



## PHẦN 4. CÁC THUẬT TOÁN TRÊN ĐỒ THỊ



Leonhard Euler  
1707-1783

Trên thực tế có nhiều bài toán liên quan tới một tập các đối tượng và những mối liên hệ giữa chúng, đòi hỏi toán học phải đặt ra một mô hình biểu diễn một cách chặt chẽ và tổng quát bằng ngôn ngữ ký hiệu, đó là đồ thị. Những ý tưởng cơ bản của nó được đưa ra từ thế kỷ thứ XVIII bởi nhà toán học Thụy Sĩ Leonhard Euler, ông đã dùng mô hình đồ thị để giải bài toán về những cây cầu Konigsberg nổi tiếng.

Mặc dù Lý thuyết đồ thị đã được khoa học phát triển từ rất lâu nhưng lại có nhiều ứng dụng hiện đại. Đặc biệt trong khoảng vài mươi năm trở lại đây, cùng với sự ra đời của máy tính điện tử và sự phát triển nhanh chóng của Tin học, Lý thuyết đồ thị càng được quan tâm đến nhiều hơn. Đặc biệt là các thuật toán trên đồ thị đã có nhiều ứng dụng trong nhiều lĩnh vực khác nhau như: Mạng máy tính, Lý thuyết mã, Tối ưu hoá, Kinh tế học v.v... Hiện nay, môn học này là một trong những kiến thức cơ sở của bộ môn khoa học máy tính.

Trong phạm vi một chuyên đề, không thể nói kỹ và nói hết những vấn đề của lý thuyết đồ thị. Tập bài giảng này sẽ xem xét lý thuyết đồ thị dưới góc độ người lập trình, tức là khảo sát những **thuật toán cơ bản nhất** có thể **dễ dàng cài đặt trên máy tính** một số ứng dụng của nó.. Công việc của người lập trình là đọc hiểu được ý tưởng cơ bản của thuật toán và cài đặt được chương trình trong bài toán tổng quát cũng như trong trường hợp cụ thể.

## §1. CÁC KHÁI NIỆM CƠ BẢN

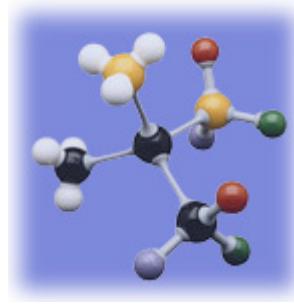
### 1.1. ĐỊNH NGHĨA ĐỒ THỊ (GRAPH)

Là một cấu trúc rời rạc gồm các đỉnh và các cạnh nối các đỉnh đó. Được mô tả hình thức:

$$G = (V, E)$$

V gọi là tập các **đỉnh** (Vertices) và E gọi là tập các **cạnh** (Edges). Có thể coi E là tập các cặp  $(u, v)$  với  $u$  và  $v$  là hai đỉnh của V.

Một số hình ảnh của đồ thị:



**Hình 52: Ví dụ về mô hình đồ thị**

Có thể phân loại đồ thị theo đặc tính và số lượng của tập các cạnh E:

Cho đồ thị  $G = (V, E)$ . Định nghĩa một cách hình thức

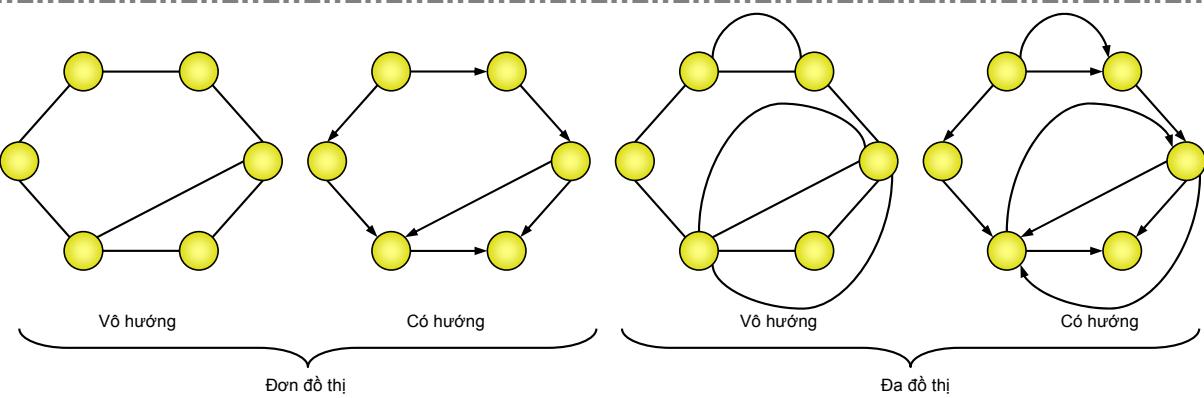
$G$  được gọi là **đơn đồ thị** nếu giữa hai đỉnh  $u, v$  của  $V$  có nhiều nhất là 1 cạnh trong  $E$  nối từ  $u$  tới  $v$ .

$G$  được gọi là **đa đồ thị** nếu giữa hai đỉnh  $u, v$  của  $V$  có thể có nhiều hơn 1 cạnh trong  $E$  nối từ  $u$  tới  $v$  (Hiển nhiên đơn đồ thị cũng là đa đồ thị).

$G$  được gọi là **đồ thị vô hướng** (undirected graph) nếu các cạnh trong  $E$  là không định hướng, tức là cạnh nối hai đỉnh  $u, v$  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.  $(u, v) \equiv (v, u)$

$G$  được gọi là **đồ thị có hướng** (directed graph) nếu các cạnh trong  $E$  là có định hướng, 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 được gọi là **cung**. 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  $(v, u)$ .

Ví dụ:



Hình 53: Phân loại đồ thị

## 1.2. CÁC KHÁI NIỆM

Như trên định nghĩa **đồ thị  $G = (V, E)$  là một cấu trúc rời rạc**, tức là các tập  $V$  và  $E$  hoặc là tập hữu hạn, hoặc là tập đếm được, có nghĩa là ta có thể đánh số thứ tự  $1, 2, 3, \dots$  cho các phần tử của tập  $V$  và  $E$ . Hơn nữa, đứng trên phương diện người lập trình cho máy tính thì ta chỉ quan tâm đến các đồ thị hữu hạn ( $V$  và  $E$  là tập hữu hạn) mà thôi, chính vì vậy từ đây về sau, nếu không chú thích gì thêm thì khi nói tới đồ thị, ta hiểu rằng đó là đồ thị hữu hạn.

