

## Chương 21. Các khái niệm cơ bản

### 21.1. Đồ thị

Đồ thị là mô hình biểu diễn một tập các đối tượng và mối quan hệ hai ngôi giữa các đối tượng:

$$\begin{aligned} \text{Graph} &= \text{Objects} + \text{Connections} \\ G &= (V, E) \end{aligned}$$

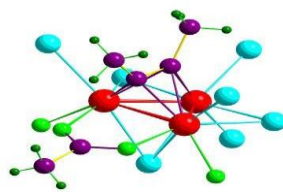
Có thể định nghĩa đồ thị  $G$  là một cặp  $(V, E)$ :  $G = (V, E)$ . Trong đó  $V$  là tập các đỉnh (vertices) biểu diễn các đối tượng và  $E$  gọi là tập các cạnh (edges) biểu diễn mối quan hệ giữa các đối tượng. Chúng ta quan tâm tới mối quan hệ hai ngôi (pairwise relations) giữa các đối tượng nên có thể coi  $E$  là tập các cặp  $(u, v)$  với  $u$  và  $v$  là hai đỉnh của  $V$  biểu diễn hai đối tượng có quan hệ với nhau.

Ta cho phép đồ thị có cả cạnh nối từ một đỉnh tới chính nó. Những cạnh như vậy gọi là *khuyên* (self-loop)

Một số hình ảnh của đồ thị:



Sơ đồ giao thông



Cấu trúc phân tử



Mạng máy tính

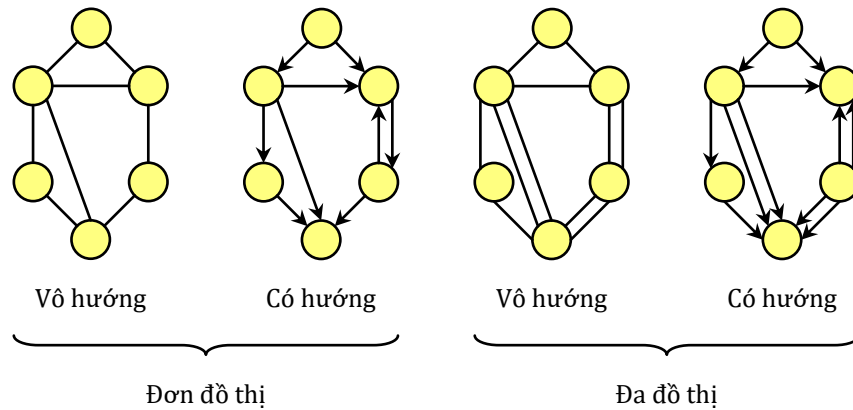
Hình 21-1. Một số hình ảnh của đồ thị

Có thể phân loại đồ thị  $G = (V, E)$  theo đặc tính và số lượng của tập các cạnh  $E$ :

- ✳  $G$  được gọi là *đồ thị vô hướng* (undirected graph) nếu các cạnh trong  $E$  là không định hướng, tức là cạnh nối hai đỉnh  $u, v \in V$  bất kỳ cũng là cạnh nối hai đỉnh  $v, u$ . Hay nói cách khác, tập  $E$  gồm các cặp  $(u, v)$  không tính thứ tự:  $(u, v) = (v, u)$ .
- ✳  $G$  được gọi là *đồ thị có hướng* (directed graph) nếu các cạnh trong  $E$  là có định hướng, tức là có thể có cạnh nối từ đỉnh  $u$  tới đỉnh  $v$  nhưng chưa chắc đã có cạnh nối từ đỉnh  $v$  tới đỉnh  $u$ . Hay nói cách khác, tập  $E$  gồm các cặp  $(u, v)$  có tính thứ tự:  $(u, v) \neq (v, u)$ . Trong đồ thị có hướng, các cạnh còn được gọi là các *cung* (arcs). Đối với một số bài toán, đồ thị vô hướng cũng có thể coi là đồ thị có hướng nếu như ta coi cạnh nối hai đỉnh  $u, v$  bất kỳ tương đương với hai cung  $(u, v)$  và  $(v, u)$ .
- ✳  $G$  được gọi là *đơn đồ thị* nếu giữa hai đỉnh  $u, v \in V$  có nhiều nhất là 1 cạnh trong  $E$  nối từ  $u$  tới  $v$ .

- ✿  $G$  được gọi là **đa đồ thị** (*multigraph*) nếu giữa hai đỉnh  $u, v \in V$  có thể có nhiều hơn 1 cạnh trong  $E$  nối từ  $u$  tới  $v$  (Hiển nhiên đơn đồ thị cũng là đa đồ thị). Nếu có nhiều cạnh nối giữa hai đỉnh  $u, v \in V$  thì những cạnh đó được gọi là *cạnh song song* (*parallel edges*)

Hình 21-2 là ví dụ về đơn đồ thị/đa đồ thị có hướng/vô hướng.



Hình 21-2. Phân loại đồ thị

## 21.2. Các khái niệm

### 21.2.1. Cạnh liên thuộc, đỉnh kề, bậc

Đối với đồ thị vô hướng  $G = (V, E)$ . Xét một cạnh  $e \in E$ , nếu  $e = (u, v)$  thì ta nói hai đỉnh  $u$  và  $v$  là *kề nhau* (*adjacent*) và cạnh  $e$  này *liên thuộc* (*incident*) với đỉnh  $u$  và đỉnh  $v$ .

Với một đỉnh  $u$  trong đồ thị vô hướng, ta định nghĩa *bậc* (*degree*) của  $u$ , ký hiệu  $\deg(u)$ , là số cạnh liên thuộc với  $u$ .

Chú ý rằng trên đơn đồ thị thì  $\deg(u)$  cũng là số đỉnh kề với  $u$ , ngoài ra nếu đồ thị có khuyên thì để hợp lý hóa các công thức trong bài, khuyên trên đồ thị vô hướng sẽ được tính hai lần.

#### Định lý 21-1

Trên đồ thị vô hướng, tổng bậc của tất cả các đỉnh bằng hai lần số cạnh:

$$G = (V, E) \Rightarrow \sum_{\forall u \in V} \deg(u) = 2|E| \quad (21.1)$$

#### Chứng minh

Khi lấy tổng tất cả các bậc đỉnh tức là mỗi cạnh  $e = (u, v)$  sẽ được tính một lần trong  $\deg(u)$  và một lần trong  $\deg(v)$ . Từ đó suy ra kết quả.

#### Hệ quả

Trong đồ thị vô hướng, số đỉnh bậc lẻ là số chẵn.

Đối với đồ thị có hướng  $G = (V, E)$ . Xét một cung  $e \in E$ , nếu  $e = (u, v)$  thì ta nói  $u$  nối tới  $v$  và  $v$  nối từ  $u$ , cung  $e$  là đi ra khỏi đỉnh  $u$  và đi vào đỉnh  $v$ . Đỉnh  $u$  khi đó được gọi là đỉnh đầu, đỉnh  $v$  được gọi là đỉnh cuối của cung  $e$ .

Với mỗi đỉnh  $v$  trong đồ thị có hướng, ta định nghĩa: *Bậc ra (out-degree)* của  $v$  ký hiệu  $\deg^+(v)$  là số cung đi ra khỏi nó; *Bậc vào (in-degree)* ký hiệu  $\deg^-(v)$  là số cung đi vào đỉnh đó.

### **Định lý 21-2**

Trên đồ thị có hướng, tổng bậc ra của tất cả các đỉnh bằng tổng bậc vào của tất cả các đỉnh và bằng số cung:

$$G = (V, E) \Rightarrow \sum_{\forall u \in V} \deg^+(u) = \sum_{\forall u \in V} \deg^-(u) = |E| \quad (21.2)$$

### **Chứng minh**

Khi lấy tổng tất cả các bán bậc ra hay bán bậc vào, mỗi cung  $(u, v)$  sẽ được tính đúng một lần trong  $\deg^+(u)$  và cũng được tính đúng một lần trong  $\deg^-(v)$ . Từ đó suy ra kết quả.

#### **21.2.2. Đường đi và chu trình**

Một dãy các đỉnh:

$$P = \langle p_0, p_1, \dots, p_k \rangle$$

sao cho  $(p_{i-1}, p_i) \in E, \forall i: 1 \leq i \leq k$  được gọi là một *đường đi (walk)*, đường đi này xuất phát từ đỉnh  $p_0$ , qua cạnh  $(p_0, p_1)$  sang đỉnh  $p_1$ , qua tiếp cạnh  $(p_1, p_2)$  sang đỉnh  $p_2, \dots$ , cuối cùng qua cạnh  $(p_{k-1}, p_k)$  sang đỉnh  $p_k$ . Nếu có một đường đi như trên thì ta nói  $p_k$  *đến được (reachable)* từ  $p_0$  hay  $p_0$  đến được  $p_k$ , ký hiệu  $p_0 \rightsquigarrow p_k$ . Đỉnh  $p_0$  được gọi là đỉnh đầu, đỉnh  $p_k$  gọi là đỉnh cuối, còn các đỉnh  $p_1, p_2, \dots, p_{k-1}$  được gọi là những *đỉnh trong (internal vertices)* của đường đi  $P$ . Đường đi có đỉnh đầu trùng với đỉnh cuối được gọi là một *chu trình (closed walk)*.

Một đường đi gọi là *đơn giản* hay *đường đi đơn (simple walk/path)* nếu tất cả các đỉnh trên đường đi là hoàn toàn phân biệt (dĩ nhiên khi đó các cạnh trên đường đi cũng hoàn toàn phân biệt). Một chu trình được gọi là chu trình đơn (*cycle*) nếu đỉnh đầu (cũng là đỉnh cuối) và các đỉnh trong hoàn toàn phân biệt\*.

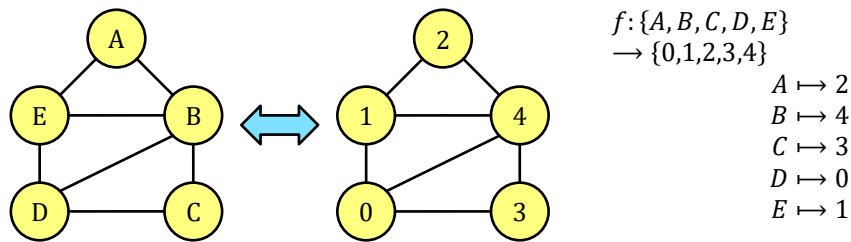
---

\* Những định nghĩa về thuật ngữ này không thống nhất trong các tài liệu về thuật toán, mỗi tác giả thường đưa ra định nghĩa của mình và dùng xuyên suốt trong tài liệu của họ mà thôi.

### 21.2.3. Một số khái niệm khác

#### \* Đồng cấu

Hai đồ thị  $G = (V, E)$  và  $G' = (V', E')$  được gọi là *đồng cấu* (*isomorphic*) nếu tồn tại một song ánh  $f: V \rightarrow V'$  sao cho số cung nối  $u$  với  $v$  trên  $E$  bằng số cung nối  $f(u)$  với  $f(v)$  trên  $E'$ . Nói cách khác, ta có thể “đặt tên” lại các đỉnh trong  $G$  để thu được đồ thị  $G'$ .



Hình 21-3. Đồng cấu

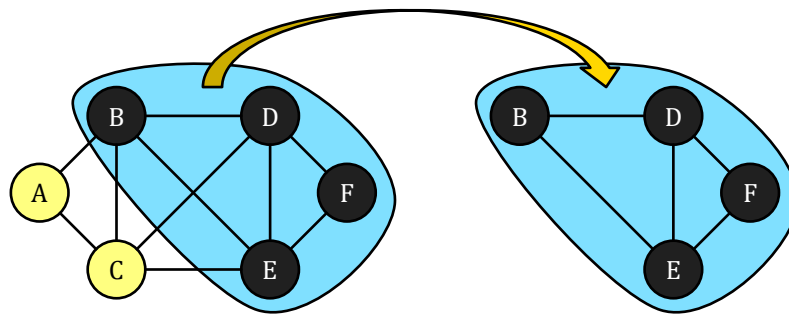
#### \* Đồ thị con

Đồ thị  $G' = (V', E')$  là *đồ thị con* (*subgraph*) của đồ thị  $G = (V, E)$  nếu  $V' \subseteq V$  và  $E' \subseteq E$ .

Đồ thị con  $G_U = (U, E_U)$  được gọi là *đồ thị con cảm ứng* (*induced graph*) từ đồ thị  $G$  bởi tập  $U \subseteq V$  nếu:

$$E_U = \{(u, v) \in E : u, v \in U\}$$

trong trường hợp này chúng ta còn nói  $G_U$  là đồ thị  $G$  hạn chế trên  $U$ .



Hình 21-4. Đồ thị con cảm ứng

#### \* Phiên bản có hướng/vô hướng

Với một đồ thị vô hướng  $G = (V, E)$ , ta gọi *phiên bản có hướng* (*directed version*) của  $G$  là một đồ thị có hướng  $G' = (V, E')$  tạo thành từ  $G$  bằng cách thay mỗi cạnh  $(u, v)$  bằng hai cung có hướng ngược chiều nhau:  $(u, v)$  và  $(v, u)$ .

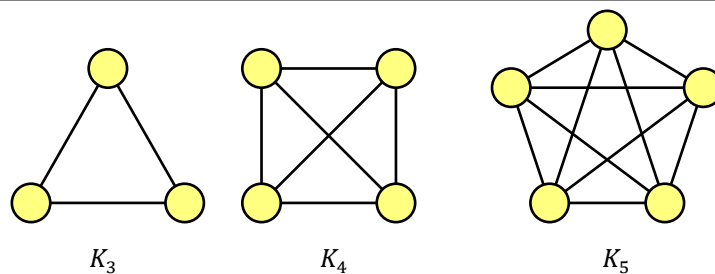
Với một đồ thị có hướng  $G = (V, E)$ , ta gọi *phiên bản vô hướng* (*undirected version*) của  $G$  là một đồ thị vô hướng  $G' = (V, E')$  tạo thành bằng cách thay mỗi cung  $(u, v)$  bằng cạnh vô hướng  $(u, v)$ . Nói cách khác,  $G'$  tạo thành từ  $G$  bằng cách bỏ đi chiều của cung.

### \* Tính liên thông

Một đồ thị vô hướng gọi là *liên thông* (*connected*) nếu giữa hai đỉnh bất kỳ của đồ thị có tồn tại đường đi. Đối với đồ thị có hướng, có hai khái niệm liên thông tùy theo chúng ta có quan tâm tới hướng của các cung hay không. Đồ thị có hướng gọi là *liên thông mạnh* (*strongly connected*) nếu giữa hai đỉnh bất kỳ của đồ thị có tồn tại đường đi. Đồ thị có hướng gọi là *liên thông yếu* (*weakly connected*) nếu phiên bản vô hướng của nó là đồ thị liên thông.

### \* Đồ thị đầy đủ

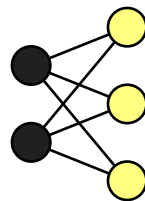
Một đồ thị vô hướng được gọi là *đầy đủ* (*complete*) nếu mọi cặp đỉnh đều là kề nhau, đồ thị đầy đủ gồm  $n$  đỉnh ký hiệu là  $K_n$ . Hình 22-4 là ví dụ về các đồ thị  $K_3$ ,  $K_4$  và  $K_5$ .



Hình 21-5. Đồ thị đầy đủ

### \* Đồ thị hai phía

Một đồ thị vô hướng gọi là *hai phía* (*bipartite*) nếu tập đỉnh của nó có thể chia làm hai tập rời nhau  $X, Y$  sao cho không tồn tại cạnh nối hai đỉnh thuộc  $X$  cũng như không tồn tại cạnh nối hai đỉnh thuộc  $Y$ . Nếu  $|X| = m$  và  $|Y| = n$  và giữa mọi cặp đỉnh  $(x, y)$  trong đó  $x \in X, y \in Y$  đều có cạnh nối thì đồ thị hai phía đó được gọi là đồ thị hai phía đầy đủ, ký hiệu  $K_{m,n}$ . Hình 22-4 là ví dụ về đồ thị hai phía đầy đủ  $K_{2,3}$ .



Hình 21-6. Đồ thị hai phía đầy đủ

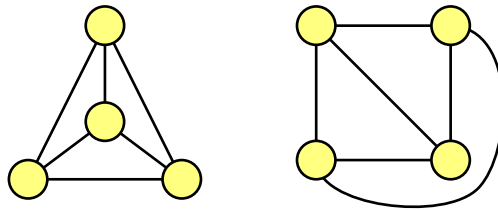
### \* Đồ thị phẳng

Một đồ thị được gọi là *đồ thị phẳng* (*planar graph*) nếu chúng ta có thể vẽ đồ thị ra trên mặt phẳng sao cho:

- ✿ Mỗi đỉnh tương ứng với một điểm trên mặt phẳng, không có hai đỉnh cùng tọa độ.
- ✿ Mỗi cạnh tương ứng với một đoạn đường liên tục nối hai đỉnh, các điểm nằm trên hai cạnh bất kỳ là không giao nhau ngoại trừ các điểm đầu mút (tương ứng với các đỉnh)

Phép vẽ đồ thị phẳng như vậy gọi là biểu diễn phẳng của đồ thị

Ví dụ như đồ thị đầy đủ  $K_4$  là đồ thị phẳng bởi nó có thể vẽ ra trên mặt phẳng như Hình 21-7



Hình 21-7. Hai cách biểu diễn phẳng của đồ thị đầy đủ 4 đỉnh

### **Định lý 21-3 (Định lý Kuratowski)**

Một đồ thị vô hướng là đồ thị phẳng nếu và chỉ nếu nó không chứa đồ thị con đẳng cấu với  $K_{3,3}$  hoặc  $K_5$ .

