên thông mạnh;
    - Tính hai phía của đồ thị;
    - Tính phẳng của đồ thị.
    - ...

## Bài toán đường đi

---

- ◆ **Bài toán đặt ra là:** "Cho đồ thị  $G=(V,E)$  và hai đỉnh  $s, t$  của nó. Hỏi có tồn tại đường đi từ  $s$  đến  $t$  hay không? Trong trường hợp câu trả lời là khẳng định cần đưa ra một đường đi từ  $s$  đến  $t$ ."
- ◆ Để giải bài toán này ta có thể thực hiện DFS\_Visit( $s$ ) hoặc BFS\_Visit( $s$ ). Kết thúc, nếu đỉnh  $t$  là được thăm thì câu trả lời là khẳng định và khi đó để đưa ra đường đi từ  $s$  đến  $t$  ta sử dụng biến ghi nhận  $\pi[v]$ :

$$t \leftarrow \pi[t] \leftarrow \pi[\pi[t]] \leftarrow \dots \leftarrow s.$$

- ◆ Nếu  $t$  không được thăm, ta khẳng định là không có đường đi cần tìm.
- ◆ **Chú ý:** Đường đi tìm được từ  $s$  đến  $t$  theo BFS\_Visit( $s$ ) là đường đi ngắn nhất (theo số cạnh).

## Bài toán liên thông

---

- ◆ **Bài toán liên thông.** Cho đồ thị vô hướng  $G=(V,E)$ . Hãy kiểm tra xem đồ thị  $G$  có phải liên thông hay không. Nếu  $G$  không là liên thông, cần đưa ra số lượng thành phần liên thông và danh sách các đỉnh của từng thành phần liên thông.
- ◆ Để giải bài toán này, ta chỉ việc thực hiện DFS( $G$ ) (hoặc BFS( $G$ )). Khi đó số lần gọi thực hiện BFS\_Visit() (DFS\_Visit()) sẽ chính là số lượng thành phần liên thông của đồ thị. Việc đưa ra danh sách các đỉnh của từng thành phần liên thông sẽ đòi hỏi phải đưa thêm vào biến ghi nhận xem mỗi đỉnh được thăm ở lần gọi nào trong BFS( $G$ ) (DFS( $G$ )).

## Bài toán liên thông mạnh

---

- ◆ **Bài toán liên thông mạnh.** Cho đồ thị có hướng  $G=(V,E)$ . Hãy kiểm tra xem đồ thị  $G$  có phải liên thông mạnh hay không?
- ◆ Kết quả sau đây cho phép qui dẫn bài toán cần giải về bài toán đường đi.
- ◆ **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  $v$ .
- ◆ **Chứng minh.** Suy trực tiếp từ định nghĩa đồ thị có hướng liên thông mạnh.

## 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.
- ◆ Dễ 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).

## 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  $G^T = (V, E^T)$ . Nếu tồn tại đỉnh  $u$  không được thăm thì  $G$  không liên thông mạnh, nếu trái lại  $G$  là liên thông mạnh.

## Đồ thị không chứa chu trình

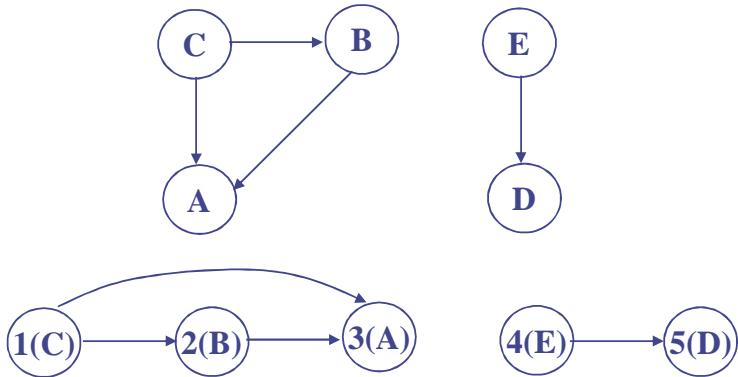
---

- **Bài toán:** Cho đồ thị  $G=(V,E)$ . Hỏi  $G$  có chứa chu trình hay không
- **Mệnh đề.** Đồ thị  $G$  là không chứa chu trình khi và chỉ khi DFS thực hiện đối với  $G$  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  $G$  là á 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, và  $u$  là đỉnh đi trước  $v$  trên chu trình. Khi  $v$  được thăm, các đỉnh khác trên chu trình đều là đỉnh trắng. Ta phải thăm được tất cả các đỉnh đạt được từ  $v$  trước khi quay trở lại từ DFS-Visit(). Vì thế cạnh  $u \rightarrow v$  được duyệt từ đỉnh  $u$  về tổ tiên  $v$  của nó, vì thế  $(u, v)$  là cạnh ngược.

## Bài toán sắp xếp tôpô (Topological Sort)

- ◆ **Bài toán đặt ra là:** Cho đồ thị có hướng không có chu trình  $G = (V, E)$ . 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**



## Cơ sở thuật toán

- ◆ **Mệnh đề.** Nếu có cung  $(u, v)$  thì  $f[u] > f[v]$  trong DFS.

- ◆ **Chứng minh.**

- Khi cung  $(u, v)$  được khảo sát, thì  $u$  có màu xám. Khi đó  $v$  có thể có một trong 3 màu: xám, trắng, đen.
- Nếu  $v$  có màu xám  $\Rightarrow (u, v)$  là cạnh ngược  $\Rightarrow$  Tồn tại chu trình?
- Nếu  $v$  có màu trắng  $\Rightarrow v$  trở thành con cháu của  $u \Rightarrow f[v] < f[u]$ .
- Nếu  $v$  có màu đen  $\Rightarrow v$  đã duyệt xong  $\Rightarrow f[v] < f[u]$ .

# Thuật toán sắp xếp tópô

---

Thuật toán có thể mô tả vắn tắt như sau:

- ◆ Thực hiện 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 DFS( $G$ ) sẽ cho ta thứ tự cần tìm.