### 1.2.1. 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  $v$  trong đồ thị, ta định nghĩa **bậc (degree)** của  $v$ , ký hiệu  $\deg(v)$  là số cạnh liên thuộc với  $v$ . Dễ thấy rằng trên đơn đồ thị thì số cạnh liên thuộc với  $v$  cũng là số đỉnh kề với  $v$ .

**Định lý:** Giả sử  $G = (V, E)$  là đồ thị vô hướng với  $m$  cạnh, khi đó tổng tất cả các bậc đỉnh trong  $V$  sẽ bằng  $2m$ :

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

**Chứng minh:** Khi lấy tổng tất cả các bậc đỉnh tức là mỗi cạnh  $e = (u, v)$  bất kỳ 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: **Bán bậc ra (out-degree)** của  $v$  ký hiệu  $\deg^+(v)$  là số cung đi ra khỏi nó; **bán bậc vào (in-degree)** ký hiệu  $\deg^-(v)$  là số cung đi vào đỉnh đó

**Định lý:** Giả sử  $G = (V, E)$  là đồ thị có hướng với  $m$  cung, khi đó tổng tất cả các bán bậc ra của các đỉnh bằng tổng tất cả các bán bậc vào và bằng  $m$ :

$$\sum_{v \in V} \deg^+(v) = \sum_{v \in V} \deg^-(v) = m$$

**Chứng minh:** Khi lấy tổng tất cả các bán bậc ra hay bán bậc vào, mỗi cung  $(u, v)$  bất kỳ sẽ được tính đúng 1 lần trong  $\deg^+(u)$  và cũng được tính đúng 1 lần trong  $\deg^-(v)$ . Từ đó suy ra kết quả

### 1.2.2. Đường đi và chu trình

Một đường đi với độ dài  $p$  là một dãy  $P=\langle v_0, v_1, \dots, v_p \rangle$  của các đỉnh sao cho  $(v_{i-1}, v_i) \in E$ , ( $\forall i: 1 \leq i \leq p$ ). Ta nói đường đi  $P$  **bao gồm** các đỉnh  $v_0, v_1, \dots, v_p$  và các cạnh  $(v_0, v_1), (v_1, v_2), \dots, (v_{p-1}, v_p)$ . Nếu có một đường đi như trên thì ta nói  $v_p$  **đến được (reachable)** từ  $v_0$  qua  $P$ . Một đường đi gọi là **đơn giản (simple)** nếu tất cả các đỉnh trên đường đi là hoàn toàn phân biệt, một **đường đi con (subpath)**  $P'$  của  $P$  là một đoạn liên tục của các dãy các đỉnh dọc theo  $P$ . Đường đi  $P$  trở thành **chu trình (circuit)** nếu  $v_0=v_p$ . Chu trình  $P$  gọi là **đơn giản (simple)** nếu  $v_1, v_2, \dots, v_p$  là hoàn toàn phân biệt

### 1.2.3. Một số khái niệm khác

Hai đồ thị  $G = (V, E)$  và  $G' = (V', E')$  được gọi là **đẳng cấu (isomorphic)** nếu tồn tại một song ánh  $f: V \rightarrow V'$  sao cho  $(u, v) \in E$  nếu và chỉ nếu  $(f(u), f(v)) \in E'$ .

Đồ thị  $G' = (V', E')$  là **đồ thị con (subgraph)** của đồ thị  $G = (V, E)$  nếu  $V' \subseteq V$  và  $E' \subseteq E$ . Khi đó  $G'$  được gọi là **đồ thị con cảm ứng (induced)** từ  $G$  bởi  $V'$  nếu  $E' = \{(u, v) \in E | u, v \in V'\}$

Cho một đồ thị vô hướng  $G = (V, E)$ , ta gọi **phiên bản có hướng (directed version)** của  $G$  là một đồ thị có hướng  $G' = (V, E')$  sao cho  $(u, v) \in E'$  nếu và chỉ nếu  $(u, v) \in E$ . Nói cách khác  $G'$  được tạo thành từ  $G$  bằng cách thay mỗi cạnh bằng hai cung có hướng ngược chiều nhau.

Cho một đồ thị có hướng  $G = (V, E)$ , ta gọi **phiên bản vô hướng (undirected version)** của  $G$  là một đồ thị vô hướng  $G' = (V, E')$  sao cho  $(u, v) \in E'$  nếu và chỉ nếu  $(u, v) \in E$  hoặc  $(v, u) \in E$ .

Một đồ thị vô hướng gọi là **liên thông (connected)** nếu với mọi cặp đỉnh  $(u, v)$  ta có  $u$  đến được  $v$ . Một đồ thị có hướng gọi là **liên thông mạnh (strongly connected)** nếu với mỗi cặp đỉnh  $(u, v)$ , ta có  $u$  đến được  $v$  và  $v$  đến được  $u$ . Một đồ thị có hướng gọi là **liên thông yếu (weakly connected)** nếu phiên bản vô hướng của nó là đồ thị liên thông.

Một đồ thị vô hướng được gọi là **đầy đủ (complete)** nếu mọi cặp đỉnh đều là kề nhau. Một đồ thị vô hướng gọi là **hai phia (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  $\in X$  cũng như không tồn tại cạnh nối hai đỉnh  $\in Y$ .

Người ta còn mở rộng khái niệm đồ thị thành **siêu đồ thị (hypergraph)**, một siêu đồ thị tương tự như đồ thị vô hướng, những mỗi **siêu cạnh (hyperedge)** không những chỉ có thể nối hai đỉnh mà còn có thể nối một tập các đỉnh với nhau.