### **Định lý 21-4 (Công thức Euler)**

Nếu một đồ thị vô hướng liên thông là đồ thị phẳng và biểu diễn phẳng của đồ thị đó gồm  $v$  đỉnh và  $e$  cạnh chia mặt phẳng thành  $f$  phần thì  $v - e + f = 2$ .

### **Định lý 21-5**

Nếu đơn đồ thị vô hướng  $G = (V, E)$  là đồ thị phẳng có ít nhất 3 đỉnh thì  $|E| \leq 3|V| - 6$ . Ngoài ra nếu  $G$  không có chu trình độ dài 3 thì  $|E| \leq 2|V| - 4$ .

Định lý 21-5 chỉ ra rằng số cạnh của đơn đồ thị phẳng là một đại lượng  $|E| = O(|V|)$  điều này rất hữu ích đối với nhiều thuật toán trên đồ thị thưa (có ít cạnh).

### **\* Đồ thị đường**

Từ đồ thị vô hướng  $G$ , ta xây dựng đồ thị vô hướng  $G'$  như sau: Mỗi đỉnh của  $G'$  tương ứng với một cạnh của  $G$ , giữa hai đỉnh  $x, y$  của  $G'$  có cạnh nối nếu và chỉ nếu tồn tại đỉnh liên thuộc với cả hai cạnh  $x, y$  trên  $G$ . Đồ thị  $G'$  như vậy được gọi là đồ thị đường của đồ thị  $G$ . Đồ thị đường được nghiên cứu trong các bài toán kiểm tra tính liên thông, tập độc lập cực đại, tô màu cạnh đồ thị, chu trình Euler và chu trình Hamilton v.v...

## Chương 22. Biểu diễn đồ thị

Khi lập trình giải các bài toán được mô hình hoá bằng đồ thị, việc đầu tiên cần làm tìm cấu trúc dữ liệu để biểu diễn đồ thị sao cho việc giải quyết bài toán được thuận tiện nhất.

Có rất nhiều phương pháp biểu diễn đồ thị, trong bài này chúng ta sẽ khảo sát một số phương pháp phổ biến nhất. Tính hiệu quả của từng phương pháp biểu diễn sẽ được chỉ rõ hơn trong từng thuật toán cụ thể.

### 22.1. Ma trận kề

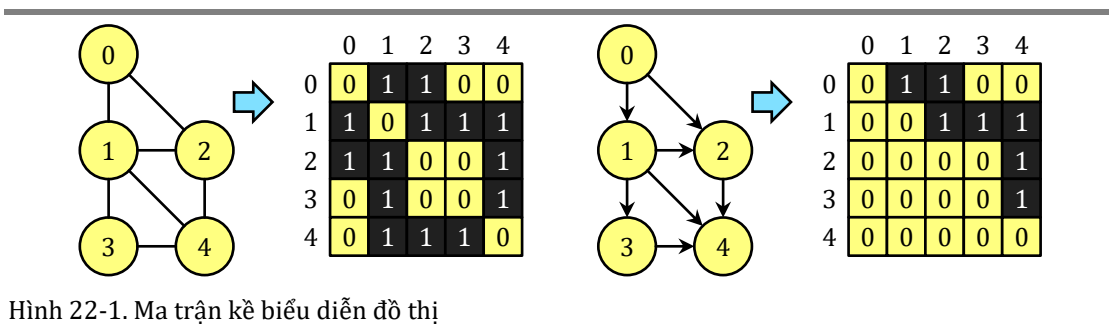
Với  $G = (V, E)$  là một đơn đồ thị có hướng trong đó  $|V| = n$ , ta có thể đánh số các đỉnh từ 0 tới  $n - 1$  và đồng nhất mỗi đỉnh với số thứ tự của nó. Bằng cách đánh số như vậy, đồ thị  $G$  có thể biểu diễn bằng ma trận vuông  $A = \{a_{ij}\}_{n \times n}$  với các hàng và các cột đánh số từ 0 tới  $n - 1$ . Trong đó:

$$a_{ij} = \begin{cases} 1, & \text{nếu } (i, j) \in E \\ 0, & \text{nếu } (i, j) \notin E \end{cases}$$

(Cũng có thể dùng hai giá trị kiểu bool: true/false thay cho hai giá trị 0/1)

Với  $\forall i$ , giá trị của các phần tử trên đường chéo chính ma trận  $A$ :  $\{a_{ii}\}$  có thể đặt tùy theo mục đích cụ thể, chẳng hạn đặt bằng 0. Ma trận  $A$  xây dựng như vậy được gọi là *ma trận kề* (*adjacency matrix*) của đồ thị  $G$ . Việc biểu diễn đồ thị vô hướng được quy về việc biểu diễn phiên bản có hướng tương ứng: thay mỗi cạnh  $(i, j)$  bởi hai cung ngược hướng nhau:  $(i, j)$  và  $(j, i)$ .

Đối với đa đồ thị thì việc biểu diễn cũng tương tự trên, chỉ có điều nếu như  $(i, j)$  là cung thì  $a_{ij}$  là số cạnh nối giữa đỉnh  $i$  và đỉnh  $j$ .



Ma trận kề có một số tính chất:

- ✿ Đối với đồ thị vô hướng  $G$ , thì ma trận kề tương ứng là ma trận đối xứng,  $A = A^T$  ( $\forall i, j: a_{ij} = a_{ji}$ ), điều này không đúng với đồ thị có hướng.
- ✿ Nếu  $G$  là đồ thị vô hướng và  $A$  là ma trận kề tương ứng thì trên ma trận  $A$ , tổng các số trên hàng  $i$  bằng tổng các số trên cột  $i$  và bằng bậc của đỉnh  $i$ :  $\deg(i)$
- ✿ Nếu  $G$  là đồ thị có hướng và  $A$  là ma trận kề tương ứng thì trên ma trận  $A$ , tổng các số trên hàng  $i$  bằng bậc ra của đỉnh  $i$ :  $\deg^+(i)$ , tổng các số trên cột  $i$  bằng bậc vào của đỉnh  $i$ :  $\deg^-(i)$

Ưu điểm của ma trận kề:

- ✿ Đơn giản, trực quan, dễ cài đặt trên máy tính
- ✿ Để kiểm tra xem hai đỉnh  $(u, v)$  của đồ thị có kề nhau hay không, ta chỉ việc kiểm tra bằng một phép so sánh:  $a_{uv} \neq 0$

Nhược điểm của ma trận kề

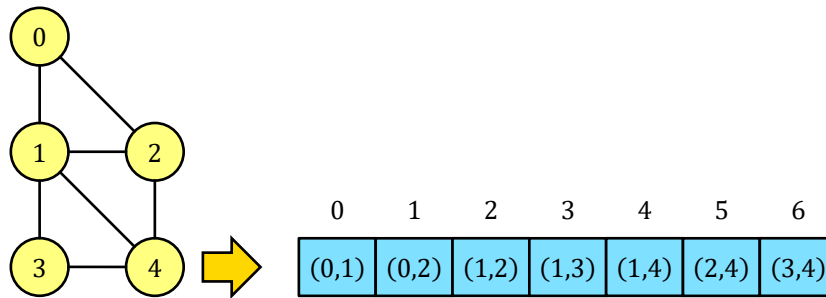
- ✿ Bất kể số cạnh của đồ thị là nhiều hay ít, ma trận kề luôn luôn đòi hỏi  $n^2$  ô nhớ để lưu các phần tử ma trận, điều đó gây lãng phí bộ nhớ.
- ✿ Một số bài toán yêu cầu thao tác liệt kê tất cả các đỉnh  $v$  kề với một đỉnh  $u$  cho trước. Trên ma trận kề việc này được thực hiện bằng cách xét tất cả các đỉnh  $v$  và kiểm tra điều kiện  $a_{uv} \neq 0$ . Như vậy, ngay cả khi đỉnh  $u$  là *đỉnh cô lập* (không kề với đỉnh nào) hoặc *đỉnh treo* (chỉ kề với 1 đỉnh) ta cũng buộc phải xét tất cả các đỉnh  $v$  và kiểm tra giá trị tương ứng  $a_{uv}$ .

## 22.2. Danh sách cạnh

Với đồ thị  $G = (V, E)$  có  $n$  đỉnh,  $m$  cạnh, ta có thể liệt kê tất cả các cạnh của đồ thị trong một danh sách, mỗi phần tử của danh sách là một cặp  $(x, y)$  tương ứng với một cạnh của  $E$ , trong trường hợp đồ thị có hướng thì mỗi cặp  $(x, y)$  tương ứng với một cung,  $x$  là đỉnh đầu và  $y$  là đỉnh cuối của cung. Cách biểu diễn này gọi là *danh sách cạnh* (*edge list*).

Có nhiều cách xây dựng cấu trúc dữ liệu để biểu diễn danh sách, nhưng phổ biến nhất là dùng mảng hoặc danh sách móc nối.





Hình 22-2. Danh sách cạnh

Ưu điểm của danh sách cạnh:

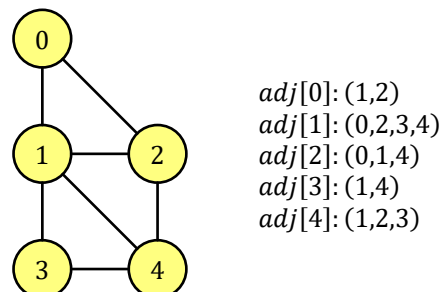
- Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ, bởi nó chỉ cần  $O(m)$  ô nhớ để lưu danh sách cạnh.
- Trong một số trường hợp, ta phải xét tất cả các cạnh của đồ thị thì cài đặt trên danh sách cạnh làm cho việc duyệt các cạnh dễ dàng hơn (Chẳng hạn như thuật toán Kruskal).

Nhược điểm của danh sách cạnh:

- Nhược điểm cơ bản của danh sách cạnh là khi ta cần duyệt tất cả các đỉnh kề với đỉnh  $v$  nào đó của đồ thị, thì chẳng có cách nào khác là phải duyệt tất cả các cạnh, lọc ra những cạnh có chứa đỉnh  $v$  và xét đỉnh còn lại.
- Việc kiểm tra hai đỉnh  $u, v$  có kề nhau hay không cũng bắt buộc phải duyệt danh sách cạnh, điều đó khá tốn thời gian trong trường hợp đồ thị dày (nhiều cạnh).

### 22.3. Danh sách kề

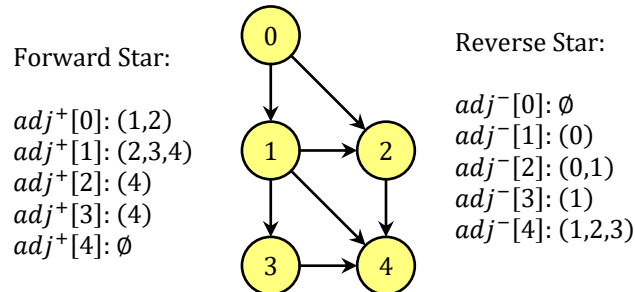
Để khắc phục nhược điểm của các phương pháp ma trận kề và danh sách cạnh, người ta đề xuất phương pháp biểu diễn đồ thị bằng *danh sách kề* (*adjacency list*). Trong cách biểu diễn này, với mỗi đỉnh  $u$  của đồ thị vô hướng, ta cho tương ứng với nó một danh sách  $adj[u]$  gồm các đỉnh kề với  $u$  (Hình 22-4).



Hình 22-3. Danh sách kề

Với đồ thị có hướng  $G = (V, E)$ . Có hai cách cài đặt danh sách kề phổ biến:

- ✿ Forward Star: Với mỗi đỉnh  $u$ , lưu trữ một danh sách  $adj^+[u]$  chứa các đỉnh nối từ  $u$ :  $adj^+[u] = \{v: (u, v) \in E\}$ .
- ✿ Reverse Star: Với mỗi đỉnh  $v$ , lưu trữ một danh sách  $adj^- [v]$  chứa các đỉnh nối tới  $v$ :  $adj^- [v] = \{u: (u, v) \in E\}$



Hình 22-4. Danh sách kề forward star và reverse star

Tùy theo từng bài toán, chúng ta sẽ chọn cấu trúc Forward Star hoặc Reverse Star để biểu diễn đồ thị. Có những bài toán yêu cầu phải biểu diễn đồ thị bằng cả hai cấu trúc Forward Star và Reverse Star.

Bất cứ cấu trúc dữ liệu nào có khả năng biểu diễn danh sách (mảng, danh sách móc nối, cây...) đều có thể sử dụng để biểu diễn danh sách kề, nhưng mảng và danh sách móc nối được sử dụng phổ biến nhất. Chẳng hạn trong C++, ta có thể dùng vector, list, set, ... để biểu diễn mỗi danh sách  $adj[.]$ .

Ưu điểm của danh sách kề: Danh sách kề cho phép dễ dàng duyệt tất cả các đỉnh kề với một đỉnh  $u$  cho trước.

Nhược điểm của danh sách kề: Danh sách kề yếu hơn ma trận kề ở việc kiểm tra  $(u, v)$  có phải là cạnh hay không, bởi trong cách biểu diễn này ta sẽ phải việc phải duyệt toàn bộ danh sách kề của  $u$  hay danh sách kề của  $v$ .

## 22.4. Danh sách liên thuộc

Danh sách liên thuộc (incidence lists) là một mở rộng của danh sách kề. Nếu như trong biểu diễn danh sách kề, mỗi đỉnh được cho tương ứng với một danh sách các đỉnh kề thì trong biểu diễn danh sách liên thuộc, mỗi đỉnh được cho tương ứng với một danh sách các cạnh liên thuộc. Chính vì vậy, những kỹ thuật cài đặt danh sách kề có thể sửa đổi một chút để cài đặt danh sách liên thuộc.

## 22.5. Chuyển đổi giữa các cách biểu diễn đồ thị

Có một số thuật toán mà tính hiệu quả của nó phụ thuộc rất nhiều vào cách thức biểu diễn đồ thị, do đó khi bắt tay vào giải quyết một bài toán đồ thị, chúng ta phải tìm cấu trúc dữ liệu phù hợp để biểu diễn đồ thị sao cho hợp lý nhất. Nếu đồ thị đầu vào được cho bởi một cách biểu diễn bất hợp lý, chúng ta cần chuyển đổi cách biểu diễn khác để thuận tiện trong việc triển khai thuật toán.

Với các lớp mẫu của C++ biểu diễn cấu trúc dữ liệu mảng động, danh sách móc nối, tập hợp, việc chuyển đổi giữa các cách biểu diễn đồ thị khá dễ dàng, ta sẽ trình bày chúng trong chương trình cài đặt các thuật toán. Cũng có một số thuật toán không phụ thuộc nhiều vào cách biểu diễn đồ thị, trong trường hợp này ta sẽ chọn cấu trúc dữ liệu dễ cài đặt nhất để việc đọc hiểu thuật toán/chương trình được thuận tiện hơn.

Trong những chương trình mô tả thuật toán của phần này, nếu không có ghi chú thêm, ta giả thiết rằng các đồ thị được cho có  $n$  đỉnh và  $m$  cạnh. Các đỉnh được đánh số từ 0 tới  $n - 1$  và các cạnh đánh số từ 0 tới  $m - 1$ . Các đỉnh cũng như các cạnh được đồng nhất với số hiệu của chúng.

### Bài tập 22-1

Cho một đồ thị có hướng  $n$  đỉnh,  $m$  cạnh được biểu diễn bằng danh sách kề, trong đó mỗi đỉnh  $u$  sẽ được cho tương ứng với một danh sách các đỉnh nối từ  $u$ . Cho một đỉnh  $v$ , hãy tìm thuật toán tính bán bậc ra và bán bậc vào của  $v$ . Xác định độ phức tạp tính toán của thuật toán

### Bài tập 22-2

Đồ thị chuyển vị của đồ thị có hướng  $G = (V, E)$  là đồ thị  $G^T = (V, E^T)$ , trong đó:

$$E^T = \{(u, v) : (v, u) \in E\}$$

Hãy tìm thuật toán xây dựng  $G^T$  từ  $G$  trong hai trường hợp:  $G$  và  $G^T$  được biểu diễn bằng ma trận kề;  $G$  và  $G^T$  được biểu diễn bằng danh sách kề.

### Bài tập 22-3

Cho đa đồ thị vô hướng  $G = (V, E)$  được biểu diễn bằng danh sách kề, hãy tìm thuật toán  $O(|V| + |E|)$  để xây dựng đơn đồ thị  $G' = (V, E')$  và biểu diễn  $G'$  bằng danh sách kề, biết rằng đồ thị  $G'$  gồm tất cả các đỉnh của đồ thị  $G$  và các cạnh song song trên  $G$  được thay thế bằng duy nhất một cạnh trong  $G'$ .

### Bài tập 22-4

Cho đa đồ thị  $G$  được biểu diễn bằng ma trận kề  $A = \{a_{ij}\}$  trong đó  $a_{ij}$  là số cạnh nối từ đỉnh  $i$  tới đỉnh  $j$ . Hãy chứng minh rằng nếu  $B = A^k$  thì  $b_{ij}$  là số đường đi từ đỉnh  $i$  tới đỉnh  $j$  qua đúng  $k$  cạnh.

Gợi ý: Sử dụng chứng minh quy nạp.

### Bài tập 22-5

Cho đơn đồ thị  $G = (V, E)$ , ta gọi bình phương của một đồ thị  $G$  là đơn đồ thị

$$G^2 = (V, E^2)$$



sao cho  $(u, v) \in E^2$  nếu và chỉ nếu tồn tại một đỉnh  $w \in V$  sao cho  $(u, w)$  và  $(w, v)$  đều thuộc  $E$ .