# Thuật toán sắp xếp tópô

---

## TopoSort( $G$ )

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

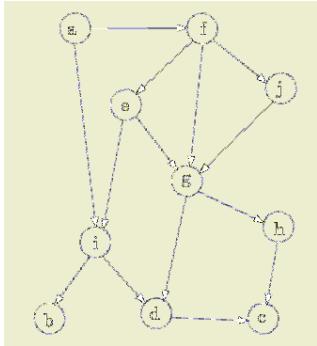
## TopVisit( $u$ ) { // Bắt đầu tìm kiếm từ $u$

```
1. color[ $u$ ] = gray; // Đánh dấu  $u$  là đã thăm
2. for  $v \in \text{Adj}(u)$ 
3.   if (color[ $v$ ] == white) TopVisit( $v$ );
4.   Nạp  $u$  vào đầu danh sách  $L$  //  $u$  đã duyệt xong
```

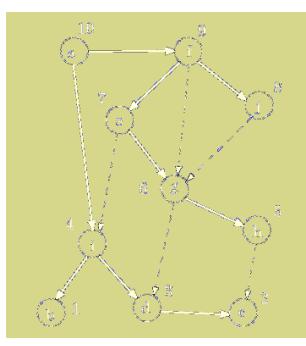
}

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

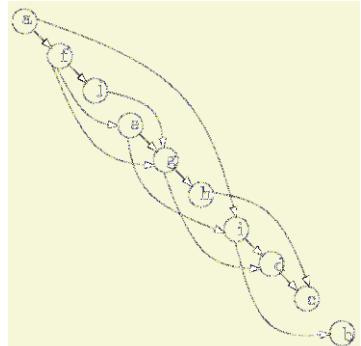
## Ví dụ



Đồ thị G



DFS(G)



Thứ tự tôpô

## 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.

# Thuật toán xoá dần đỉnh

---

◆ Từ mệnh đề ta suy ra thuật toán xoá dần đỉnh để thực hiện sắp xếp tópô 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.

# Thuật toán xoá dần đỉnh

---

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

    Tính  $\text{Degree}[v]$  – 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**

$\text{Degree}[u]=\text{Degree}[u] -1$ ;

**if**  $\text{Degree}[u]==0$

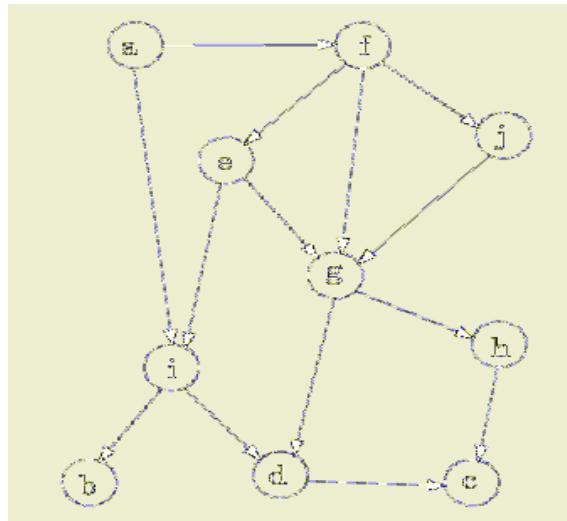
$\text{Enqueue}(Q, u)$ ;

Thời gian tính:  $\Theta(|V|+|E|)$

# Thuật toán xoá dần đỉnh

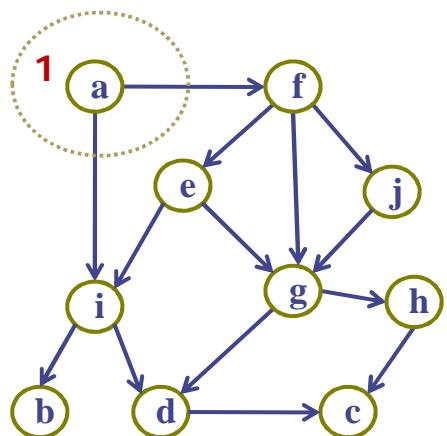
---

**Ví dụ.** Thực hiện thuật toán xoá dần đỉnh đối với đồ thị



# Thuật toán xoá dần đỉnh

---



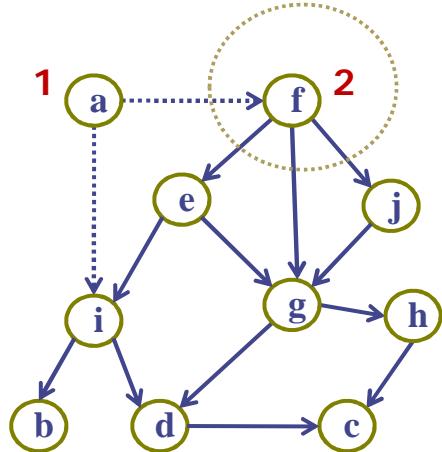
Đỉnh a có  $\text{deg}^-(a)=0$

Đánh số a bởi 1

Xoá các cung đi ra khỏi a

## Thuật toán xoá dần đỉnh

---



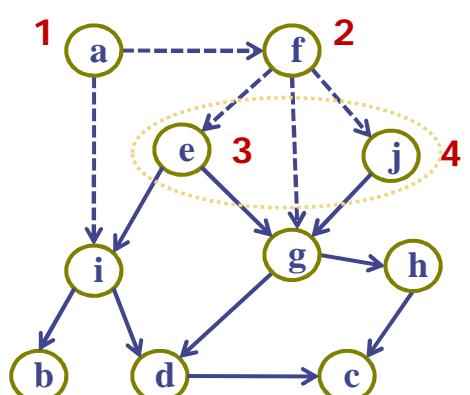
Đỉnh f có  $\text{deg}-(f)=0$

Đánh số f bởi 2

Xoá các cung đi ra khỏi f

## Thuật toán xoá dần đỉnh

---



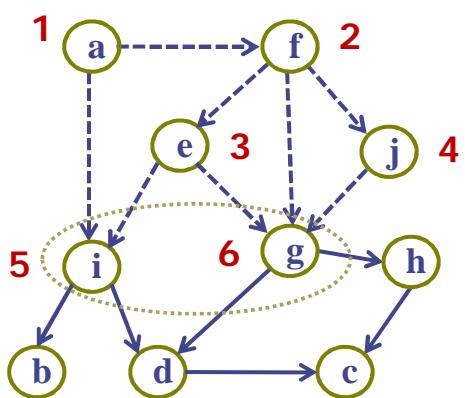
Đỉnh e và j có  $\text{deg}-(e) = \text{deg}-(j) = 0$