## §2. BIỂU DIỄN ĐỒ THỊ TRÊN MÁY TÍNH

### 2.1. MA TRẬN KÈ (ADJACENCY MATRIX)

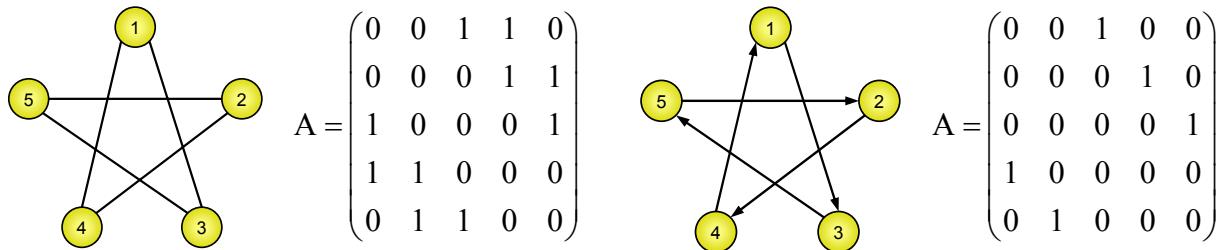
Giả sử  $G = (V, E)$  là một **đơn đồ thị** có số đỉnh (ký hiệu  $|V|$ ) là  $n$ , Không mất tính tổng quát có thể coi các đỉnh được đánh số  $1, 2, \dots, n$ . Khi đó ta có thể biểu diễn đồ thị bằng một ma trận vuông  $A = [a[i, j]]$  cấp  $n$ . Trong đó:

- ❖  $a[i, j] = 1$  nếu  $(i, j) \in E$
- ❖  $a[i, j] = 0$  nếu  $(i, j) \notin E$

Với  $\forall i$ , giá trị của  $a[i, i]$  có thể đặt tùy theo mục đích, thông thường nên đặt bằng 0;

Đối với đa đồ thị thì việc biểu diễn cũng tương tự trên, chỉ có điều nếu như  $(i, j)$  là cạnh thì không phải ta ghi số 1 vào vị trí  $a[i, j]$  mà là ghi số cạnh nối giữa đỉnh  $i$  và đỉnh  $j$ .

**Ví dụ:**



**Các tính chất của ma trận kè:**

Đối với đồ thị vô hướng  $G$ , thì ma trận kè tương ứng là ma trận đối xứng ( $a[i, j] = a[j, i]$ ), điều này không đúng với đồ thị có hướng.

Nếu  $G$  là đồ thị vô hướng và  $A$  là ma trận kè tương ứng thì trên ma trận  $A$ :

$$\text{Tổng các số trên hàng } i = \text{Tổng các số trên cột } i = \text{Bậc của đỉnh } i = \deg(i)$$

Nếu  $G$  là đồ thị có hướng và  $A$  là ma trận kè tương ứng thì trên ma trận  $A$ :

$$\text{Tổng các số trên hàng } i = \text{Bán bậc ra của đỉnh } i = \deg^+(i)$$

$$\text{Tổng các số trên cột } i = \text{Bán bậc vào của đỉnh } i = \deg^-(i)$$

Trong trường hợp  $G$  là đơn đồ thị, ta có thể biểu diễn ma trận kè  $A$  tương ứng là các phần tử logic.  $a[i, j] = \text{TRUE}$  nếu  $(i, j) \in E$  và  $a[i, j] = \text{FALSE}$  nếu  $(i, j) \notin E$

**Ưu điểm của ma trận kè:**

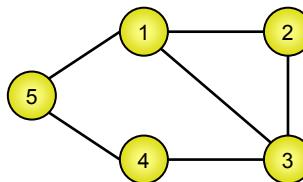
- ❖ Đơn giản, trực quan, dễ cài đặt trên máy tính
- ❖ Để kiểm tra xem hai đỉnh  $(u, v)$  của đồ thị có kè nhau hay không, ta chỉ việc kiểm tra bằng một phép so sánh:  $a[u, v] \neq 0$ .

**Nhược điểm của ma trận kè:**

- ❖ Bất kể số cạnh của đồ thị là nhiều hay ít, ma trận kè luôn luôn đòi hỏi  $n^2$  ô nhớ để lưu các phần tử ma trận, điều đó gây lãng phí bộ nhớ dẫn tới việc không thể biểu diễn được đồ thị với số đỉnh lớn.
- ❖ Với một đỉnh  $u$  bất kỳ của đồ thị, nhiều khi ta phải xét tất cả các đỉnh  $v$  khác kè với nó, hoặc xét tất cả các cạnh liên thuộc với nó. Trên ma trận kè việc đó được thực hiện bằng cách xét tất cả các đỉnh  $v$  và kiểm tra điều kiện  $a[u, v] \neq 0$ . Như vậy, ngay cả khi đỉnh  $u$  là **đỉnh cô lập** (không kè với đỉnh nào) hoặc **đỉnh treo** (chỉ kè với 1 đỉnh) ta cũng buộc phải xét tất cả các đỉnh và kiểm tra điều kiện trên dẫn tới lãng phí thời gian

## 2.2. DANH SÁCH CẠNH (EDGE LIST)

Trong trường hợp đồ thị có  $n$  đỉnh,  $m$  cạnh, ta có thể biểu diễn đồ thị dưới dạng danh sách cạnh bằng cách liệt kê tất cả các cạnh của đồ thị trong một danh sách, mỗi phần tử của danh sách là một cặp  $(u, v)$  tương ứng với một cạnh của đồ thị. (Trong trường hợp đồ thị có hướng thì mỗi cặp  $(u, v)$  tương ứng với một cung,  $u$  là đỉnh đầu và  $v$  là đỉnh cuối của cung). Danh sách được lưu trong bộ nhớ dưới dạng mảng hoặc danh sách móc nối. Ví dụ với đồ thị ở Hình 54:

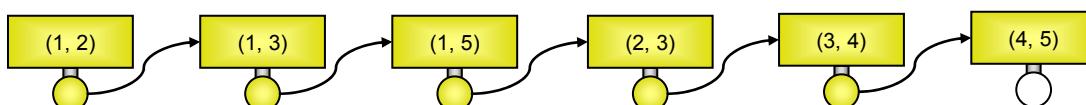


Hình 54

Cài đặt trên mảng:

1	2	3	4	5	6
(1, 2)	(1, 3)	(1, 5)	(2, 3)	(3, 4)	(4, 5)

Cài đặt trên danh sách móc nối:



Ưu điểm của danh sách cạnh:

- ❖ Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ: chẳng hạn  $m < 6n$ ), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ, bởi nó chỉ cần  $2m$  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. (Thuật toán Kruskal chẳng hạn)

Nhược điểm của danh sách cạnh:

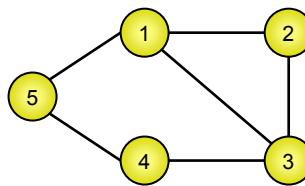
- ❖ Nhược điểm cơ bản của danh sách cạnh là khi ta cần duyệt tất cả các đỉnh kè với đỉnh  $v$  nào đó của đồ thị, thì chẳng có cách nào khác là phải duyệt tất cả các cạnh, lọc ra những

cạnh có chứa đỉnh v và xét đỉnh còn lại. Điều đó khá tốn thời gian trong trường hợp đồ thị dày (nhiều cạnh).