Hãy tìm thuật toán  $O(|V|^3)$  để xây dựng  $G^2$  từ  $G$  trong trường hợp cả  $G$  và  $G^2$  được biểu diễn bằng ma trận kề, tìm thuật toán  $O(|E||V| + |V|^2)$  để xây dựng  $G^2$  từ  $G$  trong trường hợp cả  $G$  và  $G^2$  được biểu diễn bằng danh sách kề.

### Bài tập 22-6

Xây dựng cấu trúc dữ liệu để biểu diễn đơn đồ thị vô hướng và các thao tác:

- ✿ Liệt kê các đỉnh kề với một đỉnh  $u$  cho trước trong thời gian  $O(\deg(u))$
- ✿ Kiểm tra hai đỉnh có kề nhau hay không trong thời gian  $O(1)$
- ✿ Loại bỏ một cạnh trong thời gian  $O(1)$
- ✿ Bổ sung một cạnh  $(u, v)$  nếu nó chưa có trong thời gian  $O(1)$

### Bài tập 22-7

Với đồ thị  $G = (V, E)$  được biểu diễn bằng ma trận kề, đa số các thuật toán trên đồ thị sẽ có độ phức tạp tính toán  $\Omega(|V|^2)$ , tuy nhiên không phải không có ngoại lệ. Chẳng hạn bài toán tìm “bồn chứa” (*universal sink*) trong đồ thị: bồn chứa trong đồ thị có hướng là một đỉnh nối từ tất cả các đỉnh khác và không có cung đi ra. Hãy tìm thuật toán  $O(|V|)$  để xác định sự tồn tại và chỉ ra bồn chứa trong đồ thị có hướng.

### Bài tập 22-8

Xét đồ thị vô hướng  $G = (V, E)$  với các đỉnh đánh số từ 0 tới  $n - 1$  và các cạnh đánh số từ 0 tới  $m - 1$ . Xây dựng Ma trận liên thuộc (*incidence matrix*)  $B = \{b_{ij}\}_{n \times m}$  trong đó:

$$b_{ij} = \begin{cases} 1, & \text{nếu cung } j \text{ liên thuộc với đỉnh } i \\ 0, & \text{trong trường hợp ngược lại} \end{cases}$$

Xét ma trận  $A = B \cdot B^T = \{a_{ij}\}_{n \times n}$ , chứng minh rằng nếu đồ thị không có khuyên thì:

- ✿  $\forall i \neq j, a_{ij}$  là số cạnh nối giữa đỉnh  $i$  và đỉnh  $j$  trên  $G$
- ✿  $\forall i, a_{ii}$  là bậc của đỉnh  $i$

Nói cách khác,  $B \cdot B^T$  là ma trận kề của  $G$ , các phần tử trên đường chéo chính tương ứng với bậc đỉnh.

### Bài tập 22-9

Xét đồ thị vô hướng  $G = (V, E)$  với ma trận liên thuộc  $B$  định nghĩa như trong Bài tập 22-8. Xét đồ thị đường  $L(G)$ , trong đó mỗi đỉnh của  $L(G)$  ứng với một cạnh của  $G$ . Hai đỉnh  $i, j$  của  $L(G)$  kề nhau nếu tồn tại một đỉnh  $u$  của  $G$  liên thuộc với cả hai cạnh  $i, j$ . Gọi  $A$  là ma trận kề của đồ thị  $L(G)$  xác định bởi

$$a_{ij} = |\{u \in G: \text{cả cạnh } i \text{ và cạnh } j \text{ liên thuộc với } u\}|$$

Chứng minh rằng nếu đồ thị  $G$  không có khuyên thì  $A = B^T \cdot B$

## Chương 23. Các thuật toán tìm kiếm trên đồ thị

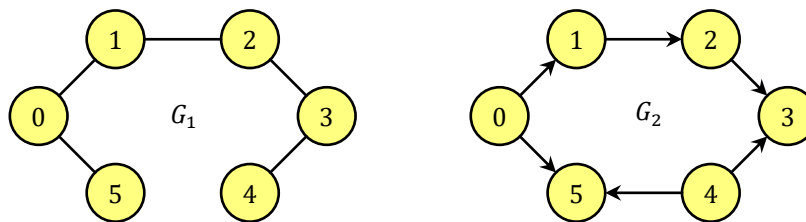
### 23.1. Bài toán tìm đường

Cho đồ thị  $G = (V, E)$  và hai đỉnh  $s, t \in V$ .

Nhắc lại định nghĩa đường đi: Một dãy các đỉnh:

$$P = \langle s = p_0, p_1, \dots, p_k = t \rangle, (\forall i: (p_{i-1}, p_i) \in E)$$

được gọi là một đường đi từ  $s$  tới  $t$ , đường đi này gồm  $k + 1$  đỉnh  $p_0, p_1, \dots, p_k$  và  $k$  cạnh  $(p_0, p_1), (p_1, p_2), \dots, (p_{k-1}, p_k)$ . Đỉnh  $s$  được gọi là đỉnh đầu và đỉnh  $t$  được gọi là đỉnh cuối của đường đi. Nếu tồn tại một đường đi từ  $s$  tới  $t$ , ta nói  $s$  đến được  $t$  và  $t$  đến được từ  $s$ :  $s \rightsquigarrow t$ .



Hình 23-1. Đồ thị và đường đi

Trên cả hai đồ thị ở Hình 23-1,  $\langle 0, 1, 2, 3 \rangle$  là đường đi từ đỉnh 0 tới đỉnh 3.  $\langle 0, 5, 4, 3 \rangle$  không phải đường đi vì không có cạnh (cung)  $(5, 4)$ .

Một bài toán quan trọng trong lý thuyết đồ thị là bài toán duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó. Vấn đề này đưa về một bài toán liệt kê mà yêu cầu của nó là không được bỏ sót hay lặp lại bất kỳ đỉnh nào. Chính vì vậy mà ta phải xây dựng những thuật toán cho phép duyệt một cách hệ thống các đỉnh, những thuật toán như vậy gọi là những thuật toán tìm kiếm trên đồ thị (*graph traversal*). Ta quan tâm đến hai thuật toán cơ bản nhất: thuật toán tìm kiếm theo chiều sâu và thuật toán tìm kiếm theo chiều rộng.

Khuôn dạng Input/Output quy định như sau:

#### Input

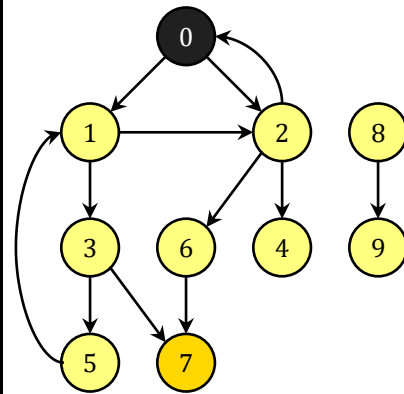
- ✿ Dòng 1 chứa 4 số  $n, m, s, t$  lần lượt là số đỉnh, số cung, đỉnh xuất phát và đỉnh cần đến của một đồ thị có hướng  $G$  ( $n, m \leq 10^6$ )
- ✿  $m$  dòng tiếp theo, mỗi dòng chứa hai số nguyên  $u, v$  cho biết có cung  $(u, v)$  trên đồ thị

#### Output

- ✿ Danh sách các đỉnh có thể đến được từ  $s$
- ✿ Đường đi từ  $s$  tới  $t$  nếu có

Sample Input	Sample Output
--------------	---------------

10 12 0 7	Reachable vertices from 0:
0 1	0, 1, 2, 4, 6, 7, 3, 5,
0 2	The path from 0 to 7:
1 2	7 <- 6 <- 2 <- 1 <- 0
1 3	
2 0	
2 4	
2 6	
3 5	
3 6	
5 1	
6 7	
8 9	



## 23.2. Biểu diễn đồ thị

Mặc dù đồ thị được cho trong input dưới dạng danh sách cạnh, đối với những thuật toán trong bài, cách biểu diễn đồ thị hiệu quả nhất là sử dụng danh sách kề hoặc danh sách liên thuộc: Mỗi đỉnh  $u$  tương ứng với một danh sách  $adj[u]$  chứa các đỉnh nối từ  $u$  (danh sách kề dạng forward star).

## 23.3. Thuật toán tìm kiếm theo chiều sâu

### 23.3.1. Ý tưởng

Tư tưởng của *thuật toán tìm kiếm theo chiều sâu* (Depth-First Search – DFS) có thể trình bày như sau: Trước hết, dĩ nhiên đỉnh  $s$  đến được từ  $s$ , tiếp theo, với mọi cung  $(s, x)$  của đồ thị thì  $x$  cũng sẽ đến được từ  $s$ . Với mỗi đỉnh  $x$  đó thì tất nhiên những đỉnh  $y$  nối từ  $x$  cũng đến được từ  $s$ ... Điều đó gợi ý cho ta viết một hàm đệ quy  $DFSVisit(u)$  mô tả việc duyệt từ đỉnh  $u$  bằng cách thăm đỉnh  $u$  và tiếp tục quá trình duyệt  $DFSVisit(v)$  với  $v$  là một đỉnh chưa thăm nối từ  $u$ .

Kỹ thuật đánh dấu được sử dụng để tránh việc liệt kê lặp các đỉnh: Khởi tạo  $avail[v] = \text{true}, \forall v \in V$ , mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại ( $avail[v] = \text{false}$ ) để các bước duyệt đệ quy kế tiếp không duyệt lại đỉnh đó nữa.

Để lưu lại đường đi từ đỉnh xuất phát  $s$ , trong hàm  $DFSVisit(u)$ , trước khi gọi đệ quy  $DFSVisit(v)$  với  $v$  là một đỉnh chưa thăm nối từ  $u$  (chưa đánh dấu), ta lưu lại vết đường đi từ  $u$  tới  $v$  bằng cách đặt  $trace[v] = u$ , tức là  $trace[v]$  lưu lại đỉnh liền trước  $v$  trong đường đi từ  $s$  tới  $v$ . Khi thuật toán DFS kết thúc, đường đi từ  $s$  tới  $t$  sẽ là:

$$\langle p_0 = t \leftarrow p_1 = trace[p_0] \leftarrow p_2 = trace[p_1] \leftarrow \dots \leftarrow s \rangle$$

```
void DFSVisit(u ∈ V)
{
    avail[u] = false; //Đánh dấu avail[u] = false ⇔ u đã thăm
    Output ← u; //liệt kê u
    for (∀v ∈ adj[u]) //duyệt mọi đỉnh v nối từ u
        if (avail[v]) //nếu v chưa thăm
        {
            trace[v] = u; //Lưu vết: đỉnh liền trước v trên đường đi từ s tới v là đỉnh u
            DFSVisit(v); //Gọi đệ quy tìm kiếm theo chiều sâu từ v
        }
}
```

Input → đồ thị G, đỉnh xuất phát s, đỉnh cần đến t;  
for (∀u ∈ V) avail[u] = true; //Các đỉnh đều đánh dấu chưa thăm  
DFSVisit(s);  
if (!avail[t]) //Từ s có đường tới t  
 «Truy theo vết từ t để tìm đường đi từ s tới t»;

### 23.3.2. Cài đặt

#### DFS.CPP ✓ Tìm đường bằng DFS

```
#include <iostream>
#include <vector>
using namespace std;
const int maxN = 1e6;

int n, m, s, t, trace[maxN];
bool avail[maxN];
vector<int> adj[maxN];

void Enter() //Nhập dữ liệu
{
    cin >> n >> m >> s >> t;
    while (m-- > 0)
    {
        int u, v;
        cin >> u >> v; //Đọc một cạnh (u, v)
        adj[u].push_back(v); //Đưa v vào danh sách kề của u
    }
}

void DFSVisit(int u) //Thuật toán tìm kiếm theo chiều sâu bắt đầu từ u
{
    avail[u] = false; //Đánh dấu u đã thăm
    cout << u << " "; //Liệt kê u
    for (int v: adj[u]) //Xét mọi đỉnh v nối từ u
        if (avail[v]) //Nếu v chưa thăm
        {
            trace[v] = u; //Lưu vết đường đi s → ... → u → v
            DFSVisit(v); //Đệ quy tìm kiếm theo chiều sâu bắt đầu từ v
        }
}
```



```
void PrintPath() //In đường đi
{
    if (avail[t]) //t chưa thăm: không có đường từ s tới t
        cout << "There's no path from " << s << " to " << t << '\n';
    else
    {
        cout << "The path from " << s << " to " << t << ":\n";
        for (int u = t; u != s; u = trace[u]) //Truy vết ngược từ t về s
            cout << u << " <- ";
        cout << s << "\n";
    }
}

int main()
{
    Enter();
    fill(avail, avail + n, true); //Khởi tạo các đỉnh đều chưa thăm
    cout << "Reachable vertices from " << s << ":\n";
    DFSVisit(s); //Thuật toán tìm kiếm theo chiều sâu bắt đầu từ s
    cout << "\n";
    PrintPath();
}
```

Về kỹ thuật, có thể không cần mảng đánh dấu  $avail[0 \dots n]$  mà dùng luôn mảng  $trace[0 \dots n]$  để đánh dấu: Khởi tạo các phần tử mảng  $trace[0 \dots n]$  là:

$$\begin{cases} trace[s] \neq -1 \\ trace[v] = -1, \forall v \neq s \end{cases}$$

Khi đó điều kiện để một đỉnh  $v$  chưa thăm là  $trace[v] = -1$ , mỗi khi từ đỉnh  $u$  thăm đỉnh  $v$ , phép gán  $trace[v] = u$  sẽ kiêm luôn công việc đánh dấu  $v$  đã thăm ( $trace[v] \neq -1$ ).

Chương trình cài đặt in ra các đỉnh đường đi theo thứ tự ngược (từ  $t$  về  $s$ ). Để in đường đi theo đúng thứ tự từ  $s$  tới  $t$ , có thể sử dụng một số kỹ thuật đơn giản:

Lưu đường đi vào mảng và in mảng ra theo thứ tự ngược lại.

Hoặc truy vết bằng đệ quy bằng lời gọi  $DoTrace(t)$ :

```
void DoTrace(int v)
{
    if (v != s)
    {
        DoTrace(trace[v]);
        cout << " -> ";
    }
    cout << v;
}
```

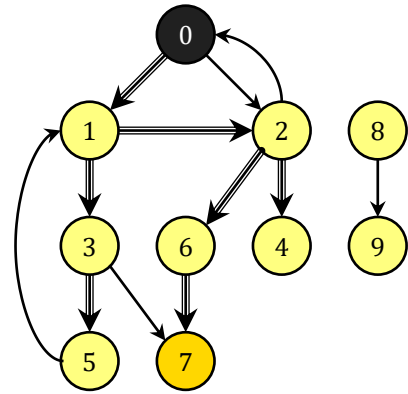
Cũng có thể thay danh sách kề dạng forward star bằng danh sách kề dạng reverse star rồi thực hiện thuật toán tìm kiếm theo chiều sâu bắt đầu từ  $t$ , sau đó truy vết từ  $s$ .

### 23.3.3. Một vài tính chất của DFS

#### \* Cây DFS

Nếu ta sắp xếp danh sách kề của mỗi đỉnh theo thứ tự tăng dần thì thuật toán DFS luôn trả về đường đi có thứ tự từ điển nhỏ nhất trong số tất cả các đường đi từ  $s$  tới  $t$ .

Quá trình tìm kiếm theo chiều sâu cho ta một cây DFS gốc  $s$ . Quan hệ cha-con trên cây được định nghĩa là: nếu từ đỉnh  $u$  tới thăm đỉnh  $v$  ( $\text{DFSVisit}(u)$  gọi  $\text{DFSVisit}(v)$ ) thì  $u$  là nút cha của nút  $v$ . Hình bên là đồ thị với danh sách kề của mỗi đỉnh được sắp xếp tăng dần và cây DFS tương ứng với đỉnh xuất phát  $s = 0$ . Chú ý rằng cấu trúc của cây DFS phụ thuộc vào thứ tự trong các danh sách kề.



#### \* Mô hình duyệt đồ thị theo DFS

Cài đặt trên chỉ là một ứng dụng của thuật toán DFS để liệt kê các đỉnh đến được từ một đỉnh. Thuật toán DFS dùng để duyệt qua các đỉnh và các cạnh của đồ thị được viết theo mô hình sau:

```
void DFSVisit( $u \in V$ )
{
     $d[u] = ++\text{Time}$ ; //  $d[u]$  = thời điểm  $u$  được thăm, cũng là đánh dấu  $\neq 0$ 
    for ( $\text{int } v: \text{adj}[u]$ ) // duyệt mọi đỉnh  $v$  nối từ  $u$ 
        if ( $d[v] == 0$ ) // nếu  $v$  chưa thăm
            DFSVisit( $v$ ); // Gọi đệ quy tìm kiếm theo chiều sâu từ  $v$ 
     $f[u] = ++\text{Time}$ ; //  $f[u]$  = thời điểm  $u$  được duyệt xong
}
```

```
Input  $\rightarrow$  đồ thị  $G$ ;
for ( $\forall u \in V$ )  $d[u] = 0$ ; // Các đỉnh đều chưa thăm:  $d[.] = 0$ 
Time = 0;
for ( $\forall u \in V$ ) // Xét mọi điểm chưa thăm
    if ( $d[u] == 0$ ) DFSVisit( $u$ );
```

Thời gian thực hiện giải thuật của DFS có thể đánh giá như sau: Với mỗi đỉnh  $u$  thì hàm  $DFSVisit(u)$  được gọi đúng 1 lần và trong hàm đó ta có chi phí xét tất cả các đỉnh  $v$  nối từ  $u$ , cũng là chi phí quét tất cả các cung đi ra khỏi  $u$ .