Đánh số e và j theo thứ tự tuỳ ý  
bởi 3 và 4

Xoá các cung đi ra khỏi e và j

# Thuật toán xoá dần đỉnh

---



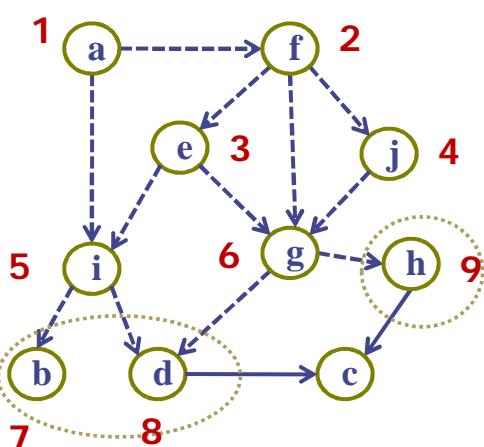
Đỉnh i và g có  $\text{deg}-(i) = \text{deg}-(g) = 0$

Đánh số i và g theo thứ tự tuỳ ý  
bởi 5 và 6

Xoá các cung đi ra khỏi i và g

# Thuật toán xoá dần đỉnh

---

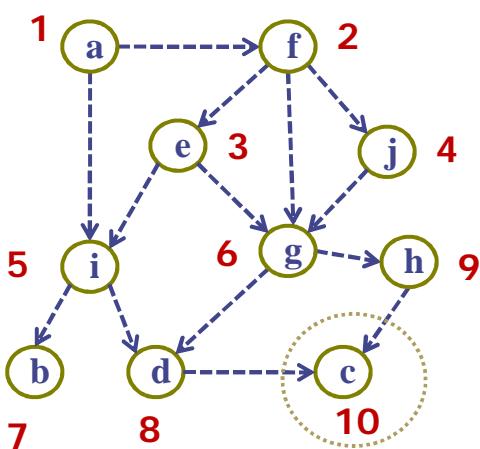


Đỉnh b, d và h có  $\text{deg}-(b) = \text{deg}-(d)$   
 $= \text{deg}(h) = 0$

Đánh số b, d và h theo thứ tự tuỳ ý  
bởi 7, 8 và 9

Xoá các cung đi ra khỏi b, d và h

# Thuật toán xoá dần đỉnh



Đỉnh c có  $\text{deg}-(c) = 0$

Đánh số c bởi 10

Thuật toán kết thúc

**BAO ĐÓNG TRUYỀN ỨNG**

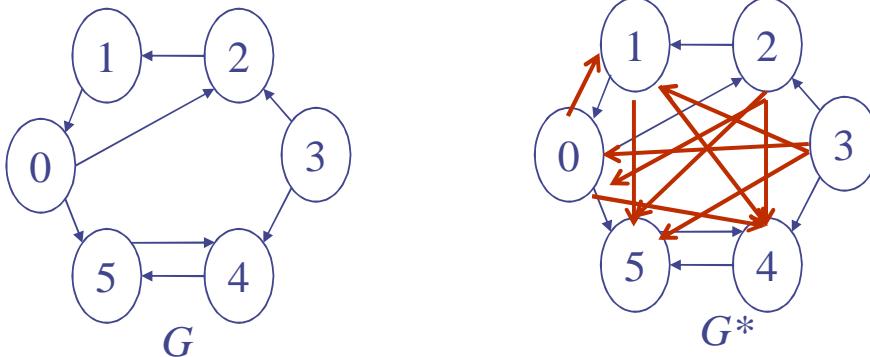
Transitive Closure

# 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

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

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



# 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;
```

- ◆ **Mệnh đề.** Thuật toán tìm được bao đóng truyền ứng với thời gian  $O(|V|^3)$ .
- ◆ **CM:** 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 bổ sung thêm câu lệnh **if** 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  $v \in V$  ( $|V|$  lần).
- 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

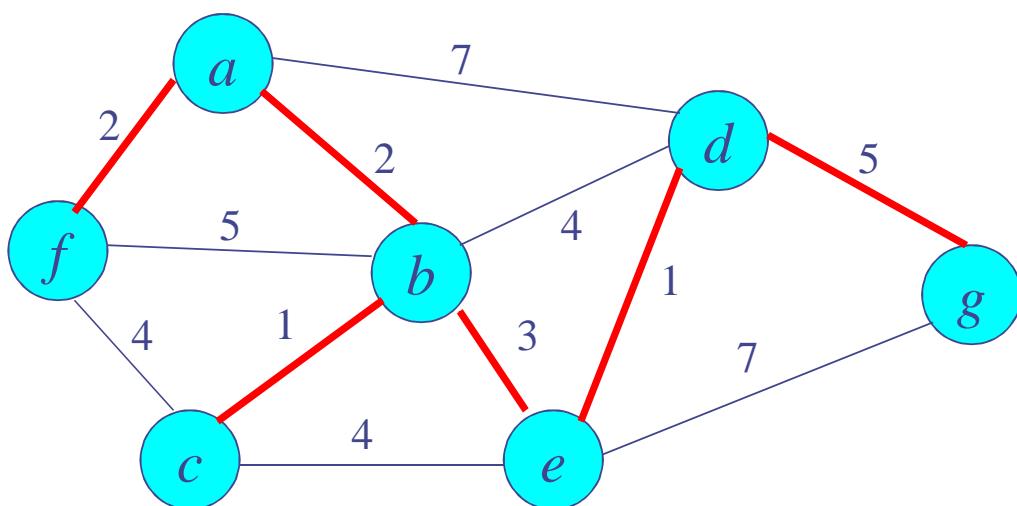
L        DFS sử dụng danh sách kề

## 5. Bài toán cây khung nhỏ nhất

# Phát biểu bài toán

- ◆ Giả sử  $G = (V, E)$  là đồ thị vô hướng liên thông có trọng số trên cạnh  $c(e)$ ,  $e \in E$ .
- ◆ **Định nghĩa.** Cây  $T = (V, E_T)$  với  $E_T \subseteq E$ , được gọi là cây khung của  $G$ . Độ dài của cây khung  $T$  là tổng trọng số trên các cạnh của nó:
- ◆ Bài toán đặt ra là tìm cây khung có độ dài nhỏ nhất.

## Ví dụ



- ◆ Cây khung nhỏ nhất có độ dài 14

# Thuật toán Kruskal

---

- ◆ Thuật toán sẽ xây dựng tập cạnh  $E_T$  của cây khung nhỏ nhất  $T = (V, E_T)$  theo từng bước.
- ◆ Trước hết sắp xếp các cạnh của đồ thị  $G$  theo thứ tự không giảm của độ dài.
- ◆ Bắt đầu từ tập  $E_T = \emptyset$ , ở mỗi bước ta sẽ lần lượt duyệt trong danh sách cạnh đã sắp xếp, từ cạnh có độ dài nhỏ đến cạnh có độ dài lớn hơn, để tìm ra cạnh mà việc bổ sung nó vào tập  $E_T$  không tạo thành chu trình trong tập này.
- ◆ Thuật toán sẽ kết thúc khi ta thu được tập  $E_T$  gồm  $n-1$  cạnh.

# Thuật toán Kruskal

---

## Kruskal\_Algorithm