### 2.3. DANH SÁCH KÈ (ADJACENCY LIST)

Để khắc phục nhược điểm của các phương pháp ma trận kè và danh sách cạnh, người ta đề xuất phương pháp biểu diễn đồ thị bằng danh sách kè. Trong cách biểu diễn này, với mỗi đỉnh v của đồ thị, ta cho tương ứng với nó một danh sách các đỉnh kè với v.

Với đồ thị  $G = (V, E)$ . V gồm n đỉnh và E gồm m cạnh. Có hai cách cài đặt danh sách kè phổ biến:



Hình 55

**Cách 1:** Dùng một mảng các đỉnh, mảng đó chia làm n đoạn, đoạn thứ i trong mảng lưu danh sách các đỉnh kè với đỉnh i: Với đồ thị ở Hình 55, danh sách kè sẽ là một mảng Adj gồm 12 phần tử:

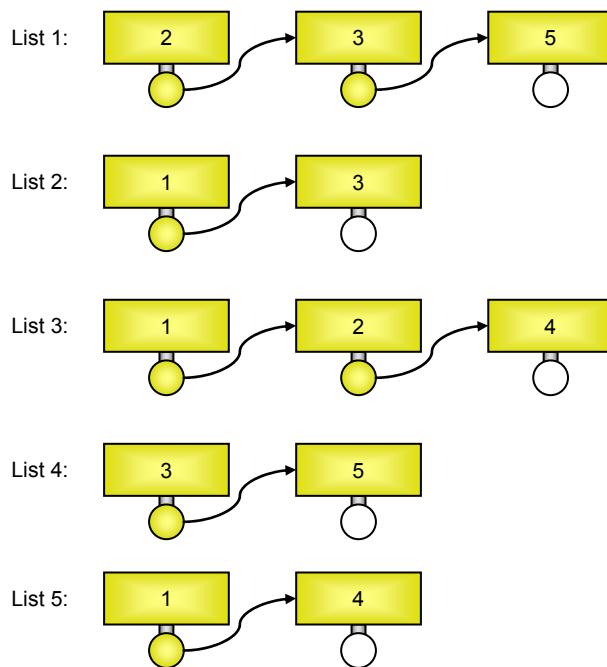
1	2	3	4	5	6	7	8	9	10	11	12
2	3	5	1	3	1	2	4	3	5	1	4

1      2      3      4      5      6      7      8      9      10      11      12  
 { }    { }    { }    { }    { }    { }    { }    { }    { }    { }    { }    { }

Để biết một đoạn nằm từ chỉ số nào đến chỉ số nào, ta có một mảng Head lưu vị trí riêng.  $Head[i]$  sẽ bằng chỉ số đứng liền trước đoạn thứ i. Quy ước  $Head[n + 1]$  bằng m. Với đồ thị ở Hình 55 thì mảng  $Head[1..6]$  sẽ là: (0, 3, 5, 8, 10, 12)

Các phần tử  $Adj[Head[i] + 1..Head[i + 1]]$  sẽ chứa các đỉnh kè với đỉnh i. Lưu ý rằng với đồ thị có hướng gồm m cung thì cấu trúc này cần phải đủ chứa m phần tử, với đồ thị vô hướng m cạnh thì cấu trúc này cần phải đủ chứa  $2m$  phần tử

**Cách 2:** Dùng các danh sách mốc nối: Với mỗi đỉnh i của đồ thị, ta cho tương ứng với nó một danh sách mốc nối các đỉnh kè với i, có nghĩa là tương ứng với một đỉnh i, ta phải lưu lại  $List[i]$  là chốt của một danh sách mốc nối. Ví dụ với đồ thị ở Hình 55, các danh sách mốc nối sẽ là:



Ưu điểm của danh sách kè:

- ❖ Đối với danh sách kè, việc duyệt tất cả các đỉnh kè với một đỉnh v cho trước là hết sức dễ dàng, cái tên “danh sách kè” đã cho thấy rõ điều này. Việc duyệt tất cả các cạnh cũng đơn giản vì một cạnh thực ra là nối một đỉnh với một đỉnh khác kè nó.

Nhược điểm của danh sách kè

- ❖ Danh sách kè yếu hơn ma trận kè 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 kè của  $u$  hay danh sách kè của  $v$ .

Đối với những thuật toán mà ta sẽ khảo sát, danh sách kè tốt hơn hẳn so với hai phương pháp biểu diễn trước. Chỉ có điều, trong trường hợp cụ thể mà ma trận kè hay danh sách cạnh **không thể hiện nhược điểm** thì ta nên dùng ma trận kè (hay danh sách cạnh) bởi cài đặt danh sách kè có phần dài dòng hơn.

## 2.4. NHẬN XÉT

Trên đây là nêu các cách biểu diễn đồ thị trong bộ nhớ của máy tính, còn nhập dữ liệu cho đồ thị thì có nhiều cách khác nhau, dùng cách nào thì tùy. Chẳng hạn nếu biểu diễn bằng ma trận kè mà cho nhập dữ liệu cả ma trận cấp  $n \times n$  ( $n$  là số đỉnh) thì khi nhập từ bàn phím sẽ rất mất thời gian, ta cho nhập kiểu danh sách cạnh cho nhanh. Chẳng hạn mảng  $A$  ( $n \times n$ ) là ma trận kè của một đồ thị vô hướng thì ta có thể khởi tạo ban đầu mảng  $A$  gồm toàn số 0, sau đó cho người sử dụng nhập các cạnh bằng cách nhập các cặp  $(i, j)$ ; chương trình sẽ tăng  $A[i, j]$  và  $A[j, i]$  lên 1. Việc nhập có thể cho kết thúc khi người sử dụng nhập giá trị  $i = 0$ . Ví dụ:

```
program GraphInput;
var
  A: array[1..100, 1..100] of Integer; {Ma trận kè của đồ thị}
  n, i, j: Integer;
begin
  Write('Number of vertices'); ReadLn(n);
```

```
FillChar(A, SizeOf(A), 0);
repeat
    Write('Enter edge (i, j) (i = 0 to exit) ');
    ReadLn(i, j); {Nhập một cặp (i, j) tưởng như là nhập danh sách cạnh}
    if i <> 0 then
        begin {nhưng lưu trữ trong bộ nhớ lại theo kiểu ma trận kè}
        Inc(A[i, j]);
        Inc(A[j, i]);
        end;
    until i = 0; {Nếu người sử dụng nhập giá trị i = 0 thì dừng quá trình nhập, nếu không thì tiếp tục}
end.
```

Trong nhiều trường hợp đều không gian lưu trữ, việc chuyển đổi từ cách biểu diễn nào đó sang cách biểu diễn khác không có gì khó khăn. Nhưng đối với thuật toán này thì làm trên ma trận kè ngắn gọn hơn, đối với thuật toán kia có thể làm trên danh sách cạnh dễ dàng hơn v.v... Do đó, với mục đích dễ hiểu, các chương trình sau này sẽ lựa chọn phương pháp biểu diễn sao cho việc cài đặt đơn giản nhất nhằm nêu bật được bản chất thuật toán. Còn trong trường hợp cụ thể bắt buộc phải dùng một cách biểu diễn nào đó khác, thì việc sửa đổi chương trình cũng không tốn quá nhiều thời gian.