- ✿ Nếu đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, mỗi cung của đồ thị sẽ bị duyệt qua đúng 1 lần, trong trường hợp này, thời gian thực hiện giải thuật DFS là  $\Theta(n + m)$  bao gồm  $n$  lần gọi hàm  $DFSVisit(.)$  và  $m$  lần duyệt qua cung.
- ✿ Nếu đồ thị được biểu diễn bằng ma trận kề, để xét tất cả các đỉnh nối từ một đỉnh  $u$  ta phải quét qua toàn bộ tập đỉnh mất thời gian  $\Theta(n)$ . Trong trường hợp này thời gian thực hiện giải thuật DFS là  $\Theta(n^2)$ .
- ✿ Nếu đồ thị được biểu diễn bằng danh sách cạnh, để xét tất cả các đỉnh nối từ một đỉnh  $u$  ta phải quét toàn bộ danh sách cạnh mất thời gian  $\Theta(m)$ . Trong trường hợp này thời gian thực hiện giải thuật DFS là  $\Theta(n \cdot m)$ .

#### ✿ Thứ tự thăm đến và duyệt xong

Hãy để ý hàm  $DFSVisit(u)$ :

- ✿ Khi bắt đầu vào hàm ta nói đỉnh  $u$  được *thăm đến* (*discover*), có nghĩa là tại thời điểm đó, quá trình tìm kiếm theo chiều sâu bắt đầu từ  $u$  sẽ xây dựng nhánh cây DFS gốc  $u$ .
- ✿ Khi chuẩn bị thoát khỏi hàm để lùi về , ta nói đỉnh  $u$  được *duyet xong* (*finish*), có nghĩa là tại thời điểm đó, quá trình tìm kiếm theo chiều sâu từ  $u$  kết thúc.

Trong mô hình duyệt DFS ở trên, một biến đếm *Time* được dùng để xác định thời điểm thăm đến  $d_u$  và thời điểm duyệt xong  $f_u$  của mỗi đỉnh  $u$ . Thứ tự thăm đến và duyệt xong này có ý nghĩa rất quan trọng trong nhiều thuật toán có áp dụng DFS, chẳng hạn như các thuật toán tìm thành phần liên thông mạnh, thuật toán sắp xếp tô pô...

#### **Định lý 23-1**

Với hai đỉnh phân biệt  $u, v$ :

- ✿ Đỉnh  $v$  được thăm đến trong thời gian từ  $d_u$  đến  $f_u$ :  $d_v \in [d_u, f_u]$  nếu và chỉ nếu  $v$  là hậu duệ của  $u$  trên cây DFS.
- ✿ Đỉnh  $v$  được duyệt xong trong thời gian từ  $d_u$  đến  $f_u$ :  $f_v \in [d_u, f_u]$  nếu và chỉ nếu  $v$  là hậu duệ của  $u$  trên cây DFS.

#### **Chứng minh**

Bản chất của việc đỉnh  $v$  được thăm đến (hay duyệt xong) trong thời gian từ  $d_u$  đến  $f_u$  chính là hàm  $DFSVisit(v)$  được gọi (hay thoát) khi mà hàm  $DFSVisit(u)$  đã bắt đầu nhưng chưa kết thúc, nghĩa là hàm  $DFSVisit(v)$  được dây chuyền đệ

quy từ  $DFSVisit(u)$  gọi tới. Điều này chỉ ra rằng  $v$  nằm trong nhánh DFS gốc  $u$ , hay nói cách khác,  $v$  là hậu duệ của  $u$ .

### **Hệ quả**

Với hai đỉnh phân biệt  $(u, v)$  thì hai đoạn  $[d_u, f_u]$  và  $[d_v, f_v]$  hoặc rời nhau hoặc chứa nhau. Hai đoạn  $[d_u, f_u]$  và  $[d_v, f_v]$  chứa nhau nếu và chỉ nếu  $u$  và  $v$  có quan hệ tiền bối–hậu duệ.

### **Chứng minh**

Dễ thấy rằng nếu hai đoạn  $[d_u, f_u]$  và  $[d_v, f_v]$  không rời nhau thì hoặc  $d_u \in [d_v, f_v]$  hoặc  $d_v \in [d_u, f_u]$ , tức là hai đỉnh  $(u, v)$  có quan hệ tiền bối–hậu duệ, áp dụng Định lý 23-1, ta có ĐPCM.

### **Định lý 23-2**

Với hai đỉnh phân biệt  $u \neq v$  mà  $(u, v) \in E$  thì  $v$  phải được thăm đến trước khi  $u$  được duyệt xong:

$$(u, v) \in E \Rightarrow d_v < f_u$$

### **Chứng minh**

Đây là một tính chất quan trọng của thuật toán DFS. Hãy để ý hàm  $DFSVisit(u)$ , trước khi thoát (duyet xong  $u$ ), nó sẽ quét tất cả các đỉnh chưa thăm nối từ  $u$  và gọi đệ quy để thăm những đỉnh đó, tức là nếu  $(u, v) \in E$ ,  $v$  phải được thăm đến trước khi  $u$  được duyệt xong:  $d_v < f_u$ .

### **Định lý 23-3 (định lý đường đi trắng)**

Đỉnh  $v$  là hậu duệ thực sự của đỉnh  $u$  trong một cây DFS nếu và chỉ nếu tại thời điểm  $d_u$  mà thuật toán thăm tới đỉnh  $u$ , tồn tại một đường đi từ  $u$  tới  $v$  mà ngoại trừ đỉnh  $u$ , tất cả các đỉnh khác trên đường đi đều chưa được thăm.

### **Chứng minh**

“ $\Rightarrow$ ”

Nếu  $v$  là hậu duệ của  $u$ , ta xét đường đi từ  $u$  tới  $v$  dọc trên các cung trên cây DFS. Tất cả các đỉnh  $w$  nằm sau  $u$  trên đường đi này đều là hậu duệ của  $u$ , nên theo Định lý 23-1, ta có  $d_u < d_w$ , tức là vào thời điểm  $d_u$ , tất cả các đỉnh  $w$  đó đều chưa được thăm

“ $\Leftarrow$ ”

Nếu tại thời điểm  $d_u$ , tồn tại một đường đi từ  $u$  tới  $v$  mà tất cả các đỉnh khác  $u$  trên đường đi đều chưa được thăm, ta sẽ chứng minh rằng mọi đỉnh trên đường đi này đều là hậu duệ của  $u$ . Thật vậy, giả sử phản chứng rằng  $y$  là đỉnh đầu tiên trên đường đi này mà không phải hậu duệ của  $u$ , tức là tồn tại đỉnh  $x$  liền trước  $y$  trên đường đi là hậu duệ của  $u$ . Theo Định lý 23-2,  $y$  phải được thăm trước khi duyệt

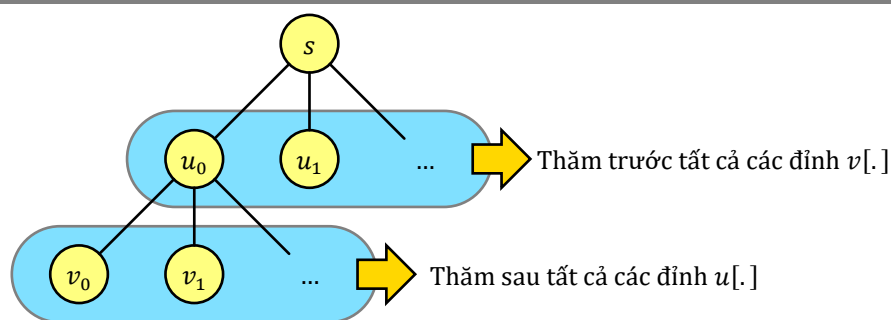
xong  $x$ :  $d_y < f_x$ ;  $x$  lại là hậu duệ của  $u$  nên theo Định lý 23-1, ta có  $f_x \leq f_u$ , vậy  $d_y < f_u$ . Mặt khác theo giả thiết rằng tại thời điểm  $d_u$  thì  $y$  chưa được thăm, tức là  $d_u < d_y$ , kết hợp lại ta có  $d_u < d_y < f_u$ , vậy thì  $y$  là hậu duệ của  $u$  theo Định lý 23-1, trái với giả thiết phản chứng.

Tên gọi “định lý đường đi trắng: white-path theorem” xuất phát từ cách trình bày thuật toán DFS bằng cơ chế tô màu đồ thị: Ban đầu các đỉnh được tô màu trắng, mỗi khi thăm đến một đỉnh thì đỉnh đó được tô màu xám và mỗi khi duyệt xong một đỉnh thì đỉnh đó được tô màu đen: Định lý khi đó có thể phát biểu: Điều kiện cần và đủ để đỉnh  $v$  là hậu duệ thực sự của đỉnh  $u$  trong một cây DFS là tại thời điểm đỉnh  $u$  được tô màu xám, tồn tại một đường đi từ  $u$  tới  $v$  mà ngoại trừ đỉnh  $u$ , tất cả các đỉnh khác trên đường đi đều có màu trắng.

## 23.4. Thuật toán tìm kiếm theo chiều rộng

### 23.4.1. Ý tưởng

Thực ra ta đã biết về thuật toán này trong khi tìm hiểu ứng dụng của cấu trúc dữ liệu hàng đợi (thuật toán loang). Tư tưởng của thuật toán tìm kiếm theo chiều rộng (Breadth-First Search – BFS) là “lập lịch” duyệt các đỉnh. Việc thăm một đỉnh sẽ lên lịch duyệt các đỉnh nối từ nó sao cho thứ tự duyệt là ưu tiên chiều rộng (đỉnh nào gần đỉnh xuất phát  $s$  hơn sẽ được duyệt trước). Đầu tiên ta thăm đỉnh  $s$ . Việc thăm đỉnh  $s$  sẽ phát sinh thứ tự thăm những đỉnh  $u_0, u_1, \dots$  nối từ  $s$  (những đỉnh gần  $s$  nhất). Tiếp theo ta thăm đỉnh  $u_0$ , khi thăm đỉnh  $u_0$  sẽ lại phát sinh yêu cầu thăm những đỉnh  $v_0, v_1, \dots$  nối từ  $u_0$ . Nhưng rõ ràng các đỉnh  $v[.]$  này “xa”  $s$  hơn những đỉnh  $u[.]$  nên chúng chỉ được thăm khi tất cả những đỉnh  $u[.]$  đã thăm. Tức là thứ tự duyệt đỉnh sẽ là:  $s, u_0, u_1, \dots, v_0, v_1, \dots$  (Hình 23-2).



Hình 23-2. Thứ tự thăm đỉnh của BFS

Thuật toán tìm kiếm theo chiều rộng sử dụng một danh sách để chứa những đỉnh đang “chờ” thăm. Tại mỗi bước, ta thăm một đỉnh đầu danh sách, loại nó ra khỏi danh sách và cho những đỉnh chưa “xếp hàng” kề với nó xếp hàng thêm vào cuối danh sách. Thuật toán sẽ kết thúc khi danh sách rỗng. Vì nguyên tắc vào trước ra

trước, danh sách chứa những đỉnh đang chờ thăm được tổ chức dưới dạng hàng đợi (queue).

```
Input → đồ thị G, đỉnh xuất phát s, đỉnh cần đến t;
Queue = {s};
for (∀u ∈ V) avail[u] = true; //Các đỉnh đều đánh dấu chưa xếp hàng
avail[s] = false; //Riêng đỉnh s được xếp hàng
do //Lặp chừng nào hàng đợi khác rỗng
{
    u = Queue.front(); Queue.pop(); //Lấy u khỏi hàng đợi
    for (∀v ∈ adj[u]) //Duyệt các đỉnh v nối từ u
        if (avail[v]) //Nếu v chưa xếp hàng
        {
            Queue.push(v); //Cho v xếp hàng trong hàng đợi
            avail[v] = false;
            trace[v] = u; //Lưu vết: đỉnh liền trước v trên đường đi từ s tới v là đỉnh u
        }
}
while (Queue ≠ ∅);
if (!avail[t]) //Từ s có đường tới t
    «Truy theo vết từ t để tìm đường đi từ s tới t»;
```

### 23.4.2. Cài đặt

#### 🌀 BFS.CPP ✓ Tìm đường bằng BFS 🌀

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
const int maxN = 1e6;

int n, m, s, t, trace[maxN];
bool avail[maxN];
vector<int> adj[maxN];

void Enter() //Nhập dữ liệu
{
    cin >> n >> m >> s >> t;
    while (m-- > 0)
    {
        int u, v;
        cin >> u >> v; //Đọc một cạnh (u, v)
        adj[u].push_back (v); //Đưa v vào danh sách kề của u
    }
}
```

```
void BFS() //Thuật toán tìm kiếm theo chiều rộng
{
    queue<int> Queue({s}); //Hàng đợi ban đầu chỉ gồm một đỉnh s
    fill (avail, avail + n, true); //Khởi tạo các đỉnh đều chưa thăm
    avail[s] = false; //Riêng đỉnh s đã thăm
    do
    {
        int u = Queue.front(); Queue.pop(); //Lấy u từ hàng đợi
        cout << u << ", ";
        for (int v: adj[u]) //Xét mọi đỉnh v nối từ u
            if (avail[v]) //Nếu v chưa thăm
            {
                trace[v] = u; //Lưu vết đường đi s → ... → u → v
                Queue.push(v); //Đẩy v vào hàng đợi
                avail[v] = false;
            }
    }
    while (!Queue.empty());
}

void PrintPath() //In đường đi
{
    if (avail[t]) //t chưa thăm: không có đường từ s tới t
        cout << "There's no path from " << s << " to " << t << '\n';
    else
    {
        cout << "The path from " << s << " to " << t << ":\n";
        for (int u = t; u != s; u = trace[u]) //Truy vết ngược từ t về s
            cout << u << " <- ";
        cout << s << "\n";
    }
}

int main()
{
    Enter();
    cout << "Reachable vertices from " << s << ":\n";
    BFS(); //Thuật toán tìm kiếm theo chiều rộng
    cout << "\n";
    PrintPath();
}
```

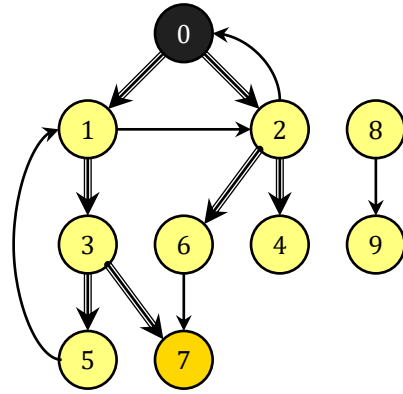
Tương tự như thuật toán tìm kiếm theo chiều sâu, ta có thể dùng mảng *trace*[0 ... *n*) kèm luôn chức năng đánh dấu.

### 23.4.3. Một vài tính chất của BFS

#### ✧ Cây BFS

Thuật toán BFS luôn trả về đường đi qua ít cạnh nhất trong số tất cả các đường đi từ *s* tới *t*. Nếu các danh sách kề được sắp xếp theo thứ tự tăng dần thì thuật toán BFS sẽ trả về đường đi có thứ tự từ điển nhỏ nhất trong số những đường đi qua ít cạnh nhất.

Quá trình tìm kiếm theo chiều rộng cho ta một cây DFS gốc  $s$ . Quan hệ cha-con trên cây được định nghĩa là: nếu từ đỉnh  $u$  tới thăm đỉnh  $v$  (khi đỉnh  $u$  được rút ra khỏi hàng đợi thì  $v$  là một trong số những đỉnh chưa thăm nối từ  $u$  được xét đến và đẩy vào hàng đợi) thì  $u$  là nút cha của nút  $v$ . Hình bên là đồ thị với danh sách kề của mỗi đỉnh được sắp xếp tăng dần và cây BFS tương ứng với đỉnh xuất phát  $s = 0$ . Chú ý rằng cấu trúc của cây BFS cũng phụ thuộc vào thứ tự trong các danh sách kề.



### \* Mô hình duyệt đồ thị theo BFS

Tương tự như thuật toán DFS, mô hình tổng quát của thuật toán BFS cũng dùng để xác định một thứ tự trên các đỉnh của đồ thị:

```
void BFSVisit( $s \in V$ )
{
    Queue = ( $s$ ); //hàng đợi chỉ gồm một đỉnh  $s$ 
     $d[s] = ++Time$ ; //d[u] = thời điểm u được thăm đến, cũng là đánh dấu  $\neq \emptyset$ 
    do
    {
         $u = \text{Queue.front}(); \text{Queue.pop}();$ 
         $f[u] = ++Time$ ;
        Output  $\leftarrow u$ ;
        for ( $\text{int } v: \text{adj}[u]$ ) //duyet mọi đỉnh  $v$  nối từ  $u$ 
            if ( $d[v] == \emptyset$ ) //nếu  $v$  chưa thăm
            {
                Queue.push( $v$ );
                 $d[v] = ++Time$ ; //Ghi nhận thời điểm thăm đến  $u$ , cũng là đánh dấu  $d[v] \neq \emptyset$ 
            }
    }
    while (Queue  $\neq \emptyset$ );
}
```