```
ET := ∅;  
while |ET| < (n-1) and ( E ≠ ∅ ) do  
{  
    Chọn e là cạnh có độ dài nhỏ nhất trong E;  
    E := E \ {e};  
    if ( ET ∪ {e} không chứa chu trình ) then ET := ET ∪{e};  
}  
if ( |ET| < n-1 ) then Đồ thị không liên thông;
```

# Cài đặt thuật toán Kruskal

---

◆ Có 2 thao tác đòi hỏi nhiều tính toán nhất trong 1 bước lặp của thuật toán Kruskal:

- Chọn  $e$  là cạnh có độ dài nhỏ nhất trong  $E$ ;
- Kiểm tra xem tập cạnh  $E_T \cup \{e\}$  có chứa chu trình hay không?

## Chọn $e$ là cạnh có độ dài nhỏ nhất trong $E$

---

◆ Ta sẽ thực hiện trước việc sắp xếp các cạnh của đồ thị theo thứ tự không giảm của độ dài. Đối với đồ thị có  $m$  cạnh, bước này đòi hỏi thời gian  $O(m \log m)$ . Khi đó trong bước lặp việc chọn cạnh lớn nhất đòi hỏi thời gian  $O(1)$ .

◆ Tuy nhiên, để xây dựng cây khung nhỏ nhất với  $n-1$  cạnh, nói chung, thường ta chỉ phải xét  $p < m$  cạnh. Do đó thay vì sắp xếp toàn bộ dãy cạnh ta sẽ sử dụng heap-min:

- Để tạo đống đầu tiên ta mất thời gian  $O(m)$ ,
- Việc vun lại đống sau khi lấy ra phần tử nhỏ nhất ở gốc đòi hỏi thời gian  $O(\log m)$ .
- Suy ra thuật toán sẽ đòi hỏi thời gian  $O(m+p \log m)$  cho việc sắp xếp các cạnh. Trong việc giải các bài toán thực tế, số  $p$  thường nhỏ hơn rất nhiều so với  $m$ .

## Kiểm tra: Tập $E_T \cup \{e\}$ có chứa chu trình hay không?

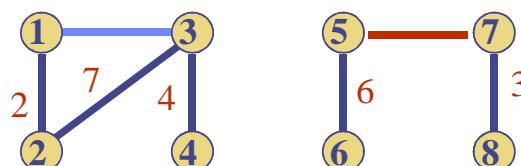
- ◆ Ký hiệu  $E^* = E_T \cup \{e\}$
- ◆ Việc này có thể thực hiện nhờ sử dụng thuật toán kiểm tra xem đồ thị  $G=(V,E^*)$  có chứa chu trình hay không đã trình bày trong mục trước. Thời gian cần thiết là  $O(n)$ , trong đó  $n = |V|$ .
- ◆ Với những đề xuất vừa nêu ta thu được cài đặt thuật toán Kruskal với thời gian

$$O(m+m \log m) + O(n.m) = O(nm + m \log m)$$

**Chú ý:** Có cách thực hiện khác dựa trên *cấu trúc dữ liệu các tập không giao nhau* để thực hiện thao tác kiểm tra tập  $E_T \cup \{e\}$  có chứa chu trình hay không?

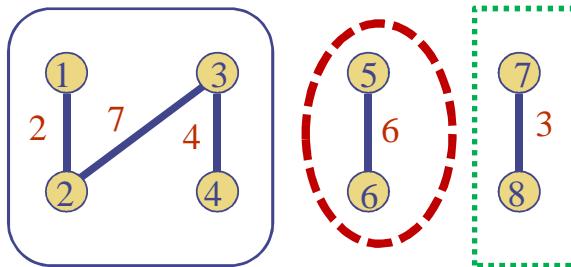
## Cấu trúc dữ liệu cho thuật toán Kruskal

- ◆ Bổ sung cạnh  $(u, v)$  vào  $E_T$  có tạo thành chu trình?



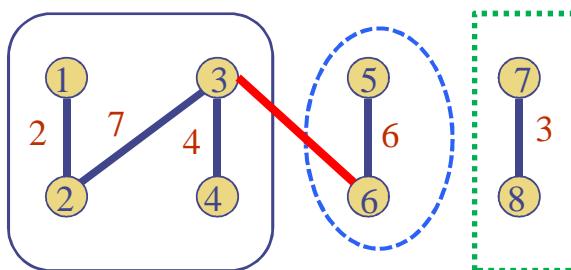
- Mỗi thành phần của  $T$  (đang xây dựng) là một cây.
- Khi  $u$  và  $v$  thuộc cùng một thành phần liên thông thì việc bổ sung  $(u,v)$  sẽ tạo thành chu trình.
- Khi  $u$  và  $v$  thuộc các thành phần liên thông khác nhau thì việc bổ sung  $(u,v)$  sẽ không tạo thành chu trình.

## Cấu trúc dữ liệu cho thuật toán Kruskal



- Mỗi tplt của  $T$  được xác định bởi các đỉnh trong nó.
- Biểu diễn mỗi tplt bởi tập các đỉnh thuộc nó.
  - $\{1, 2, 3, 4\}, \{5, 6\}, \{7, 8\}$
- Hai đỉnh thuộc cùng một tplt khi và chỉ khi chúng thuộc cùng một tập đỉnh.

## Cấu trúc dữ liệu cho thuật toán Kruskal



- Khi cạnh  $(u, v)$  được bổ sung vào  $T$ , hai thành phần chứa  $u$  và  $v$  sẽ được nối lại thành một tplt.
- Trong cách biểu diễn các tplt dưới dạng tập hợp, tập con chứa  $u$  và tập con chứa  $v$  sẽ được hợp lại thành một tập.
  - $\{1, 2, 3, 4\} + \{5, 6\} \Rightarrow \{1, 2, 3, 4, 5, 6\}$

## Cấu trúc dữ liệu cho thuật toán Kruskal

- Thoát tiên,  $E_T$  là rỗng. Có  $|V|$  tplt, mỗi thành phần gồm 1đỉnh.

(1)      (3)      (5)      (7)

(2)      (4)      (6)      (8)

- Các tập khởi tạo là:

- {1} {2} {3} {4} {5} {6} {7} {8}

- Nếu việc bổ sung cạnh  $(u, v)$  vào  $E_T$  không tạo thành chu trình thì cạnh này được bổ sung và  $E_T$ .

$r_1 = \text{find}(u); r_2 = \text{find}(v);$

**if** ( $r_1 \neq r_2$ ) **then**

$E_T = E_T \cup (u, v);$

$\text{union}(r_1, r_2);$

## Cấu trúc dữ liệu các tập không giao nhau (Disjoint-set Data Structures)

**Vấn đề đặt ra là:** Cho tập  $V$  gồm  $n$  phần tử, ta cần xây dựng cấu trúc dữ liệu biểu diễn phân hoạch tập  $V$  ra thành các tập con  $V_1, V_2, \dots, V_k$  hỗ trợ thực hiện hiệu quả các thao tác sau:

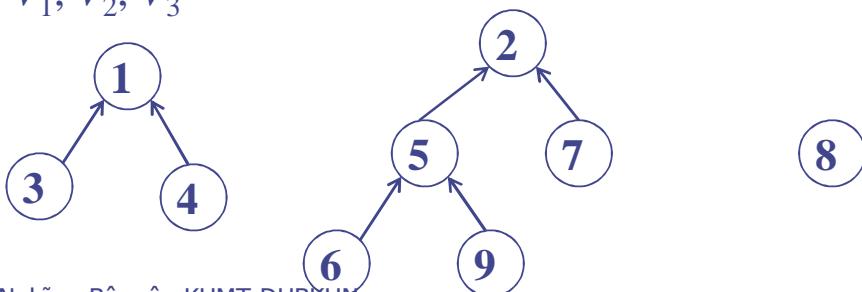
- **Makeset( $x$ ):** Tạo một tập con chứa duy nhất phần tử  $x$ .
- **Union( $x, y$ ):** Thay thế các tập  $V_i$  và  $V_j$  (trong đó  $x \in V_i$  và  $y \in V_j$ ) bởi tập  $V_i \cup V_j$  trong phân hoạch đang xét.
- **Find( $x$ ):** Tìm tên  $r(V_i)$  của tập  $V_i$  chứa phần tử  $x$ .

Như vậy,  $\text{Find}(x)$  và  $\text{Find}(y)$  trả lại cùng một giá trị khi và chỉ khi  $x$  và  $y$  thuộc cùng một tập con trong phân hoạch.

Cấu trúc dữ liệu đáp ứng yêu cầu này có tên là **cấu trúc dữ liệu Union-Find** (hoặc Disjoint-set data structure).

## Cấu trúc dữ liệu các tập không giao nhau (Disjoint-set Data Structures)

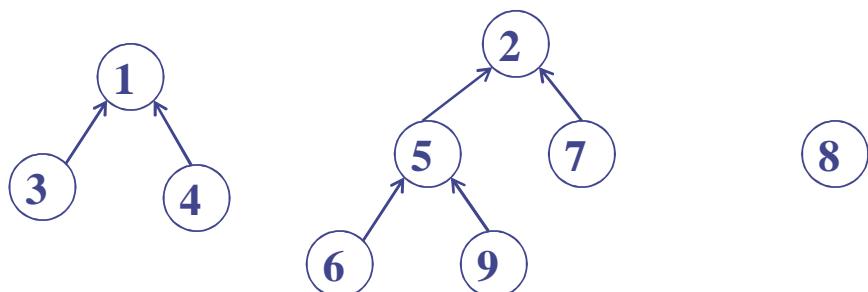
- ◆ Trước hết để biểu diễn mỗi tập con  $X \subset V$ , chúng ta sẽ sử dụng cấu trúc cây có gốc: Chọn một phần tử nào đó của  $X$  làm gốc (tên của tập con  $X$  chính là phần tử tương ứng với gốc), mỗi phần tử  $x \in X$  sẽ có một biến trỏ  $\text{parent}[x]$  trỏ đến cha của nó, nếu  $x$  là gốc thì  $\text{parent}[x] = x$ .
- ◆ **Ví dụ:** Giả sử có  $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ .  $V_1 = \{1, 3, 4\}$ ,  $V_2 = \{2, 5, 6, 7, 9\}$ ,  $V_3 = \{8\}$ . Ta có ba cây mô tả ba tập  $V_1, V_2, V_3$



Nguyễn Đức Nghĩa - Bộ môn KHMT ĐHBKHN

113

## Cấu trúc dữ liệu các tập không giao nhau (Disjoint-set Data Structures)



- ◆ Mảng parent để biểu diễn rừng gồm 3 cây tương ứng với  $V_1, V_2, V_3$ :

v	1	2	3	4	5	6	7	8	9
parent[v]	1	2	1	1	2	5	2	8	5

## Makeset(x) và FindSet

---