### §3. CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

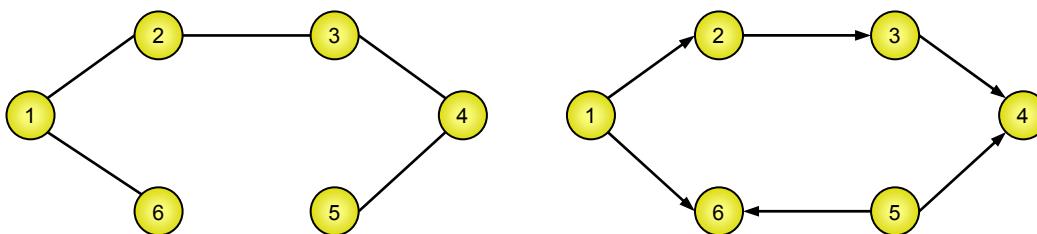
#### 3.1. BÀI TOÁN

Cho đồ thị  $G = (V, E)$ .  $u$  và  $v$  là hai đỉnh của  $G$ . Một **đường đi** (path) độ dài  $p$  từ đỉnh  $s$  đến đỉnh  $f$  là dãy  $x[0..p]$  thoả mãn  $x[0] = s$ ,  $x[p] = f$  và  $(x[i], x[i+1]) \in E$  với  $\forall i: 0 \leq i < p$ .

Đường đi nói trên còn có thể biểu diễn bởi dãy các cạnh:  $(s = x[0], x[1]), (x[1], x[2]), \dots, (x[p-1], x[p] = f)$

Đỉnh  $u$  được gọi là đỉnh đầu, đỉnh  $v$  được gọi là đỉnh cuối của đường đi. Đường đi có đỉnh đầu trùng với đỉnh cuối gọi là **chu trình** (Circuit), đường đi không có cạnh nào đi qua hơn 1 lần gọi là **đường đi đơn**, tương tự ta có khái niệm **chu trình đơn**.

Ví dụ: Xét một đồ thị vô hướng và một đồ thị có hướng trong Hình 56:



Hình 56: Đồ thị và đường đi

Trên cả hai đồ thị,  $(1, 2, 3, 4)$  là đường đi đơn độ dài 3 từ đỉnh 1 tới đỉnh 4.  $(1, 6, 5, 4)$  không phải đường đi vì không có cạnh (cung) nối từ đỉnh 6 tới đỉnh 5.

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** và ở đây 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** cùng với một số ứng dụng của chúng.

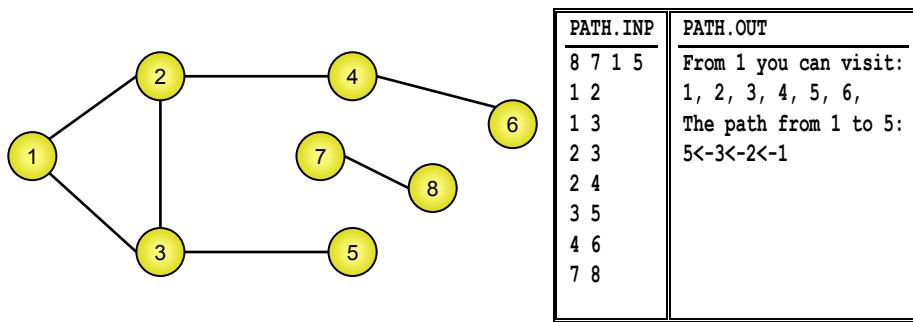
Những cài đặt dưới đây là cho đơn đồ thị vô hướng, muốn làm với đồ thị có hướng hay đa đồ thị cũng không phải sửa đổi gì nhiều.

**Input:** file văn bản PATH.INP. Trong đó:

- ❖ Dòng 1 chứa số đỉnh  $n$  ( $\leq 100$ ), số cạnh  $m$  của đồ thị, đỉnh xuất phát  $s$ , đỉnh kết thúc  $f$  cách nhau một dấu cách.
- ❖  $m$  dòng tiếp theo, mỗi dòng có dạng hai số nguyên dương  $u, v$  cách nhau một dấu cách, thể hiện có cạnh nối đỉnh  $u$  và đỉnh  $v$  trong đồ thị.

**Output:** file văn bản PATH.OUT:

- ❖ Danh sách các đỉnh có thể đến được từ  $s$
- ❖ Đường đi từ  $s$  tới  $f$



### 3.2. THUẬT TOÁN TÌM KIẾM THEO CHIỀU SÂU (DEPTH FIRST SEARCH)

Tư tưởng của thuật toán có thể trình bày như sau: Trước hết, mọi đỉnh x kề với S tất nhiên sẽ đến được từ S. Với mỗi đỉnh x kề với S đó thì tất nhiên những đỉnh y kề với x cũng đến được từ S... Điều đó gợi ý cho ta viết một thủ tục đệ quy DFS(u) mô tả việc duyệt từ đỉnh u bằng cách thăm đỉnh u và tiếp tục quá trình duyệt DFS(v) với v là một đỉnh chưa thăm kề với u.

Để không một đỉnh nào bị liệt kê tới hai lần, ta sử dụng kỹ thuật đánh dấu, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại 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 thủ tục DFS(u), trước khi gọi đệ quy DFS(v) với v là một đỉnh kề với u mà 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 quá trình tìm kiếm theo chiều sâu kết thúc, đường đi từ s tới f sẽ là:

$$f \leftarrow p[1] = \text{Trace}[f] \leftarrow p[2] = \text{Trace}[p[1]] \leftarrow \dots \leftarrow s.$$

```

procedure DFS(u∈V);
begin
  Free[u] := False; {Free[u] = False ⇔ u đã thăm}
  for (∀v ∈ V: Free[v] and ((u, v)∈E)) do {Duyệt mọi đỉnh v chưa thăm kề với u}
    begin
      Trace[v] := u; {Lưu vết đường đi, đỉnh liền trước v trên đường đi từ s tới v là u}
      DFS(v); {Gọi đệ quy duyệt tương tự đối với v}
    end;
  end;
begin {Chương trình chính}
  {Nhập dữ liệu: đồ thị, đỉnh xuất phát S, đỉnh đích F};
  for (∀v ∈ V) do Free[v] := True; {Đánh dấu mọi đỉnh đều chưa thăm}
  DFS(S);
  {Thông báo từ s có thể thăm được những đỉnh v mà Free[v] = False};
  if Free[f] then {s đi tới được f}
    {Truy theo vết từ f để tìm đường đi từ s tới f};
end.