```
Input  $\rightarrow$  đồ thị  $G$ ;
for ( $\forall u \in V$ )  $d[u] = \emptyset$ ; //Các đỉnh đều chưa thăm:  $d[.] = \emptyset$ 
Time = 0;
for ( $\forall u \in V$ ) //Xét mọi điểm chưa thăm
    if ( $d[u] == \emptyset$ ) BFSVisit( $u$ );
```

Thời gian thực hiện giải thuật của BFS tương tự như đối với DFS, bằng  $\Theta(|V| + |E|)$  nếu đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, bằng  $\Theta(|V|^2)$  nếu đồ thị được biểu diễn bằng ma trận kề, và bằng  $\Theta(|V||E|)$  nếu đồ thị được biểu diễn bằng danh sách cạnh.

### \* Thứ tự thăm đến và duyệt xong

Tương tự như thuật toán DFS, đối với thuật toán BFS người ta cũng quan tâm tới thứ tự thăm đến và duyệt xong: Khi một đỉnh được đẩy vào hàng đợi, ta nói đỉnh đó được thăm đến (được thăm) và khi một đỉnh được lấy ra khỏi hàng đợi, ta nói



đỉnh đó được duyệt xong. Trong mô hình cài đặt trên, mỗi đỉnh  $u$  sẽ tương ứng với thời điểm thăm đến  $d_u$  và thời điểm duyệt xong  $f_u$

Vì cách hoạt động của hàng đợi: đỉnh nào thăm đến trước sẽ phải duyệt xong trước, chính vì vậy, việc liệt kê các đỉnh có thể thực hiện khi chúng được thăm đến hay duyệt xong mà không ảnh hưởng tới thứ tự. Như cách cài đặt ở trên, mỗi đỉnh được đánh dấu mỗi khi đỉnh đó được thăm đến và được liệt kê mỗi khi nó được duyệt xong.

Có thể sửa đổi một chút mô hình cài đặt bằng cách thay cơ chế đánh dấu thăm đến/chưa thăm đến bằng duyệt xong/chưa duyệt xong:

```
Input → đồ thị G;
for (∀u ∈ V) avail[u] = true; //Các đỉnh đều chưa duyệt xong
Queue = ∅;
for (∀u ∈ V) Queue.push(u); //Khởi tạo hàng đợi chứa tất cả các đỉnh
do
{
    u = Queue.front(); Queue.pop(); //Lấy u khỏi hàng đợi
    if (!avail[u]) continue; //Nếu u đã duyệt xong, bỏ qua
    Output ← u; //Liệt kê u
    avail[u] = false; //Đánh dấu u đã duyệt xong
    for (∀v ∈ adj[u]) //Xét các đỉnh v kề u
        if (avail[v]) //Nếu v chưa duyệt xong
        {
            trace[v] = u; //Lưu vết đường đi
            Queue.push(v); //Đẩy v vào hàng đợi
        }
}
while (Queue ≠ ∅);
```

Kết quả của hai cách cài đặt không khác nhau, sự khác biệt chỉ nằm ở lượng bộ nhớ cần sử dụng cho hàng đợi *Queue*: Ở cách cài đặt thứ nhất, do cơ chế đánh dấu thăm đến/chưa thăm đến, mỗi đỉnh sẽ được đưa vào *queue* đúng một lần và lấy ra khỏi *queue* đúng một lần nên chúng ta cần không quá  $n$  ô nhớ để chứa các phần tử của *queue*. Ở cách cài đặt thứ hai, có thể có nhiều hơn  $n$  đỉnh đứng xếp hàng trong *queue* vì một đỉnh  $v$  có thể được đẩy vào *queue* tới  $1 + \deg^+(v)$  lần (tính cả bước khởi tạo hàng đợi chứa tất cả các đỉnh), có nghĩa là khi tổ chức dữ liệu, chúng ta phải dự trù  $\sum_{v \in V} (1 + \deg^+(v)) = m + n$  ô nhớ cho *queue*. Con số này đối với đồ thị vô hướng là  $2m + n$  ô nhớ.

Rõ ràng đối với BFS, cách cài đặt như ban đầu sẽ tiết kiệm bộ nhớ hơn. Nhưng có điểm đặc biệt là trong mô hình trên, nếu thay cấu trúc hàng đợi bởi cấu trúc stack ta sẽ được thứ tự duyệt đỉnh DFS (thực ra để ra kết quả giống hệt DFS cần lật ngược thứ tự đỉnh trong các danh sách kề nữa nhưng chỉ là kỹ thuật nhỏ). Đây chính là phương pháp khử đệ quy của DFS để cài đặt thuật toán trên các ngôn ngữ không cho phép đệ quy.

### Bài tập 23-1

Viết chương trình cài đặt thuật toán DFS không đệ quy.

### Bài tập 23-2

Xét đồ thị có hướng  $G = (V, E)$ , dùng thuật toán DFS duyệt đồ thị  $G$ . Cho một phản ví dụ để chứng minh giả thuyết sau là sai: Nếu từ đỉnh  $u$  có đường đi tới đỉnh  $v$  và  $u$  được thăm đến trước  $v$ , thì  $v$  nằm trong nhánh DFS gốc  $u$ .

### Bài tập 23-3

Cho đồ thị vô hướng  $G = (V, E)$ , tìm thuật toán  $O(|V|)$  để phát hiện một chu trình đơn trong  $G$ .

### Bài tập 23-4

Cho đồ thị có hướng  $G = (V, E)$  có  $n$  đỉnh, và mỗi đỉnh  $i$  được gán một nhãn là số nguyên  $a_i$ , tập cung  $E$  của đồ thị được định nghĩa là  $(u, v) \in E \Leftrightarrow a_u \geq a_v$ . Giả sử rằng thuật toán DFS được sử dụng để duyệt đồ thị, hãy khảo sát tính chất của dãy các nhãn nếu ta xếp các đỉnh theo thứ tự từ đỉnh duyệt xong đầu tiên đến đỉnh duyệt xong sau cùng.

### Bài tập 23-5

Mê cung hình chữ nhật kích thước  $m \times n$  gồm các ô vuông đơn vị. Trên mỗi ô ghi một trong ba ký tự:

- ✿ O: Nếu ô đó an toàn
- ✿ X: Nếu ô đó có cạm bẫy
- ✿ E: Nếu là ô có một nhà thám hiểm đang đứng.

Duy nhất chỉ có 1 ô ghi chữ E. Nhà thám hiểm có thể từ một ô đi sang một trong số các ô chung cạnh với ô đang đứng. Một cách đi thoát khỏi mê cung là một hành trình đi qua các ô an toàn ra một ô biên. Hãy chỉ giúp cho nhà thám hiểm một hành trình thoát ra khỏi mê cung đi qua ít ô nhất. Yêu cầu thuật toán có độ phức tạp  $O(mn)$

## Chương 24. Tính liên thông của đồ thị

### 24.1. Định nghĩa

#### 24.1.1. Tính liên thông trên đồ thị vô hướng

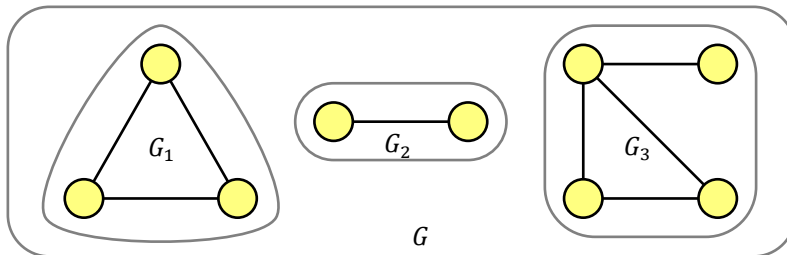
Đồ thị vô hướng  $G = (V, E)$  được gọi là *liên thông* (*connected*) nếu giữa mọi cặp đỉnh của  $G$  luôn tồn tại đường đi. Đồ thị chỉ gồm một đỉnh duy nhất cũng được coi là đồ thị liên thông.

Cho đồ thị vô hướng  $G = (V, E)$  và  $U$  là một tập con khác rỗng của tập đỉnh  $V$ . Ta nói  $U$  là một *thành phần liên thông* (*connected component*) của  $G$  nếu:

- Đồ thị  $G$  hạn chế trên tập  $U$ :  $G_U = (U, E_U)$  là đồ thị liên thông.
- Không tồn tại một tập  $W$  chứa  $U$  mà đồ thị  $G$  hạn chế trên  $W$  là liên thông (tính tối đại của  $U$ ).

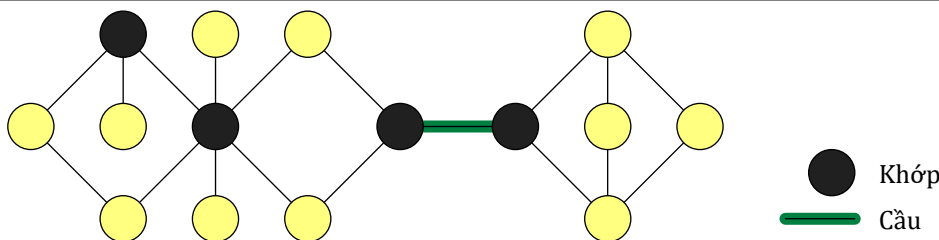
(Ta cũng đồng nhất khái niệm thành phần liên thông  $U$  với thành phần liên thông  $G_U = (U, E_U)$ ).

Một đồ thị liên thông chỉ có một thành phần liên thông là chính nó. Một đồ thị không liên thông sẽ có nhiều hơn 1 thành phần liên thông. Hình 24-1 là ví dụ về đồ thị  $G$  và các thành phần liên thông  $G_1, G_2, G_3$  của nó.



Hình 24-1. Đồ thị và các thành phần liên thông

Đôi khi, việc xóa đi một đỉnh và tất cả các cạnh liên thuộc với nó sẽ tạo ra một đồ thị con mới có nhiều thành phần liên thông hơn đồ thị ban đầu, các đỉnh như thế gọi là *đỉnh cắt* (*cut vertices*) hay *nút khớp* (*articulation nodes*). Hoàn toàn tương tự, những cạnh mà khi ta bỏ nó đi sẽ tạo ra một đồ thị có nhiều thành phần liên thông hơn so với đồ thị ban đầu được gọi là *cạnh cắt* (*cut edges*) hay *cầu* (*bridges*).

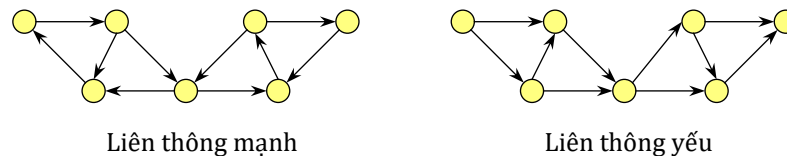


Hình 24-2. Khớp và cầu

### 24.1.2. Tính liên thông trên đồ thị có hướng

Cho đồ thị có hướng  $G = (V, E)$ , có hai khái niệm về tính liên thông của đồ thị có hướng tùy theo chúng ta có quan tâm tới hướng của các cung không.

$G$  gọi là *liên thông mạnh* (*strongly connected*) nếu luôn tồn tại đường đi (theo các cung định hướng) giữa hai đỉnh bất kỳ của đồ thị,  $G$  gọi là *liên thông yếu* (*weakly connected*) nếu phiên bản vô hướng của nó là đồ thị liên thông.



Hình 24-3. Liên thông mạnh và liên thông yếu

## 24.2. Bài toán xác định các thành phần liên thông

Một bài toán quan trọng trong lý thuyết đồ thị là bài toán kiểm tra tính liên thông của đồ thị vô hướng hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông của đồ thị vô hướng.

Để liệt kê các thành phần liên thông của đồ thị vô hướng  $G = (V, E)$ , phương pháp cơ bản nhất là bắt đầu từ một đỉnh xuất phát bất kỳ, ta liệt kê những đỉnh đến được từ đỉnh đó vào một thành phần liên thông, liệt kê xong đỉnh nào loại bỏ luôn đỉnh đó khỏi đồ thị. Quá trình lặp lại với một đỉnh xuất phát mới chưa bị loại bỏ cho tới khi tập đỉnh của đồ thị trở thành  $\emptyset$ . Việc loại bỏ đỉnh của đồ thị có thể thực hiện bằng cơ chế đánh dấu những đỉnh bị loại:

```
void Scan( $u \in V$ )
{
    «Dùng BFS hoặc DFS liệt kê và đánh dấu thăm những đỉnh có thể đến được từ  $u$ »;
}

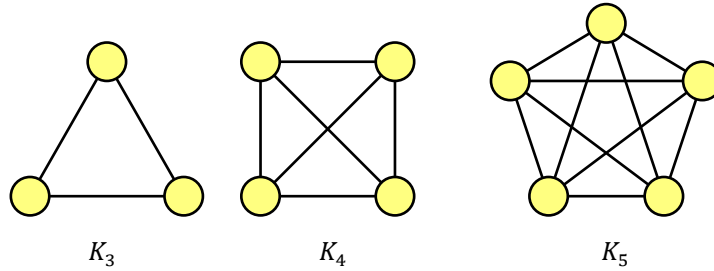
for ( $\forall u \in V$ )
    «Đánh dấu  $v$  chưa thăm»;
for ( $\forall u \in V$ )
    if « $u$  chưa thăm» then
    {
        Output  $\leftarrow$  «Các đỉnh của một thành phần liên thông:»;
        Scan( $u$ );
    }
```

Thời gian thực hiện giải thuật đúng bằng thời gian thực hiện giải thuật duyệt đồ thị (DFS hoặc BFS).

## 24.3. Bao đóng của đồ thị vô hướng

### 24.3.1. Định nghĩa

Đồ thị đầy đủ với  $n$  đỉnh, ký hiệu  $K_n$ , là một đơn đồ thị vô hướng mà giữa hai đỉnh bất kỳ của nó đều có 1 cạnh nối. Đồ thị đầy đủ  $K_n$  có đúng  $C_n^k = \frac{n(n-1)}{2}$  cạnh, bậc của mọi đỉnh đều là  $n - 1$ .



Hình 24-4. Đồ thị đầy đủ

### 24.3.2. Bao đóng đồ thị

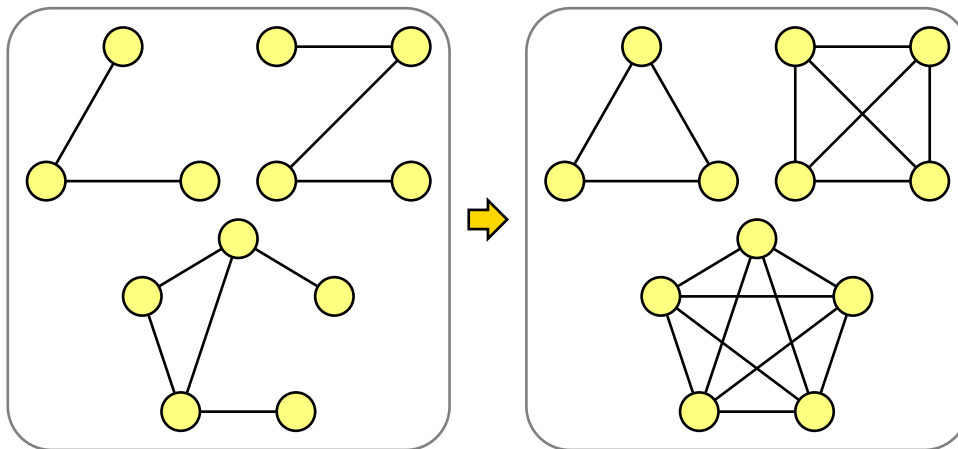
Với đồ thị  $G = (V, E)$ , người ta xây dựng đồ thị  $\overline{G} = (V, \overline{E})$  cũng gồm những đỉnh của  $G$  còn các cạnh xây dựng như sau:

Giữa hai đỉnh  $u, v$  của  $\overline{G}$  có cạnh nối  $\Leftrightarrow$  Giữa hai đỉnh  $u, v$  của  $G$  có đường đi

Đồ thị  $\overline{G}$  xây dựng như vậy được gọi là bao đóng của đồ thị  $G$ .

Từ định nghĩa của đồ thị đầy đủ, và đồ thị liên thông, ta suy ra:

- ✿ Một đơn đồ thị vô hướng là liên thông nếu và chỉ nếu bao đóng của nó là đồ thị đầy đủ
- ✿ Một đơn đồ thị vô hướng có  $k$  thành phần liên thông nếu và chỉ nếu bao đóng của nó có  $k$  thành phần liên thông là đồ thị đầy đủ.



Hình 24-5. Đồ thị vô hướng và bao đóng của nó

Bởi việc kiểm tra một đơn đồ thị có phải đồ thị đầy đủ hay không có thể thực hiện khá dễ dàng (đếm số cạnh chẳng hạn) nên người ta nảy ra ý tưởng có thể kiểm tra tính liên thông của đồ thị thông qua việc kiểm tra tính đầy đủ của bao đóng. Vấn đề đặt ra là phải có thuật toán xây dựng bao đóng của một đồ thị cho trước.

### 24.3.3. Thuật toán Warshall

Thuật toán Warshall **Invalid source specified.** – gọi theo tên của Stephen Warshall, người đã mô tả thuật toán này vào năm 1960, đôi khi còn được gọi là thuật toán Roy-Warshall vì Bernard Roy cũng đã mô tả thuật toán này vào năm 1959. Thuật toán đó có thể mô tả rất gọn:

Giả sử đơn đồ thị vô hướng  $G = (V, E)$  có  $n$  đỉnh đánh số từ 0 tới  $n - 1$ , thuật toán Warshall xét tất cả các đỉnh  $k \in V$ , với mỗi đỉnh  $k$  được xét, thuật toán lại xét tiếp tất cả các cặp đỉnh  $(i, j)$ : nếu đồ thị có cạnh  $(i, k)$  và cạnh  $(k, j)$  thì ta tự nối thêm cạnh  $(i, j)$  nếu nó chưa có. Tư tưởng này dựa trên một quan sát đơn giản như sau: Nếu từ  $i$  có đường đi tới  $k$  và từ  $k$  lại có đường đi tới  $j$  thì chắc chắn từ  $i$  sẽ có đường đi tới  $j$ .

Với đơn đồ thị được biểu diễn bởi ma trận kề  $A = \{a_{ij}\}_{n \times n}$  trong đó:

$$a_{ij} = \text{true} \Leftrightarrow (i, j) \in E$$

Thuật toán Warshall tính lại ma trận  $A$  để nó trở thành ma trận kề của bao đóng theo cách sau:

```
for (k = 0; k < n; ++k)
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      a[i][j] |= a[i][k] && a[k][j];
```

Việc chứng minh tính đúng đắn của thuật toán đòi hỏi phải lật lại các lý thuyết về bao đóng bắc cầu và quan hệ liên thông, ta sẽ không trình bày ở đây mà sẽ trình bày rõ hơn về cơ chế hoạt động trong thuật toán Floyd tìm đường đi ngắn nhất giữa mọi cặp đỉnh. Tuy nhiên có ba điểm cần lưu ý:

- ✱ Thuật toán Warshall hoàn toàn có thể áp dụng để tính bao đóng  $\overline{G} = (V, \overline{E})$  của đồ thị có hướng  $G = (V, E)$  với định nghĩa bao đóng tương tự như trên đồ thị vô hướng:  $(u, v) \in \overline{E} \Leftrightarrow u \rightsquigarrow v$  trên  $G$
- ✱ Việc đảo lộn thứ tự ba vòng lặp for có thể cho một thuật toán sai. Thực ra có thể đảo thứ tự hai vòng lặp “for (i...” và “for (j...” nhưng vòng lặp “for (k...” chắc chắn phải nằm ở ngoài cùng.

- Tuy thuật toán Warshall rất dễ cài đặt nhưng đòi hỏi thời gian thực hiện giải thuật khá lớn:  $\Theta(n^3)$ . Chính vì vậy thuật toán Warshall chỉ nên sử dụng khi thực sự cần tới bao đóng của đồ thị, còn nếu chỉ cần liệt kê các thành phần liên thông thì các thuật toán tìm kiếm trên đồ thị tỏ ra hiệu quả hơn nhiều.

Dưới đây, ta sẽ thử cài đặt thuật toán Warshall tìm bao đóng của đơn đồ thị vô hướng sau đó đếm số thành phần liên thông của đồ thị:

Việc cài đặt thuật toán sẽ qua những bước sau:

- Dùng ma trận kề  $A$  biểu diễn đồ thị, quy ước rằng  $a_{ii} = \text{true}, \forall i$
- Dùng thuật toán Warshall tìm bao đóng, khi đó  $A$  là ma trận kề của đồ thị bao đóng
- Dựa vào ma trận kề  $A$ , đỉnh 0 và những đỉnh kề với nó sẽ thuộc thành phần liên thông thứ nhất; với đỉnh  $u$  nào đó không kề với đỉnh 0, thì  $u$  cùng với những đỉnh kề nó sẽ thuộc thành phần liên thông thứ hai; với đỉnh  $v$  nào đó không kề với cả đỉnh 0 và đỉnh  $u$ , thì  $v$  cùng với những đỉnh kề nó sẽ thuộc thành phần liên thông thứ ba v.v...

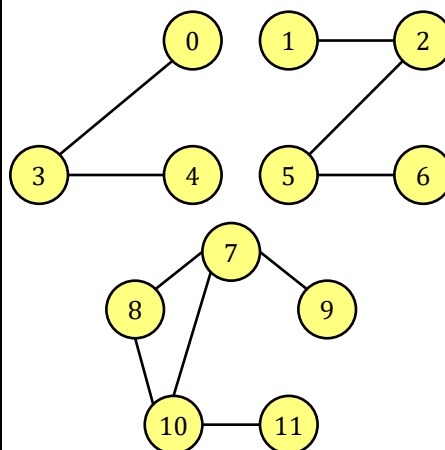
### Input

- Dòng 1: Chứa số đỉnh  $n \leq 200$  và số cạnh  $m$  của đồ thị
- $m$  dòng tiếp theo, mỗi dòng chứa một cặp số  $u$  và  $v$  tương ứng với một cạnh  $(u, v)$

### Output

Liệt kê các thành phần liên thông của đồ thị

Sample Input	Sample Output
12 10 0 3 1 2 2 5 3 4 5 6 7 8 7 9 7 10 8 10 10 11	Connected Component: 0, 3, 4, Connected Component: 1, 2, 5, 6, Connected Component: 7, 8, 9, 10, 11,



☞ WARSHALL.CPP ✓ Thuật toán Warshall ☞

```
#include <iostream>
using namespace std;
const int maxN = 200;
int n, m;
bool a[maxN][maxN]; //Ma trận kề
bool avail[maxN]; //Đánh dấu đỉnh chưa được liệt kê

void ReadInput() //Nhập dữ liệu
{
    cin >> n >> m;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            a[i][j] = i == j; //a[i][i] = true,  $\forall i$ 
    while (m-- > 0)
    {
        int u, v;
        cin >> u >> v;
        a[u][v] = a[v][u] = true; //Đồ thị vô hướng
    }
}

void ComputeTransitiveClosure() //Thuật toán Warshall
{
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                a[i][j] |= a[i][k] && a[k][j];
}

void Print()
{
    fill(avail, avail + n, true); //avail[.] = true: đỉnh chưa được liệt kê
    for (int u = 0; u < n; ++u) //Xét mọi đỉnh
        if (avail[u]) //Gặp đỉnh chưa được liệt kê vào thành phần liên thông nào
        { //Liệt kê thành phần liên thông chứa u
            cout << "Connected Component: \n";
            for (int v = 0; v < n; ++v)
                if (a[u][v]) //Thành phần liên thông gồm những đỉnh kề u trên bao đóng
                {
                    cout << v << ", ";
                    avail[v] = false; //Liệt kê đỉnh nào đánh dấu đỉnh đó
                }
            cout << "\n";
        }
}

int main()
{
    ReadInput();
    ComputeTransitiveClosure();
    Print();
}
```

#### 24.4. Bài toán xác định các thành phần liên thông mạnh

Đối với đồ thị có hướng, người ta quan tâm đến bài toán kiểm tra tính liên thông mạnh, hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông mạnh của đồ



thị có hướng. Các thuật toán tìm kiếm thành phần liên thông mạnh hiệu quả hiện nay đều dựa trên thuật toán tìm kiếm theo chiều sâu Depth-First Search.

Ta sẽ khảo sát và cài đặt hai thuật toán liệt kê thành phần liên thông mạnh với khuôn dạng Input/Output như sau:

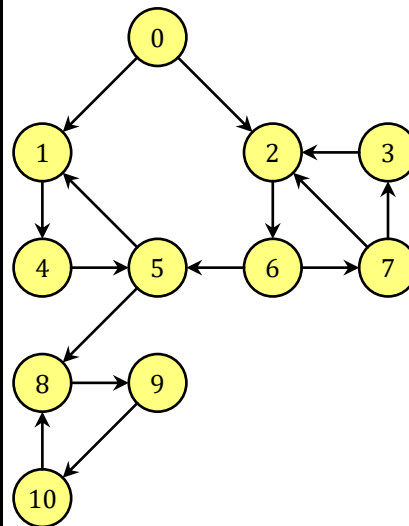
### Input

- ✿ Dòng đầu: Chứa số đỉnh  $n \leq 10^6$  và số cung  $m \leq 10^6$  của đồ thị.
- ✿  $m$  dòng tiếp theo, mỗi dòng chứa hai số nguyên  $u, v$  tương ứng với một cung  $(u, v)$  của đồ thị.

### Output

Các thành phần liên thông mạnh.

Sample Input	Sample Output
11 15	Strongly connected component:
0 1	10, 9, 8,
0 2	Strongly connected component:
1 4	5, 4, 1,
2 6	Strongly connected component:
3 2	3, 7, 6, 2,
4 5	Strongly connected component:
5 1	0,
5 8	
6 5	
6 7	
7 2	
7 3	
8 9	
9 10	
10 8	



#### 24.4.1. Phân tích

Xét thuật toán tìm kiếm theo chiều sâu:

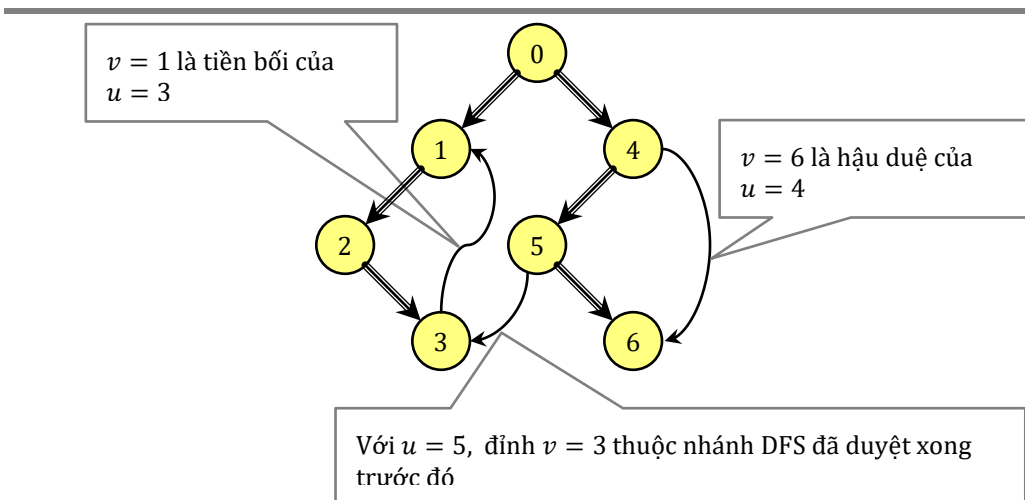
```
void DFSVisit(u ∈ V)
{
    avail[u] = false; //Đánh dấu u đã thăm
    for (∀v: (u, v) ∈ E)
        if (avail[v]) DFSVisit(v);
}
```

```
for (∀u ∈ V) avail[u] = true;
for (∀u ∈ V)
    if (avail[u]) DFSVisit(u);
```

Ta biết rằng thuật toán tìm kiếm theo chiều sâu xây dựng một rừng các cây DFS theo quan hệ: Nếu hàm  $DFSVisit(u)$  gọi hàm  $DFSVisit(v)$  thì  $u$  là cha của  $v$  trên cây DFS.

Để ý hàm  $DFSVisit(u)$ . Hàm này xét tất cả những đỉnh  $v$  nối từ  $u$ :

- ✿ Nếu  $v$  chưa được thăm thì đi theo cung đó thăm  $v$ , tức là cho đỉnh  $v$  trở thành con của đỉnh  $u$  trong cây tìm kiếm DFS, cung  $(u, v)$  khi đó được gọi là cung DFS (Tree edge).
- ✿ Nếu  $v$  đã thăm thì có ba khả năng xảy ra đối với vị trí của  $u$  và  $v$  trong cây tìm kiếm DFS:
  - ✿  $v$  là tiền bối (ancestor) của  $u$ , tức là  $v$  được thăm trước  $u$  và hàm  $DFSVisit(u)$  do đây chuyển đệ quy từ hàm  $DFSVisit(v)$  gọi tới. Cung  $(u, v)$  khi đó được gọi là cung ngược (back edge)
  - ✿  $v$  là hậu duệ (descendant) của  $u$ , tức là  $u$  được thăm trước  $v$ , nhưng hàm  $DFSVisit(u)$  sau khi tiến đệ quy theo một hướng khác đã gọi  $DFSVisit(v)$  rồi. Nên khi đây chuyển đệ quy lùi lại về hàm  $DFSVisit(u)$  sẽ thấy  $v$  là đã thăm nên không thăm lại nữa. Cung  $(u, v)$  khi đó gọi là cung xuôi (forward edge).
  - ✿  $v$  thuộc một nhánh DFS đã duyệt trước đó, cung  $(u, v)$  khi đó gọi là cung chéo (cross edge)



Hình 24-6. Ba loại cung ngoài cây DFS

Ta nhận thấy một đặc điểm của thuật toán tìm kiếm theo chiều sâu, thuật toán không chỉ duyệt qua các đỉnh, nó còn duyệt qua tất cả những cung nữa. Ngoài những cung nằm trên cây DFS, những cung còn lại có thể chia làm ba loại: cung ngược, cung xuôi, cung chéo.

Chú ý rằng việc phân chia các loại cung là phụ thuộc vào cấu trúc cây DFS, cấu trúc cây DFS lại phụ thuộc vào thứ tự duyệt danh sách kề của các đỉnh. Hình 24-6 là ví

dự về các loại cung trên cây DFS khi danh sách kề của các đỉnh được sắp xếp theo thứ tự tăng dần của số hiệu đỉnh.

#### 24.4.2. Cây DFS và các thành phần liên thông mạnh

##### **Bổ đề 24-1**

Nếu  $x$  và  $y$  là hai đỉnh thuộc thành phần liên thông mạnh  $C$  thì với mọi đường đi từ  $x$  tới  $y$  cũng như từ  $y$  tới  $x$ . Tất cả đỉnh trung gian trên đường đi đều phải thuộc  $C$ .

##### **Chứng minh**

Gọi  $v$  là một đỉnh trên đường đi  $x \rightsquigarrow y$ . Vì  $x$  và  $y$  thuộc cùng một thành phần liên thông mạnh nên cũng tồn tại một đường đi  $y \rightsquigarrow x$ . Nối tiếp hai đường đi này lại ta sẽ được một chu trình đi từ  $x$  tới  $y$  rồi quay lại  $x$  trong đó  $v$  là một đỉnh nằm trên chu trình. Điều này chỉ ra rằng nếu đi dọc theo chu trình ta có thể đi từ  $x$  tới  $v$  cũng như từ  $v$  tới  $x$ , nghĩa là  $v$  và  $x$  thuộc cùng một thành phần liên thông mạnh.

##### **Bổ đề 24-2**

Với một thành phần liên thông mạnh  $C$  bất kỳ, sẽ tồn tại duy nhất một đỉnh  $r \in C$  sao cho mọi đỉnh của  $C$  đều thuộc nhánh cây DFS gốc  $r$ .

##### **Chứng minh**

Trong số các đỉnh của  $C$ , chọn  $r$  là đỉnh được thăm đầu tiên theo thuật toán DFS. Ta sẽ chứng minh  $C$  nằm trong nhánh DFS gốc  $r$ . Thật vậy: với một đỉnh  $v$  bất kỳ của  $C$ , do  $C$  liên thông mạnh nên phải tồn tại một đường đi từ  $r$  tới  $v$ :

$$\langle r = x_0, x_1, \dots, x_k = v \rangle$$

Từ Bổ đề 24-1, tất cả các đỉnh  $x_1, x_2, \dots, x_k$  đều thuộc  $C$ . Ngoài ra do cách chọn  $r$  là đỉnh được thăm đầu tiên nên theo Định lý 23-3 (định lý đường đi trắng), tất cả các đỉnh  $x_1, x_2, \dots, x_k = v$  phải là hậu duệ của  $r$  tức là chúng đều thuộc nhánh DFS gốc  $r$ .

Đỉnh  $r$  trong chứng minh định lý – đỉnh thăm trước tất cả các đỉnh khác trong  $C$  – gọi là chốt của thành phần liên thông mạnh  $C$ . Xét về vị trí trên cây DFS, mỗi thành phần liên thông mạnh có duy nhất một chốt, chính là đỉnh nằm cao nhất so với các đỉnh khác thuộc thành phần đó, hay nói cách khác: là tiền bối của tất cả các đỉnh thuộc thành phần đó.

##### **Bổ đề 24-3**

Với một chốt  $r$  không là tiền bối của bất kỳ chốt nào khác thì các đỉnh thuộc nhánh DFS gốc  $r$  chính là thành phần liên thông mạnh chứa  $r$ .

### Chứng minh:

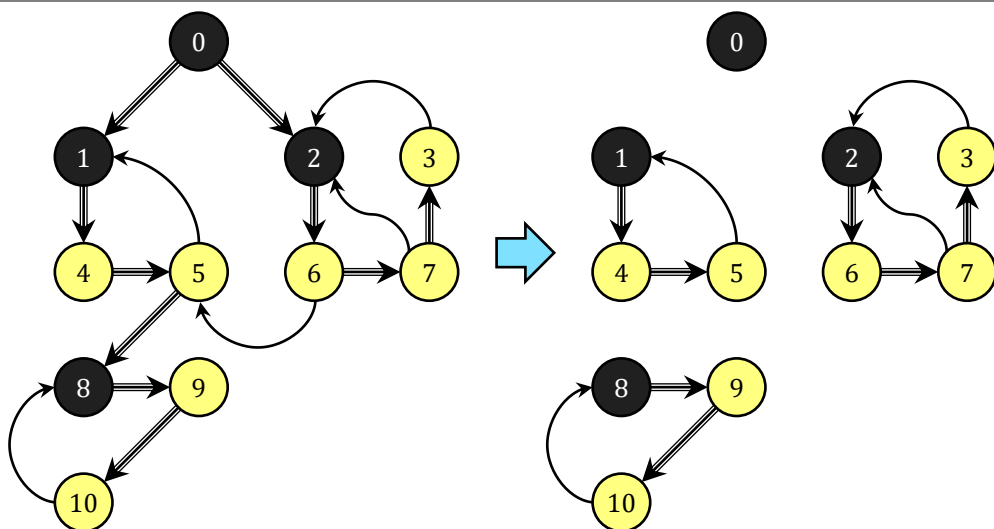
Với mọi đỉnh  $v$  nằm trong nhánh DFS gốc  $r$ , gọi  $s$  là chốt của thành phần liên thông mạnh chứa  $v$ . Ta sẽ chứng minh  $r = s$ . Thật vậy, theo Bổ đề 24-2,  $v$  phải nằm trong nhánh DFS gốc  $s$ . Vậy  $v$  nằm trong cả nhánh DFS gốc  $r$  và nhánh DFS gốc  $s$ , nghĩa là  $r$  và  $s$  có quan hệ tiền bối-hậu duệ. Theo giả thiết  $r$  không là tiền bối của bất kỳ chốt nào khác nên  $r$  phải là hậu duệ của  $s$ . Ta có đường đi  $s \rightsquigarrow r \rightsquigarrow v$ , mà  $s$  và  $v$  thuộc cùng một thành phần liên thông mạnh nên theo Bổ đề 24-1,  $r$  cũng phải thuộc thành phần liên thông mạnh đó. Mỗi thành phần liên thông mạnh có duy nhất một chốt mà  $r$  và  $s$  đều là chốt nên  $r = s$ .

Bổ đề 24-2 khẳng định thành phần liên thông mạnh chứa  $r$  nằm trong nhánh DFS gốc  $r$ , theo chứng minh trên ta lại có: Mọi đỉnh trong nhánh DFS gốc  $r$  nằm trong thành phần liên thông mạnh chứa  $r$ . Kết hợp lại được: Nhánh DFS gốc  $r$  chính là thành phần liên thông mạnh chứa  $r$ .

### 24.4.3. Thuật toán Tarjan

#### \* Ý tưởng

Thuật toán Tarjan [15] có thể phát biểu như sau: Chọn  $r$  là chốt không là tiền bối của một chốt nào khác, chọn lấy thành phần liên thông mạnh thứ nhất là nhánh DFS gốc  $r$ . Sau đó loại bỏ nhánh DFS gốc  $r$  ra khỏi cây DFS, lại tìm thấy một chốt  $s$  khác mà nhánh DFS gốc  $s$  không chứa chốt nào khác, lại chọn lấy thành phần liên thông mạnh thứ hai là nhánh DFS gốc  $s$ ... Tương tự như vậy cho thành phần liên thông mạnh thứ ba, thứ tư, v.v... Có thể hình dung thuật toán Tarjan “bẻ” cây DFS bằng cách cắt bỏ các cung nối tới chốt để được các nhánh rời rạc, mỗi nhánh là một thành phần liên thông mạnh.



Hình 24-7. Thuật toán Tarjan “bẻ” cây DFS

Mô hình cài đặt của thuật toán Tarjan:

```
void DFSVisit(u ∈ V)
{
    avail[u] = false; //Đánh dấu u đã thăm
    for (∀v: (u, v) ∈ E) //Xét mọi đỉnh v nối từ u
        if (avail[v]) DFSVisit(v); //v chưa thăm: Thăm v bằng DFS
    if («u là chốt»)
        «Liệt kê và loại bỏ các đỉnh thuộc thành phần liên thông mạnh ứng với chốt u»;
}

for (∀u ∈ V) avail[u] = true; //Đánh dấu các đỉnh đều chưa thăm
for (∀u ∈ V)
    if (avail[u]) //Gặp đỉnh u chưa thăm
        DFSVisit(u); //DFS từ u
```

Trình bày dài dòng như vậy, nhưng bây giờ chúng ta mới thảo luận tới vấn đề quan trọng nhất: Làm thế nào kiểm tra một đỉnh  $r$  nào đó có phải là chốt hay không?

#### **Định lý 24-4**

Trong mô hình cài đặt của thuật toán Tarjan, việc kiểm tra đỉnh  $r$  có phải là chốt không được thực hiện khi đỉnh  $r$  được duyệt xong, khi đó  $r$  là chốt nếu và chỉ nếu không tồn tại cung nối từ nhánh DFS gốc  $r$  tới một đỉnh thăm trước  $r$ .

#### **Chứng minh**

Ta nhắc lại các tính chất của 4 loại cung:

- ✳ Cung DFS và cung xuôi nối từ đỉnh thăm trước đến đỉnh thăm sau, hơn nữa chúng đều là cung nối từ tiền bối tới hậu duệ
- ✳ Cung ngược và cung chéo nối từ đỉnh thăm sau tới đỉnh thăm trước, cung ngược nối từ hậu duệ tới tiền bối còn cung chéo nối hai đỉnh không có quan hệ tiền bối–hậu duệ.

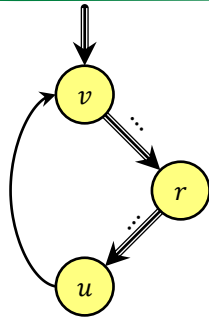
Nếu trong nhánh DFS gốc  $r$  không có cung tới đỉnh thăm trước  $r$  thì tức là không tồn tại cung ngược và cung chéo đi ra khỏi nhánh DFS gốc  $r$ . Điều đó chỉ ra rằng từ  $r$ , đi theo các cung của đồ thị sẽ chỉ đến được những đỉnh nằm trong nội bộ nhánh DFS gốc  $r$  mà thôi. Thành phần liên thông mạnh chứa  $r$  phải nằm trong tập các đỉnh có thể đến từ  $r$ , tập này lại chính là nhánh DFS gốc  $r$ , vậy nên  $r$  là chốt.

Ngược lại, nếu từ đỉnh  $u$  của nhánh DFS gốc  $r$  có cung  $(u, v)$  tới đỉnh  $v$  thăm trước  $r$ , ta chứng minh rằng  $r$  không thể là chốt.

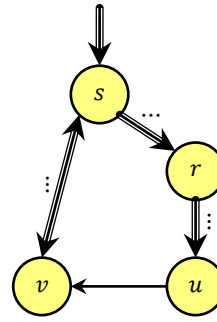
Vì  $r$  là tiền bối của  $u$  nên  $r$  được thăm trước  $u$ . Đỉnh  $v$  thăm trước  $r$  nên  $v$  cũng được thăm trước  $u$ . Vậy thì cung  $(u, v)$  chỉ có thể là cung ngược hoặc cung chéo.

Nếu cung  $(u, v)$  là cung ngược thì  $v$  là tiền bối của  $u$ . Vậy cả  $r$  và  $v$  đều là tiền bối của  $u$  nhưng  $r$  thăm sau  $v$  nên  $r$  là hậu duệ của  $v$ . Ta có một chu trình  $v \rightsquigarrow r \rightsquigarrow u \rightarrow v$  nên cả  $v$  và  $r$  thuộc cùng một thành phần liên thông mạnh. Xét về vị trí trên cây DFS,  $v$  là tiền bối của  $r$  nên  $r$  không thể là chốt.





TH  $(u, v)$  là cung ngược



TH  $(u, v)$  là cung chéo

Nếu cung  $(u, v)$  là cung chéo, ta gọi  $s$  là chốt của thành phần liên thông mạnh chứa  $v$ . Tại thời điểm hàm  $\text{DFSVisit}(u)$  xét tới cung  $(u, v)$ , đỉnh  $r$  đã được thăm nhưng chưa duyệt xong do  $r$  là tiền bối của  $u$ .

Ta xét tới đỉnh  $s$ , đỉnh này được thăm trước  $v$  do  $s$  là chốt của thành phần liên thông mạnh chứa  $v$ , đỉnh  $v$  lại được thăm trước  $r$  theo giả thiết, còn đỉnh  $r$  được thăm trước  $u$  vì  $r$  là tiền bối của  $u$ . Như vậy đỉnh  $s$  đã được thăm trước  $u$ . Mặt khác, khi đỉnh  $u$  được thăm thì đỉnh  $s$  chưa được duyệt xong, vì nếu  $s$  được duyệt xong thì thuật toán đã loại bỏ tất cả các đỉnh thuộc thành phần liên thông mạnh chốt  $s$ , đỉnh  $v$  đã bị loại khỏi đồ thị thì cung  $(u, v)$  sẽ không được tính đến nữa.

Suy ra khi hàm  $\text{DFSVisit}(u)$  được gọi, hai hàm  $\text{DFSVisit}(r)$  và  $\text{DFSVisit}(s)$  đều đã được gọi nhưng chưa thoát, tức là chúng nằm trên một dây chuyền đệ quy, hay  $r$  và  $s$  có quan hệ tiền bối-hậu duệ. Vì  $s$  được thăm trước  $r$  nên  $s$  sẽ là tiền bối của  $r$ , ta có chu trình  $s \rightsquigarrow r \rightsquigarrow u \rightarrow v \rightsquigarrow s$  nên  $r$  và  $s$  thuộc cùng một thành phần liên thông mạnh, thành phần này đã có chốt  $s$  rồi nên  $r$  không thể là chốt nữa.

Từ Định lý 24-4, việc sẽ kiểm tra đỉnh  $r$  có là chốt hay không có thể thay bằng việc kiểm tra xem có tồn tại cung nối từ một đỉnh thuộc nhánh DFS gốc  $r$  tới một đỉnh thăm trước  $r$  hay không?

Dưới đây là một cách cài đặt rất thông minh, nội dung của nó là đánh số thứ tự các đỉnh theo thứ tự thăm đến. Định nghĩa  $\text{num}[u]$  là số thứ tự của đỉnh  $u$  theo cách đánh số đó. Ta tính thêm  $\text{low}[u]$  là giá trị  $\text{num}[\cdot]$  nhỏ nhất trong các đỉnh có thể đến được từ một đỉnh  $v$  nào đó của nhánh DFS gốc  $u$  bằng một cung. Cụ thể cách tính  $\text{low}[u]$  như sau:

Trong hàm  $\text{DFSVisit}(u)$ , trước hết ta đánh số thứ tự thăm cho đỉnh  $u$ :  $\text{num}[u]$  và khởi tạo  $\text{low}[u] = +\infty$ . Sau đó xét các đỉnh  $v$  nối từ  $u$ , có hai khả năng:

✿ Nếu  $v$  đã thăm thì ta cực tiểu hoá  $\text{low}[u]$  theo công thức:

$$\text{low}[u]_{\text{mới}} = \min(\text{low}[u]_{\text{cũ}}, \text{num}[v])$$

- ✱ Nếu  $v$  chưa thăm thì ta gọi đệ quy  $DFSVisit(v)$  để xây dựng nhánh DFS gốc  $v$  (con của  $u$ ), cũng là để tính giá trị  $low[v]$  (bằng  $num[.]$  nhỏ nhất của các đỉnh có thể đến được từ nhánh DFS gốc  $v$ ), sau đó cực tiểu hoá  $low[u]$  theo công thức:

$$low[u]_{\text{mới}} = \min(low[u]_{\text{cũ}}, low[v])$$

Khi duyệt xong một đỉnh  $u$  (chuẩn bị thoát khỏi hàm  $DFSVisit(u)$ ), ta so sánh  $low[u]$  và  $num[u]$ , nếu như  $low[u] \geq num[u]$  thì  $u$  là chốt, bởi không có cung nối từ một đỉnh thuộc nhánh DFS gốc  $u$  tới một đỉnh thăm trước  $u$ . Khi đó chỉ việc liệt kê các đỉnh thuộc thành phần liên thông mạnh chứa  $u$  chính là nhánh DFS gốc  $u$ .

Để công việc dễ dàng hơn nữa, ta sử dụng một ngăn xếp để chứa các đỉnh thuộc một nhánh DFS: Khi thăm đến một đỉnh  $u$ , ta đẩy ngay đỉnh  $u$  đó vào ngăn xếp, thì khi duyệt xong đỉnh  $u$ , mọi đỉnh thuộc nhánh DFS gốc  $u$  sẽ được đẩy vào ngăn xếp ngay sau  $u$ . Nếu  $u$  là chốt, ta chỉ việc lấy các đỉnh ra khỏi ngăn xếp cho tới khi lấy tới đỉnh  $u$  là sẽ được nhánh DFS gốc  $u$  cũng chính là thành phần liên thông mạnh chứa  $u$ .

### ✱ Mô hình

Dưới đây là mô hình cài đặt chi tiết của thuật toán Tarjan

```
void DFSVisit(u ∈ V)
{
    num[u] = «Số thứ tự thăm»; //Đỉnh nào thăm trước đánh số trước
    low[u] = +∞;
    «Đẩy u vào ngăn xếp»;
    for (∀v: (u, v) ∈ E)
    {
        if («v đã bị xóa») continue; // bỏ qua
        if («v đã thăm»)
            low[u] = min(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
        else //v chưa thăm
        {
            DFSVisit(v); //Đi thăm v, cũng là tính luôn low[v]
            low[u] = min(low[u], low[v]); //Cực tiểu hóa low[u] theo low[v]
        }
    }
    //Đến đây u đã duyệt xong
    if (low[u] ≥ num[u]) //u là chốt, liệt kê thành phần liên thông mạnh chốt u
        «Xuất các đỉnh lấy ra từ ngăn xếp và xóa luôn cho tới khi u bị xóa»;
}

for (∀u ∈ V) «Đánh dấu u chưa thăm»;
for (∀u ∈ V)
    if («u chưa thăm»)
        DFSVisit(u); //DFS từ u
```

Bởi thuật toán Tarjan chỉ là sửa đổi của thuật toán DFS, các phép vào/ra ngăn xếp được thực hiện không quá  $n$  lần. Vậy nên thời gian thực hiện giải thuật vẫn là  $\Theta(|V| + |E|)$  trong trường hợp đồ thị được biểu diễn bằng danh sách kề hoặc danh

sách liên thuộc, là  $\Theta(|V|^2)$  nếu dùng ma trận kề và là  $\Theta(|V||E|)$  nếu dùng danh sách cạnh.

### ✳ Cài đặt

Chương trình cài đặt dưới đây sử dụng luôn các nhãn *num*[.] để đánh dấu một đỉnh đã thăm hay chưa cũng như để đánh dấu một đỉnh đã bị loại bỏ hay chưa. Vì các đỉnh được đánh số từ 0 trở đi theo thứ tự thăm nên có thể:

- ✳ Dùng hằng số  $-1$  (available) gán cho *num*[*v*] nếu đỉnh *v* chưa được thăm
- ✳ Dùng hằng số  $-2$  (deleted) gán cho *num*[*v*] nếu đỉnh *v* đã bị xóa.

#### 🌀 TARJAN.CPP ✓ Thuật toán Tarjan 🌀

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
const int maxN = 1e6;
const int available = -1;
const int deleted = -2;
int n, m; //Số đỉnh và số cung
vector<int> adj[maxN]; //Danh sách kề
int num[maxN], low[maxN];
stack<int> Stack;

void ReadInput() //Nhập dữ liệu
{
    cin >> n >> m;
    while (m-- > 0) //Đọc m cung
    {
        int u, v;
        cin >> u >> v; //Đọc cung (u, v)
        adj[u].push_back(v); //Đưa v vào danh sách kề của u
    }
}

inline void Minimize(int& Target, int Value) //Target = min(Target, Value)
{
    if (Value < Target) Target = Value;
}
```



```
void DFSVisit(int u) //Thuật toán DFS
{
    static int Time = 0; //Biến đếm số thứ tự thăm
    num[u] = Time++; //Đánh số u theo thứ tự thăm, cũng là đánh dấu đã thăm (≠ available)
    low[u] = maxN; //+∞
    Stack.push(u); //Đẩy u vào ngăn xếp
    for (int v: adj[u]) //Xét các đỉnh v nối từ u
    {
        if (num[v] == deleted) continue; //v đã bị xóa, bỏ qua
        if (num[v] != available) //v đã thăm
            Minimize(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
        else //v chưa thăm
        {
            DFSVisit(v); //Thăm v, tính low[v]
            Minimize(low[u], low[v]); //Cực tiểu hóa low[u] theo low[v]
        }
    }
    if (low[u] >= num[u]) //u đã duyệt xong, phát hiện u là chốt
    {
        cout << "Strongly connected component:\n"; //Liệt kê thành phần liên thông mạnh
        int v;
        do
        {
            v = Stack.top(); Stack.pop(); //Lấy một đỉnh v khỏi ngăn xếp
            cout << v << ", "; //Output v thuộc thành phần liên thông mạnh chốt u
            num[v] = deleted; //Đánh dấu xóa v
        }
        while (v != u); //Khi u đã bị lấy ra khỏi ngăn xếp thì dừng
        cout << "\n";
    }
}

void Tarjan()
{
    fill(num, num + n, available);
    for (int u = 0; u < n; ++u)
        if (num[u] == -1)
            DFSVisit(u);
}

int main()
{
    ReadInput();
    Tarjan();
}
```

#### 24.4.4. Thuật toán Kosaraju-Sharir

##### \* Ý tưởng của thuật toán

Có một thuật toán khác để liệt kê các thành phần liên thông mạnh là thuật toán Kosaraju-Sharir [16]. Thuật toán này thực hiện qua hai bước:

- ✿ Bước 1: Dùng thuật toán tìm kiếm theo chiều sâu với hàm  $DFSVisit$ , nhưng thêm vào một thao tác nhỏ: đánh số lại các đỉnh theo thứ tự duyệt xong.
- ✿ Bước 2: Đảo chiều các cung của đồ thị, xét lần lượt các đỉnh theo thứ tự từ đỉnh duyệt xong sau cùng tới đỉnh duyệt xong đầu tiên, với mỗi đỉnh đó, ta lại dùng thuật toán tìm kiếm trên đồ thị (BFS hay DFS) liệt kê những đỉnh nào đến được từ đỉnh đang xét, đó chính là một thành phần liên thông mạnh. Liệt kê xong thành phần nào, ta loại ngay các đỉnh của thành phần đó khỏi đồ thị.