```
MakeSet(x) {  
    parent[x] := x;  
}
```

Thời gian:  $O(1)$ .

```
Find(x);  
{  
    while x ≠ parent[x] do x = parent[x];  
    return x;  
}
```

Thời gian:  $O(h)$ , trong đó  $h$  là độ cao của cây chứa  $x$ .

## Cấu trúc dữ liệu các tập không giao nhau (Disjoint-set Data Structures)

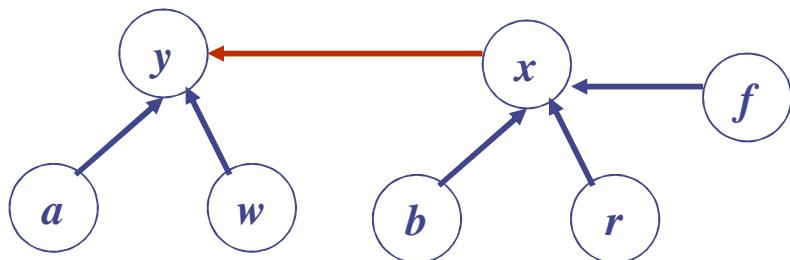
---

Để nối tập con chứa  $x$  và tập con chứa  $y$  chúng ta có thể  
chữa lại biến trả của gốc của cây chứa  $x$  để cho nó trả đến  
gốc của cây con chứa  $y$ . Điều đó được thực hiện nhờ thủ  
tục sau

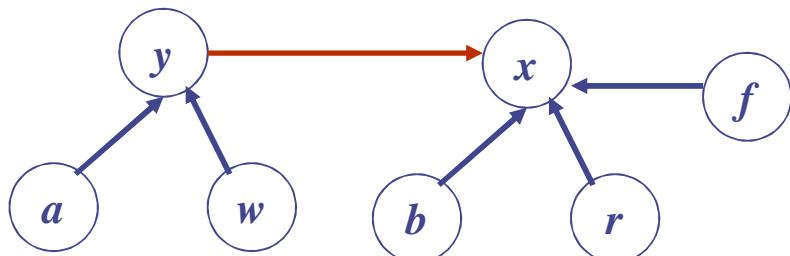
```
Union(x, y) {  
    u:= Find(x); (* Tìm u là gốc của cây con chứa x *)  
    v:= Find(y); (* Tìm v gốc của cây con chứa y *)  
    parent[u] := v;  
}
```

Thời gian:  $O(h)$

## Ví dụ Union(x,y)



$x$  trở đến  $y$   
 $b, r$  và  $f$   
chìm xuống sâu hơn



$y$  trở đến  $x$   
 $a$  và  $w$   
chìm xuống sâu hơn

## Phân tích độ phức tạp

- ◆ Có thể thấy thời gian tính của hàm Find(x) phụ thuộc vào độ cao của cây chứa  $x$ . Trong trường hợp cây có  $k$  đỉnh và có dạng như một đường đi thì độ cao của cây sẽ là  $k-1$ .

- ◆ **Ví dụ:**



- ◆ Sau khi thực hiện Union(A,B); Union(B,C); Union(C,D); Union(D,E) có thể thu được cây



- ◆ **Do đó hàm Find(x) có đánh giá thời gian tính là  $O(n)$ .**

## Cấu trúc dữ liệu các tập không giao nhau

---

- ◆ Liệu có cách nào để giảm độ cao của các cây con?
- ◆ Có một cách thực hiện rất đơn giản: Khi nối hai cây chúng ta sẽ điều chỉnh con trỏ của gốc của cây con có ít đỉnh hơn, chứ không thực hiện việc nối một cách tuỳ tiện.
- ◆ Để ghi nhận số phần tử của một cây chúng ta sẽ sử dụng thêm biến Num[v] chứa số phần tử của cây con với gốc tại v.

## MAKESET và Union cải tiến

---

**MAKESET(x) {**