```

Trong cài đặt cụ thể, ta không cần mảng đánh dấu Free[1..n] mà dùng luôn mảng Trace[1..n] để đánh dấu, khởi tạo các phần tử Trace[s] := -1 và Trace[v] := 0 với  $\forall v \neq s$ . Điều kiện để một đỉnh v chưa thăm là Trace[v] = 0, mỗi khi từ đỉnh u thăm đỉnh v, phép gán Trace[v] := u sẽ kiêm luôn công việc đánh dấu v đã thăm.

```

P_4_03_1.PAS * Thuật toán tìm kiếm theo chiều sâu
{$MODE DELPHI} (*This program uses 32-bit Integer [-231..231 - 1]*)
program Depth_First_Search;
const
  InputFile = 'PATH.INP';

```

```

OutputFile = 'PATH.OUT';
max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  Trace: array[1..max] of Integer;
  n, s, f: Integer;

procedure Enter; {Nhập dữ liệu}
var
  i, u, v, m: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  FillChar(a, SizeOf(a), False);
  ReadLn(fi, n, m, s, f);
  for i := 1 to m do
    begin
      ReadLn(fi, u, v);
      a[u, v] := True;
      a[v, u] := True; {Đồ thị vô hướng nên cạnh (u, v) cũng là cạnh (v, u)}
    end;
  Close(fi);
end;

procedure DFS(u: Integer); {Thuật toán tìm kiếm theo chiều sâu bắt đầu từ u}
var
  v: Integer;
begin
  for v := 1 to n do
    if a[u, v] and (Trace[v] = 0) then {Duyệt ∀v chưa thăm kể với u}
      begin
        Trace[v] := u; {Lưu vết đường đi cũng là đánh dấu v đã thăm}
        DFS(v); {Tìm kiếm theo chiều sâu bắt đầu từ v}
      end;
end;

procedure Result; {In kết quả}
var
  fo: Text;
  v: Integer;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  Writeln(fo, 'From ', s, ' you can visit: ');
  for v := 1 to n do
    if Trace[v] <> 0 then Write(fo, v, ', '); {In ra những đỉnh đến được từ s}
  Writeln(fo);
  Writeln(fo, 'The path from ', s, ' to ', f, ': ');
  if Trace[f] = 0 then {Nếu Trace[f] = 0 thì s không thể tới được f}
    WriteLn(fo, 'not found')
  else {s tới được f}
    begin
      while f <> s do {Truy vết đường đi}
        begin
          Write(fo, f, '-');
          f := Trace[f];
        end;
      WriteLn(fo, s);
    end;
  Close(fo);
end;

begin
  Enter;

```

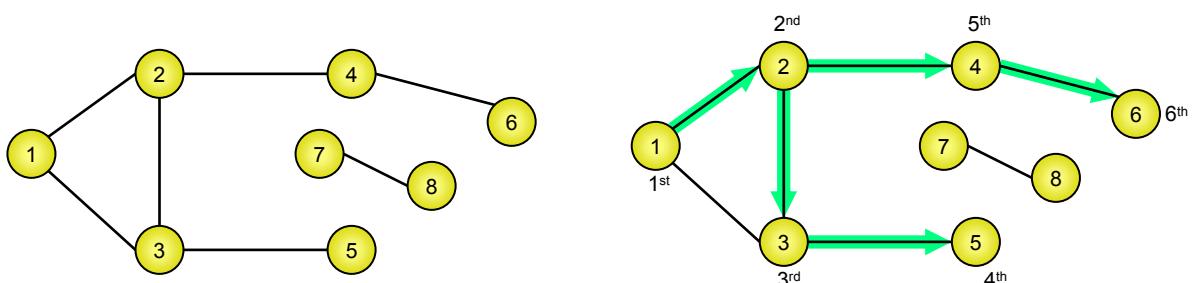
```

FillChar(Trace, SizeOf(Trace), 0); {Mọi đỉnh đều chưa thăm}
Trace[s] := -1; {Ngoại trừ s đã thăm}
DFS(s);
Result;
end.

```

Nhận xét:

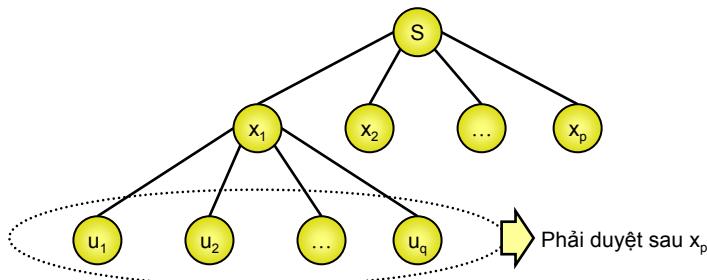
- ❖ Vì có kỹ thuật đánh dấu, nên thủ tục DFS sẽ được gọi  $\leq n$  lần ( $n$  là số đỉnh)
- ❖ Có thể có nhiều đường đi từ  $s$  tới  $f$ , nhưng thuật toán DFS luôn trả về một đường đi có thứ tự từ điển nhỏ nhất.
- ❖ Quá trình tìm kiếm theo chiều sâu cho ta một cây DFS gốc  $s$ . Quan hệ cha-con trên cây được định nghĩa là: nếu từ đỉnh  $u$  tới thăm đỉnh  $v$  ( $DFS(u)$  gọi  $DFS(v)$ ) thì  $u$  là nút cha của nút  $v$ . Hình 57 là đồ thị và cây DFS tương ứng với đỉnh xuất phát  $s = 1$ .