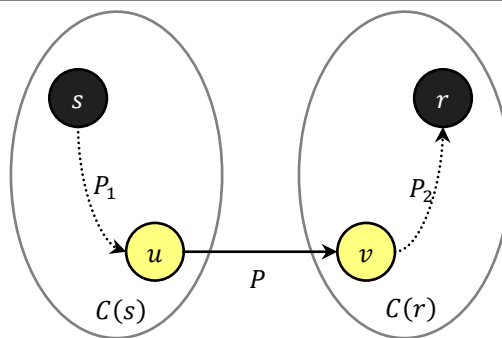
##### Bổ đề 24-5

Với  $r$  là đỉnh duyệt xong sau cùng thì  $r$  là chốt của một thành phần liên thông mạnh không có cung đi vào.

##### Chứng minh

Dễ thấy rằng đỉnh  $r$  duyệt xong sau cùng chắc chắn là gốc của một cây DFS nên  $r$  sẽ là chốt của một thành phần liên thông mạnh, ký hiệu  $C(r)$ .

Gọi  $s$  là chốt của một thành phần liên thông mạnh  $C(s)$  khác. Ta chứng minh rằng không thể tồn tại cung đi từ  $C(s)$  sang  $C(r)$ , giả sử phản chứng rằng có cung  $(u, v)$  trong đó  $u \in C(s)$  và  $v \in C(r)$ . Khi đó tồn tại một đường đi  $P_1: s \rightsquigarrow u$  trong nội bộ  $C(s)$  và tồn tại một đường đi  $P_2: v \rightsquigarrow r$  nội bộ  $C(r)$ . Nối đường đi  $P_1: s \rightsquigarrow u$  với cung  $(u, v)$  và nối tiếp với đường đi  $P_2: v \rightsquigarrow r$  ta được một đường đi  $P: s \rightsquigarrow r$  (Hình 24-8)



Hình 24-8

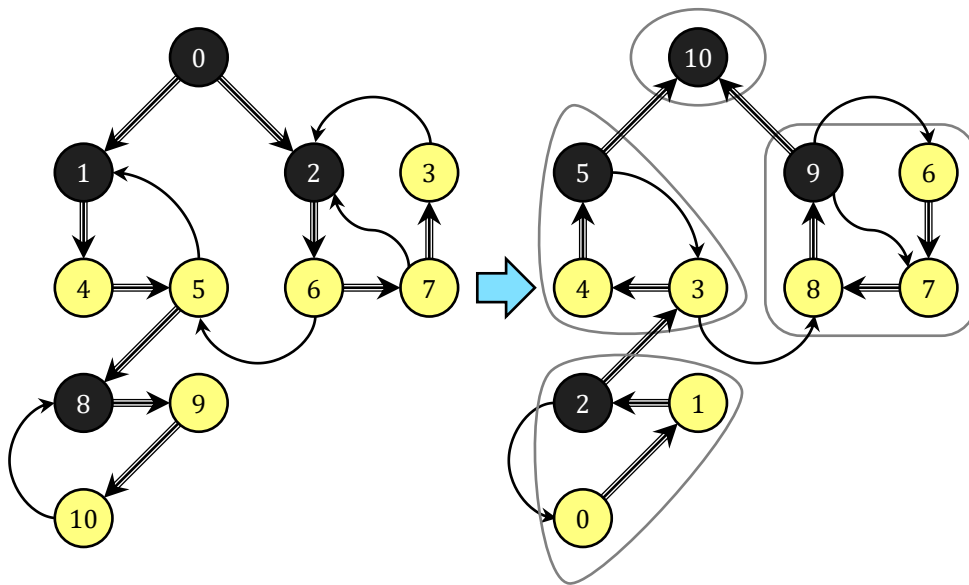
Theo tính chất của chốt là đỉnh thăm trước mọi đỉnh khác thuộc cùng thành phần liên thông mạnh,  $s$  sẽ được thăm trước mọi đỉnh  $\in P_1$  và  $r$  sẽ được thăm trước mọi đỉnh  $\in P_2$ . Có hai khả năng xảy ra:

- ✳ Nếu  $s$  được thăm trước  $r$  thì vào thời điểm  $s$  được thăm, mọi đỉnh khác trên đường đi  $P$  chưa thăm. Theo Định lý 23-3 (định lý đường đi trắng),  $s$  sẽ là tiền bối của  $r$  và phải được duyệt xong sau  $r$ . Trái với giả thiết  $r$  là đỉnh duyệt xong sau cùng.
- ✳ Nếu  $s$  được thăm sau  $r$ , nghĩa là vào thời điểm  $r$  được thăm thì  $s$  chưa thăm, lại do  $r$  được duyệt xong sau cùng nên vào thời điểm  $r$  duyệt xong thì  $s$  đã duyệt xong. Theo Định lý 23-1,  $s$  sẽ là hậu duệ của  $r$ . Vậy từ  $s$  có đường đi tới  $r$  và ngược lại, nghĩa là  $r$  và  $s$  thuộc cùng một thành phần liên thông mạnh. Mâu thuẫn.

Bổ đề được chứng minh.

Bổ đề 24-5 chỉ ra tính đúng đắn của thuật toán Kosaraju-Sharir: Đỉnh  $r$  duyệt xong sau cùng chắc chắn là chốt của một thành phần liên thông mạnh và thành phần liên thông mạnh này gồm mọi đỉnh đến được  $r$ . Việc liệt kê các đỉnh thuộc thành phần liên thông mạnh chốt  $r$  được thực hiện trong thuật toán thông qua thao tác đảo chiều các cung của đồ thị rồi liệt kê các đỉnh đến được từ  $r$ .

Loại bỏ thành phần liên thông mạnh với chốt  $r$  khỏi đồ thị. Cây DFS gốc  $r$  lại phân rã thành nhiều cây con. Lập luận tương tự như trên với đỉnh duyệt xong sau cùng (Hình 24-9).



Hình 24-9. Đánh số lại, đảo chiều các cung và thực hiện BFS/DFS với cách chọn đỉnh xuất phát ngược lại với thứ tự duyệt xong (thứ tự 10, 9, 5, 2 ở hình bên phải)

#### ✳ Cài đặt

Trong việc lập trình thuật toán Kosaraju-Sharir, việc đánh số lại các đỉnh được thực hiện bằng danh sách: Tại bước duyệt đồ thị lần 1, mỗi khi duyệt xong một đỉnh thì đỉnh đó được đưa vào cuối danh sách. Sau khi đảo chiều các cung của đồ

thị, chúng ta chỉ cần duyệt từ cuối danh sách sẽ được các đỉnh đúng thứ tự ngược với thứ tự duyệt xong. Danh sách nói trên có thể biểu diễn bằng một ngăn xếp.

Ngoài ra, đối với thuật toán Kosaraju-Sharir, đồ thị cần biểu diễn bởi cả hai loại danh sách kề: forward/reverse star cho tiện xử lý cả trên đồ thị nguyên bản và đồ thị đảo chiều.

#### KOSARAJUSHARIR.PAS ✓ Thuật toán Kosaraju-Sharir

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
const int maxN = 1e6;
const int available = -1;
const int deleted = -2;
int n, m; //Số đỉnh và số cung
vector<int> adjf[maxN], adjr[maxN]; //Danh sách kề forward/reverse star
stack<int> Stack;
bool avail[maxN];

void ReadInput() //Nhập dữ liệu
{
    cin >> n >> m;
    while (m-- > 0) //Đọc m cung
    {
        int u, v;
        cin >> u >> v; //Đọc 1 cung (u, v)
        adjf[u].push_back(v); //Đưa v vào danh sách kề forward star của u
        adjr[v].push_back(u); //Đưa u vào danh sách kề reverse star của v
    }
}

void Scan(int u) //DFS trên đồ thị nguyên bản, các đỉnh duyệt xong được đẩy vào ngăn xếp
{
    avail[u] = false;
    for (int v: adjf[u]) //Xét danh sách forward star
        if (avail[v]) Scan(v); //Gặp v chưa thăm, DFS tiếp từ v
    Stack.push(u); //Duyệt xong u, đẩy u vào ngăn xếp
}

void Enum(int u) //DFS trên đồ thị đảo chiều, liệt kê các đỉnh chưa thăm đến được từ u
{
    cout << u << ", ";
    avail[u] = false;
    for (int v: adjr[u]) //Xét danh sách reverse star
        if (avail[v]) Enum(v); //Gặp v chưa thăm, DFS tiếp từ v
}
```

```
void KosarajuSharir()
{
    fill(avail, avail + n, true); //Đánh dấu mọi đỉnh đều chưa thăm
    for (int u = 0; u < n; ++u)
        if (avail[u]) Scan(u); //DFS trên đồ thị nguyên bản
    fill(avail, avail + n, true); //Đánh dấu lại mọi đỉnh đều chưa thăm
    while (!Stack.empty()) //Xét ngược thứ tự duyệt xong
    {
        int u = Stack.top(); Stack.pop();
        if (avail[u]) //u là đỉnh duyệt xong sau cùng chưa liệt kê
        {
            cout << "Strongly connected component:\n"; //Thành phần liên thông mạnh chứa u
            Enum(u);
            cout << "\n";
        }
    }
}

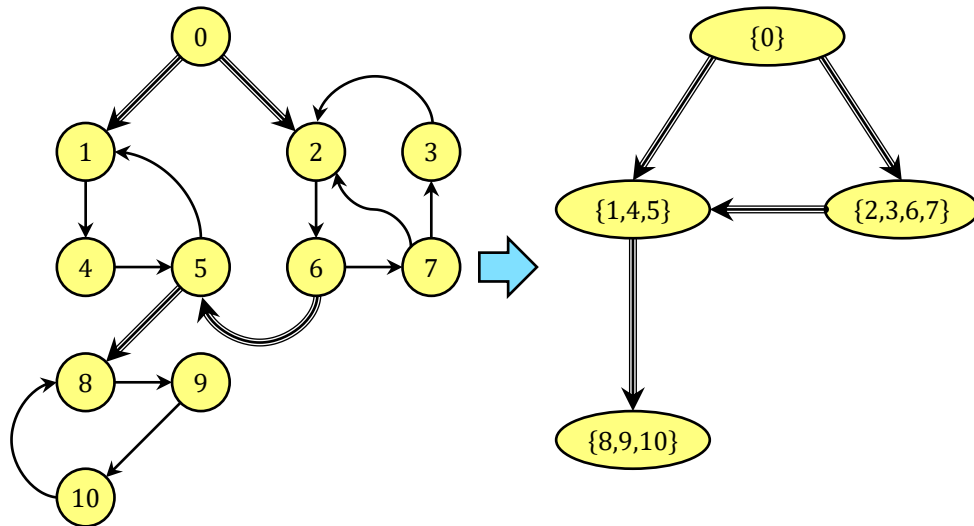
int main()
{
    ReadInput();
    KosarajuSharir();
}
```

Thời gian thực hiện giải thuật có thể tính bằng hai lượt DFS, vậy nên thời gian thực hiện giải thuật sẽ là  $\Theta(|V| + |E|)$  trong trường hợp đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, là  $\Theta(|V|^2)$  nếu dùng ma trận kề và là  $\Theta(|V||E|)$  nếu dùng danh sách cạnh.

## 24.5. Sắp xếp tô pô

Xét đồ thị có hướng  $G = (V, E)$ , ta xây dựng đồ thị có hướng  $G^{SCC} = (V^{SCC}, E^{SCC})$  như sau: Mỗi đỉnh thuộc  $V^{SCC}$  tương ứng với một thành phần liên thông mạnh của  $G$ . Một cung  $(r, s) \in E^{SCC}$  nếu và chỉ nếu tồn tại một cung  $(u, v) \in E$  trên  $G$  trong đó  $u \in r; v \in s$ .

Đồ thị  $G^{SCC}$  gọi là đồ thị các thành phần liên thông mạnh (Hình 24-10)



Hình 24-10. Đồ thị có hướng và đồ thị các thành phần liên thông mạnh

Đồ thị  $G^{SCC}$  là đồ thị có hướng không có chu trình (*directed acyclic graph-DAG*) vì nếu  $G^{SCC}$  có chu trình, ta có thể hợp tất cả các thành phần liên thông mạnh tương ứng với các đỉnh dọc trên chu trình để được một thành phần liên thông mạnh lớn hơn trên đồ thị  $G$ .

Trong thuật toán Tarjan, khi một thành phần liên thông mạnh được liệt kê, thành phần đó sẽ tương ứng với một đỉnh không có cung đi ra trên  $G^{SCC}$ . Còn trong thuật toán Kosaraju-Sharir, khi một thành phần liên thông mạnh được liệt kê, thành phần đó sẽ tương ứng với một đỉnh không có cung đi vào trên  $G^{SCC}$ . Cả hai thuật toán đều loại bỏ thành phần liên thông mạnh mỗi khi liệt kê xong, tức là loại bỏ đỉnh tương ứng trên  $G^{SCC}$ .

Nếu ta đánh số các đỉnh của  $G^{SCC}$  theo thứ tự các thành phần liên thông mạnh được liệt kê thì thuật toán Kosaraju-Sharir sẽ cho ta một cách đánh số gọi là *sắp xếp tô pô* (*topological sorting*) trên  $G^{SCC}$ : Các cung trên  $G^{SCC}$  khi đó sẽ chỉ nối từ đỉnh mang chỉ số nhỏ tới đỉnh mang chỉ số lớn. Nếu đánh số các đỉnh của  $G^{SCC}$  theo thuật toán Tarjan thì ngược lại, các cung trên  $G^{SCC}$  khi đó sẽ chỉ nối từ đỉnh mang chỉ số lớn tới đỉnh mang chỉ số nhỏ.

### Bài tập 24-1

Chứng minh rằng đồ thị có hướng  $G = (V, E)$  là không có chu trình nếu và chỉ nếu quá trình thực hiện thuật toán tìm kiếm theo chiều sâu trên  $G$  không có cung ngược.

### Bài tập 24-2

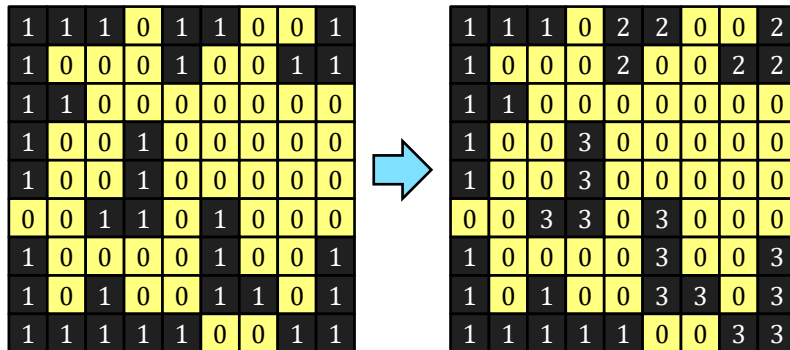
Cho đồ thị có hướng không có chu trình  $G = (V, E)$  và hai đỉnh  $s, t$ . Hãy tìm thuật toán đếm số đường đi từ  $s$  tới  $t$  (chỉ cần đếm số lượng, không cần liệt kê các đường).

### Bài tập 24-3

Trên mặt phẳng với hệ tọa độ Decartes vuông góc cho  $n$  đường tròn, mỗi đường tròn xác định bởi bộ 3 số thực  $(x, y, r)$  ở đây  $(x, y)$  là tọa độ tâm và  $r$  là bán kính. Hai đường tròn gọi là thông nhau nếu chúng có điểm chung. Hãy chia các đường tròn thành một số tối thiểu các nhóm sao cho hai đường tròn bất kỳ trong một nhóm bất kỳ có thể đi được sang nhau sau một số hữu hạn các bước di chuyển giữa hai đường tròn thông nhau.

### Bài tập 24-4

Cho một lưới ô vuông kích thước  $m \times n$  gồm các số nhị phân  $\in \{0,1\}$  ( $m, n \leq 1000$ ). Ta định nghĩa một hình là một miền liên thông các ô kề cạnh mang số 1. Hai hình được gọi là giống nhau nếu hai miền liên thông tương ứng có thể đặt chồng khít lên nhau qua một phép dời hình. Hãy phân loại các hình trong lưới ra thành một số các nhóm thỏa mãn: Mỗi nhóm gồm các hình giống nhau và hai hình bất kỳ thuộc thuộc hai nhóm khác nhau thì không giống nhau:



### Bài tập 24-5

Cho đồ thị có hướng  $G = (V, E)$ , hãy tìm thuật toán và viết chương trình để chọn ra một tập ít nhất các đỉnh  $S \subseteq V$  để mọi đỉnh của  $V$  đều có thể đến được từ ít nhất một đỉnh của  $S$  bằng một đường đi trên  $G$ .

### Bài tập 24-6

Một đồ thị có hướng  $G = (V, E)$  gọi là *nửa liên thông* (semi-connected) nếu với mọi cặp đỉnh  $u, v \in V$  thì hoặc  $u$  có đường đi đến  $v$ , hoặc  $v$  có đường đi đến  $u$ .

a) Chứng minh rằng đồ thị có hướng  $G = (V, E)$  là nửa liên thông nếu và chỉ nếu trên  $G$  tồn tại đường đi qua tất cả các đỉnh (không nhất thiết phải là đường đi đơn)

b) Tìm thuật toán và viết chương trình kiểm tra tính nửa liên thông của đồ thị.