```
parent[x] := x;  
Num[x]:=1;  
}
```

**Union(x, y){**

```
u:= Find(x); // Tìm u là gốc của cây con chứa x  
v:= Find(y); // Tìm u là gốc của cây con chứa y  
if Num[u] <= Num[v] {  
    parent[u] := v;  Num[v]:= Num[u]+Num[v];  
} else {  
    parent[v] := u; Num[u]:= Num[u]+Num[v];  
}  
}
```

## Cấu trúc dữ liệu các tập không giao nhau

---

◆ **Bổ đề.** Giả sử quá trình thực hiện nối cây bắt đầu từ các cây chỉ có 1 đỉnh. Khi đó độ cao của các cây xuất hiện khi thực hiện thủ tục nối không vượt quá  $\log n$ .

◆ **CM.** Qui nạp theo số đỉnh của cây.

◆ Từ bổ đề suy ra các thao tác Find và Union được thực hiện với thời gian  $O(\log n)$  nhờ sử dụng cách nối cây cải tiến.

## Thuật toán Kruskal sử dụng cấu trúc dữ liệu Union-Find

---

### Kruskal( $G, w$ )

1.  $E_T \leftarrow \emptyset$
2. **for**  $v \in V$  **do**
3.     Make-Set( $v$ )
4. Sắp xếp các cạnh trong  $E$  theo thứ tự không giảm của trọng số
5. **for**  $(u, v) \in E$  **do**
6.     **if**  $\text{Find}(u) \neq \text{Find}(v)$
7.         **then**  $E_T \leftarrow E_T \cup \{(u, v)\}$
8.             Union( $u, v$ )
9. **return**  $E_T$

## Phân tích thời gian tính Thuật toán Kruskal sử dụng cấu trúc dữ liệu Union-Find

---

---

- ◆ Dòng 1-3 (khởi tạo):  $O(|V|)$
- ◆ Dòng 4 (sắp xếp):  $O(|E| \log |E|)$
- ◆ Dòng 6-8 (các thao tác với phân hoạch):  $O(|E| \log |E|)$
- ◆ Tổng cộng:  **$O(|E| \log |E|)$**

## 6. Bài toán đường đi ngắn nhất

## Phát biểu bài toán

---

- ◆ **Định nghĩa.** Cho đồ thị có hướng  $G = (V, E)$  với trọng số trên cạnh  $c(e)$ ,  $e \in E$ . Giả sử  $s, t \in V$  và  $P(s, t)$  là đường đi từ  $s$  đến  $t$  trên đồ thị

$$P(s, t): \quad s = v_0, v_1, \dots, v_{k-1}, v_k = t.$$

- ◆ Ta gọi độ dài của đường đi  $P(s, t)$  là tổng trọng số trên các cung của nó, tức là nếu ký hiệu độ dài này là  $\rho(P(s, t))$ , thì theo định nghĩa

$$\rho(P(s, t)) = \sum_{e \in P(s, t)} c(e) = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$$

- ◆ Ta gọi đường đi ngắn nhất từ  $s$  đến  $t$  là đường đi có độ dài nhỏ nhất trong số tất cả các đường đi từ  $s$  đến  $t$  trên đồ thị.
- ◆ Người ta thường sử dụng ký hiệu  $\delta(s, t)$  để chỉ độ dài của đường đi ngắn nhất từ  $s$  đến  $t$ , và gọi nó là khoảng cách từ  $s$  đến  $t$ .

## Các dạng bài toán ĐĐNN

---

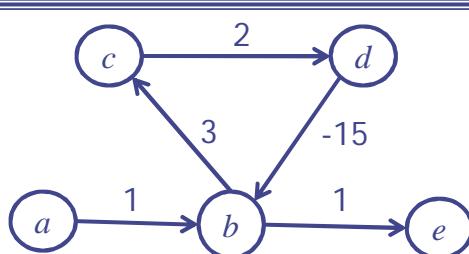
- ◆ Có 3 dạng bài toán đường đi ngắn nhất cơ bản
- Bài toán 1) Tìm đường đi ngắn nhất giữa 2 đỉnh cho trước.
  - Bài toán 2) Tìm đường đi ngắn nhất từ một đỉnh nguồn  $s$  đến tất cả các đỉnh còn lại.
  - Bài toán 3) Tìm đường đi ngắn nhất giữa hai đỉnh bất kì.
- ◆ Các bài toán được dẫn ra theo thứ tự từ đơn giản đến phức tạp hơn.
- ◆ Nếu ta có thuật toán để giải một trong ba bài toán thì thuật toán đó cũng có thể sử dụng để giải hai bài toán còn lại.

## Chu trình âm

◆ Đường đi ngắn nhất giữa hai đỉnh nào đó có thể không tồn tại.

- Chẳng hạn, nếu không có đường đi từ  $s$  đến  $t$ , thì rõ ràng cũng không có đường đi ngắn nhất từ  $s$  đến  $t$ .
- Ngoài ra, nếu đồ thị chứa cạnh có trọng số âm thì có thể xảy ra tình huống: độ dài đường đi giữa hai đỉnh nào đó có thể làm bé tùy ý.