Hình 57: Đồ thị và cây DFS

### 3.3. THUẬT TOÁN TÌM KIẾM THEO CHIỀU RỘNG (BREATH FIRST SEARCH)

Cơ sở của phương pháp cài đặt này là “lập lịch” duyệt các đỉnh. Việc thăm một đỉnh sẽ lên lịch duyệt các đỉnh kề nó sao cho thứ tự duyệt là ưu tiên chiều rộng (đỉnh nào gần  $S$  hơn sẽ được duyệt trước). Ví dụ: Bắt đầu ta thăm đỉnh  $S$ . Việc thăm đỉnh  $S$  sẽ phát sinh thứ tự duyệt những đỉnh ( $x[1], x[2], \dots, x[p]$ ) kề với  $S$  (những đỉnh gần  $S$  nhất). Khi thăm đỉnh  $x[1]$  sẽ lại phát sinh yêu cầu duyệt những đỉnh ( $u[1], u[2], \dots, u[q]$ ) kề với  $x[1]$ . Nhưng rõ ràng các đỉnh  $u$  này “xa”  $S$  hơn những đỉnh  $x$  nên chúng chỉ được duyệt khi tất cả những đỉnh  $x$  đã duyệt xong. Tức là thứ tự duyệt đỉnh sau khi đã thăm  $x[1]$  sẽ là:  $(x[2], x[3], \dots, x[p], u[1], u[2], \dots, u[q])$ .



Hình 58: Thứ tự thăm đỉnh của BFS

Giả sử ta có một danh sách chứa những đỉnh đang “chờ” thăm. Tại mỗi bước, ta thăm một đỉnh đầu danh sách và cho những đỉnh chưa “xếp hàng” kề với nó xếp hàng thêm vào cuối

danh sách. Chính vì nguyên tắc đó nên danh sách chứa những đỉnh đang chờ sẽ được tổ chức dưới dạng hàng đợi (Queue)

Nếu ta có Queue là một hàng đợi với thủ tục Push(v) để đẩy một đỉnh v vào hàng đợi và hàm Pop trả về một đỉnh lấy ra từ hàng đợi thì mô hình của giải thuật có thể viết như sau:

```

for (∀v ∈ V) do Free[v] := True;
Free[s] := False; {Khởi tạo ban đầu chỉ có đỉnh S bị đánh dấu}
Queue := ∅; Push(s); {Khởi tạo hàng đợi ban đầu chỉ gồm một đỉnh s}
repeat {Lặp tái khi hàng đợi rỗng}
  u := Pop; {Lấy từ hàng đợi ra một đỉnh u}
  for (∀v ∈ V: Free[v] and ((u, v) ∈ E)) do {Xét những đỉnh v kề u chưa bị đưa vào hàng đợi}
    begin
      Trace[v] := u; {Lưu vết đường đi}
      Free[v] := False; {Đánh dấu v}
      Push(v); {Đẩy v vào hàng đợi}
    end;
until Queue = ∅;
{Thông báo từ s có thể thăm được những đỉnh v mà Free[v] = False};
if Free[f] then {s đi tới được f}
  {Truy theo vết từ f để tìm đường đi từ s tới f};

```

Tương tự như thuật toán tìm kiếm theo chiều sâu, ta có thể dùng mảng Trace[1..n] kiêm luôn chức năng đánh dấu.

```

P_4_03_2.PAS * Thuật toán tìm kiếm theo chiều rộng
{$MODE DELPHI} (*This program uses 32-bit Integer [-231..231 - 1]*)
program Breadth_First_Search;
const
  InputFile = 'PATH.INP';
  OutputFile = 'PATH.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  Trace: array[1..max] of Integer;
  n, s, f: Integer;

procedure Enter; {Nhập dữ liệu}
var
  i, u, v, m: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  FillChar(a, SizeOf(a), False);
  ReadLn(fi, n, m, s, f);
  for i := 1 to m do
    begin
      ReadLn(fi, u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  Close(fi);
end;

procedure BFS; {Thuật toán tìm kiếm theo chiều rộng}
var
  Queue: array[1..max] of Integer;
  Front, Rear, u, v: Integer;
begin
  Front := 1; Rear := 1; {front: chỉ số đầu hàng đợi, rear: chỉ số cuối hàng đợi}
  Queue[1] := s; {Khởi tạo hàng đợi ban đầu chỉ gồm một đỉnh s}

```

```

FillChar(Trace, SizeOf(Trace), 0); {Các đỉnh đều chưa đánh dấu}
Trace[s] := -1; {Ngoại trừ đỉnh s}
repeat {Lặp tới khi hàng đợi rỗng}
  u := Queue[Front]; Inc(Front); {Lấy từ hàng đợi ra một đỉnh u}
  for v := 1 to n do
    if a[u, v] and (Trace[v] = 0) then {Xét những đỉnh v chưa thăm kề với u}
      begin
        Inc(Rear); Queue[Rear] := v; {Đẩy v vào hàng đợi}
        Trace[v] := u; {Lưu vết đường đi, cũng là đánh dấu luôn}
      end;
  until Front > Rear;
end;

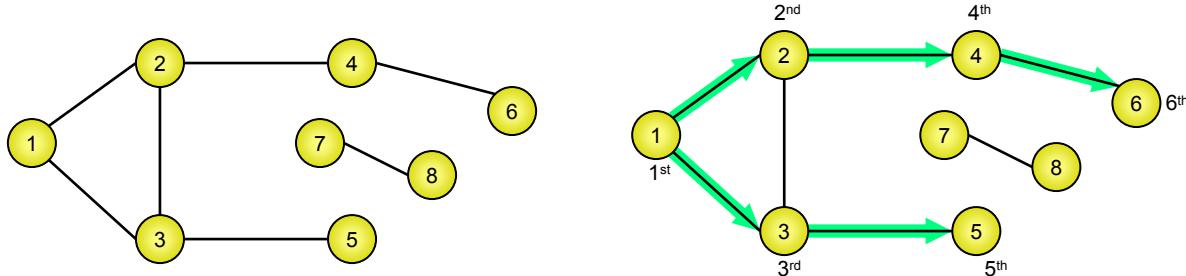
procedure Result; {In kết quả}
var
  fo: Text;
  v: Integer;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  Writeln(fo, 'From ', s, ' you can visit: ');
  for v := 1 to n do
    if Trace[v] <> 0 then Write(fo, v, ', ');
  Writeln(fo);
  Writeln(fo, 'The path from ', s, ' to ', f, ': ');
  if Trace[f] = 0 then
    Writeln(fo, 'not found')
  else
    begin
      while f <> s do
        begin
          Write(fo, f, '<-');
          f := Trace[f];
        end;
      Writeln(fo, s);
    end;
  Close(fo);
end;

begin
  Enter;
  BFS;
  Result;
end.