## Chu trình âm



◆ Xét đường đi từ  $a$  đến  $e$ :

$$a, k(b, c, d, b), e.$$

nghĩa là ta đi  $k$  lần vòng theo chu trình

$$C: b, c, d, b$$

trước khi đến  $e$ . Độ dài của đường đi này là bằng:

$$c(a,b) + k \rho(C) + c(b,e) = 2 - 10k \rightarrow -\infty, \text{ khi } k \rightarrow +\infty.$$

# Thuật toán Dijkstra

---

- ◆ Thuật toán Dijkstra được đề xuất để giải bài toán: Tìm đường đi ngắn nhất từ một đỉnh nguồn  $s$  đến tất cả các đỉnh còn lại trên đồ thị với **trọng số không âm** trên cạnh.
- ◆ Trong quá trình thực hiện thuật toán, với mỗi đỉnh  $v$  ta sẽ lưu trữ nhãn của đỉnh gồm các thông tin sau:
  - $k[v]$ : biến bun có giá trị đúng nếu ta đã tìm được đường đi ngắn nhất từ  $s$  đến  $v$ , ban đầu biến này được khởi tạo giá trị false.
  - $d[v]$ : khoảng cách ngắn nhất hiện biết từ  $s$  đến  $v$ . Ban đầu biến này được khởi tạo giá trị  $+\infty$  đối với mọi đỉnh, ngoại trừ  $d[s]$  được đặt bằng 0.
  - $p[v]$ : là đỉnh đi trước đỉnh  $v$  trong đường đi có độ dài  $d[v]$ . Ban đầu, các biến này được khởi tạo rỗng (chưa biết).

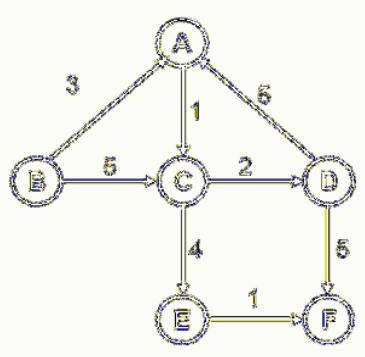
# Thuật toán Dijkstra

---

- ◆ Thuật toán lặp lại các thao tác sau đây cho đến khi tất cả các đỉnh được khảo sát xong (nghĩa là  $k[v] = \text{true}$  với mọi  $v$ ):
  - Trong tập đỉnh với  $k[v] = \text{false}$ , chọn đỉnh  $v$  có  $d[v]$  là nhỏ nhất.
  - Đặt  $k[v] = \text{true}$ .
  - Với mỗi đỉnh  $w$  kề với  $v$  và có  $k[w] = \text{false}$ , ta kiểm tra  $d[w] > d[v] + c(v, w)$ . Nếu đúng thì đặt lại  $d[w] = d[v] + c(v, w)$  và đặt  $p[w] = v$ .

## Ví dụ

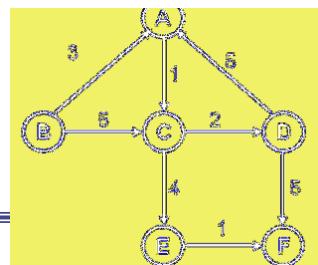
- ◆ Tìm đường đi ngắn nhất từ đỉnh B đến các đỉnh còn lại trên đồ thị sau đây



Nguyễn Đức Nghĩa - Bộ môn KHMT ĐHBKHN

131

## Bảng tính toán theo thuật toán Dijkstra



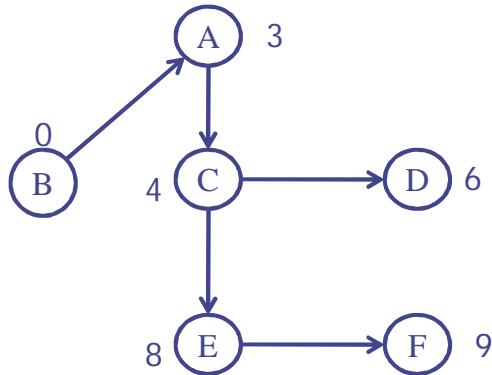
Bước lặp	A			B			C			D			E			F		
	d	p	k	d	p	k	d	p	k	d	p	k	d	p	k	d	p	k
Khởi tạo	$\infty$	-	F	0	-	F	$\infty$	-	F	$\infty$	-	F	$\infty$	-	F	$\infty$	-	F
1	3	B	F	0	-	T	5	B	F	$\infty$	-	F	$\infty$	-	F	$\infty$	-	F
2	3	B	T				4	A	F	$\infty$	-	F	$\infty$	-	F	$\infty$	-	F
3							4	A	T	6	C	F	8	C	F	$\infty$	-	F
4										6	C	T	8	C	F	11	D	F
5											8	C	T	11	D	F		
6												9	E	T				

Nguyễn Đức Nghĩa - Bộ môn KHMT ĐHBKHN

132

## Kết quả thực hiện

- ◆ Tập các cạnh  $\{(p[v], v) : v \in V - \{B\}\}$  tạo thành một cây được gọi là **cây đường đi ngắn nhất** từ đỉnh B đến tất cả các đỉnh còn lại. Cây này được cho trong hình vẽ sau đây:



## Cài đặt thuật toán với các cấu trúc dữ liệu

- ◆ Để cài đặt thuật toán Dijkstra chúng ta sử dụng bộ nhãn của các đỉnh:
- ◆ Nhãn của mỗi đỉnh v gồm 3 thành phần cho biết các thông tin:
- $k[v]$  - đã tìm được đường đi ngắn nhất từ đỉnh nguồn đến v hay chưa,
  - $d[v]$  - khoảng cách (độ dài đường đi) từ s đến v hiện biết
  - $p[v]$  - đỉnh đi trước đỉnh v trong đường đi tốt nhất hiện biết.
- ◆ Các thành phần này sẽ được cất giữ tương ứng trong các biến  $k[v]$ ,  $d[v]$  và  $p[v]$ .

# Cài đặt trực tiếp

---

```
Dijkstra_Table(G, s)
1. for u ∈ V do {
2.   d[u] ← infinity;
3.   p[u] ← NIL;
4.   k[u] ← FALSE;
5. }
6. d[s] ← 0;
// s là đỉnh nguồn
7. T = V;

8. while T ≠ ∅ do {
9.   u ← đỉnh có d[u] là nhỏ nhất trong T;
10.  k[u]=TRUE;
11.  T = T-{u};
12.  for (v ∈ Adj(u)) && !k[v] do
13.    if d[v] > d[u] + c[u, v] {
14.      d[v] = d[u] + c[u, v];
15.      p[v] = u;
16.    }
17. }
```

Dễ dàng nhận thấy rằng **Dijkstra\_Table(G, s)** đòi hỏi thời gian  $O(|V|^2+|E|)$ .

## Cài đặt thuật toán Dijkstra sử dụng hàng đợi có ưu tiên

---

- ◆ Do tại mỗi bước ta cần tìm ra đỉnh với nhãn khoảng cách nhỏ nhất, nên để thực hiện thao tác này một cách hiệu quả ta sẽ sử dụng hàng đợi có ưu tiên (Priority Queue – PQ).
- ◆ Dưới đây ta mô tả thuật toán Dijkstra với hàng đợi có ưu tiên:

## Cài đặt thuật toán Dijkstra sử dụng PQ: Khởi tạo

---

### Dijkstra\_Heap(G, s)

```
1. for u ∈ V do {  
2.   d[u] ← infinity;  
3.   p[u] ← NIL;  
4.   k[u] ← FALSE;  
5. }  
6. d[s] ← 0; // s là đỉnh nguồn  
7. Q ← Build_Min_Heap(d[V]);  
// Khởi tạo hàng đợi có ưu tiên Q từ d[V] = (d[v], v∈V)
```

## Cài đặt thuật toán Dijkstra sử dụng PQ: Lặp

---

```
8. while Not Empty(Q) do {  
9.   u ← Extract-Min(Q); // loại bỏ gốc của Q và đưa vào u  
10.  k[u]=TRUE;  
11  for (v ∈ Adj(u)) && !k[v] do  
12    if d[v] > d[u] + c[u, v] {  
13      d[v] = d[u] + c[u, v];  
14      p[v] = u;  
15      Decrease_Key(Q,v,d[v]);  
16    }  
17. }
```

## Phân tích thời gian tính của thuật toán

---

- ◆ Vòng lặp for ở dòng 1 đòi hỏi thời gian  $O(|V|)$ .
- ◆ Việc khởi tạo đống min đòi hỏi thời gian  $O(|V|)$ .
- ◆ Vòng lặp while ở dòng 8 lặp  $|V|$  lần do đó thao tác Extract-Min thực hiện  $|V|$  lần và đòi hỏi thời gian  $O(|V| \log|V|)$ .
- ◆ Thao tác Decrease\_Key ở dòng 15 phải thực hiện không quá  $O(|E|)$  lần. Do đó thời gian thực hiện thao tác này trong thuật toán là  $O(|E| \log|V|)$ .
- ◆ Vậy tổng cộng thời gian tính của thuật toán là

$$O((|E| + |V|) \log|V|).$$

---

# Questions?