```

Nhận xét:

- ❖ Có thể có nhiều đường đi từ s tới f nhưng thuật toán BFS luôn trả về một đường đi ngắn nhất (theo nghĩa đi qua ít cạnh nhất).
- ❖ Quá trình tìm kiếm theo chiều rộng cho ta một cây BFS gốc s. Quan hệ cha - con trên cây được định nghĩa là: nếu từ đỉnh u tới thăm đỉnh v thì u là nút cha của nút v. Hình 59 là ví dụ về cây BFS.



Hình 59: Đồ thị và cây BFS

### 3.4. ĐỘ PHÚC TẠP TÍNH TOÁN CỦA BFS VÀ DFS

Quá trình tìm kiếm trên đồ thị bắt đầu từ một đỉnh có thể thăm tất cả các đỉnh còn lại, khi đó cách biểu diễn đồ thị có ảnh hưởng lớn tới chi phí về thời gian thực hiện giải thuật:

- ❖ Trong trường hợp ta biểu diễn đồ thị bằng danh sách kè, cả hai thuật toán BFS và DFS đều có độ phức tạp tính toán là  $O(n + m) = O(\max(n, m))$ . Đây là cách cài đặt tốt nhất.
- ❖ Nếu ta biểu diễn đồ thị bằng ma trận kè như ở trên thì độ phức tạp tính toán trong trường hợp này là  $O(n + n^2) = O(n^2)$ .
- ❖ Nếu ta biểu diễn đồ thị bằng danh sách cạnh, thao tác duyệt những đỉnh kè với đỉnh u sẽ dẫn tới việc phải duyệt qua toàn bộ danh sách cạnh, đây là cài đặt tồi nhất, nó có độ phức tạp tính toán là  $O(n.m)$ .

#### Bài tập

##### Bài 1

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.

##### Bài 2

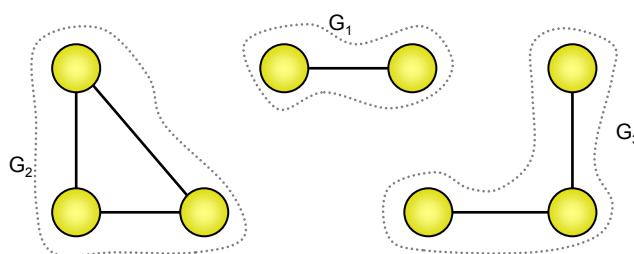
Có nhiều cách dựng cây gốc s chứa tất cả các đỉnh đến được từ s thỏa mãn: nếu u là nút cha của nút v trên cây thì  $(u, v)$  phải là cạnh của G. Cây BFS và cây DFS chỉ là hai trường hợp đặc biệt. Ta đã biết cây BFS là cây thấp nhất, vậy phải chăng cây DFS là cây cao nhất trong số các cây này?

## §4. TÍNH LIÊN THÔNG CỦA ĐỒ THỊ

### 4.1. ĐỊNH NGHĨA

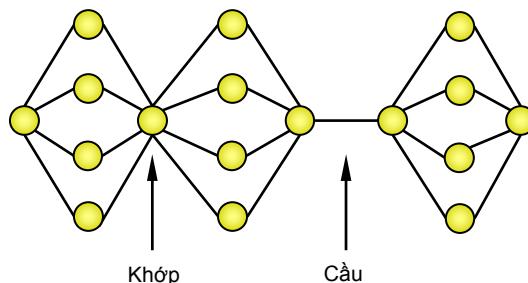
#### 4.1.1. Đôi với đồ thị vô hướng $G = (V, E)$

$G$  gọi là **liên thông** (connected) nếu luôn tồn tại đường đi giữa mọi cặp đỉnh phân biệt của đồ thị. Nếu  $G$  không liên thông thì chắc chắn nó sẽ là hợp của hai hay nhiều đồ thị con\* liên thông, các đồ thị con này đôi một không có đỉnh chung. Các đồ thị con liên thông rời nhau như vậy được gọi là các thành phần liên thông của đồ thị đang xét (Xem ví dụ).



Hình 60: Đồ thị  $G$  và các thành phần liên thông  $G_1, G_2, G_3$  của nó

Đôi khi, việc xoá 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** hay **điểm khớp**. 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** hay **cầu**.



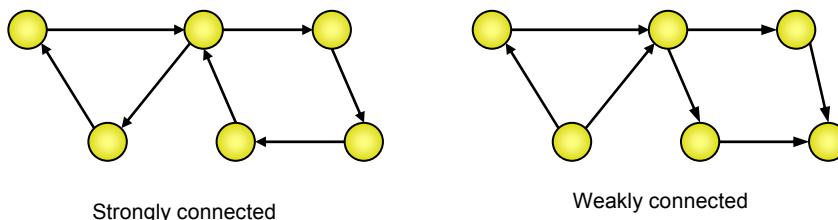
Hình 61: Khớp và cầu

#### 4.1.2. Đôi với 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.

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

**Hình 62: Liên thông mạnh và liên thông yếu**

## 4.2. TÍNH LIÊN THÔNG TRONG ĐỒ THỊ VÔ HƯỚNG

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.

Giả sử đồ thị vô hướng  $G = (V, E)$  có  $n$  đỉnh đánh số  $1, 2, \dots, n$ .

Để liệt kê các thành phần liên thông của  $G$  phương pháp cơ bản nhất là:

Dánh dấu đỉnh 1 và những đỉnh có thể đến từ 1, thông báo những đỉnh đó thuộc thành phần liên thông thứ nhất.

Nếu tất cả các đỉnh đều đã bị đánh dấu thì  $G$  là đồ thị liên thông, nếu không thì sẽ tồn tại một đỉnh  $v$  nào đó chưa bị đánh dấu, ta sẽ đánh dấu  $v$  và các đỉnh có thể đến được từ  $v$ , thông báo những đỉnh đó thuộc thành phần liên thông thứ hai.

Và cứ tiếp tục như vậy cho tới khi tất cả các đỉnh đều đã bị đánh dấu

```
procedure Scan(u)
begin
  (Dùng BFS hoặc DFS liệt kê và đánh dấu những đỉnh có thể đến được từ u);
end;
```

```
begin
  for  $\forall v \in V$  do (khởi tạo  $v$  chưa đánh dấu);
  Count := 0;
  for  $u := 1$  to  $n$  do
    if < $u$  chưa đánh dấu> then
      begin
        Count := Count + 1;
        WriteLn('Thành phần liên thông thứ ', Count, ' gồm các đỉnh : ');
        Scan(u);
      end;
  end.
```

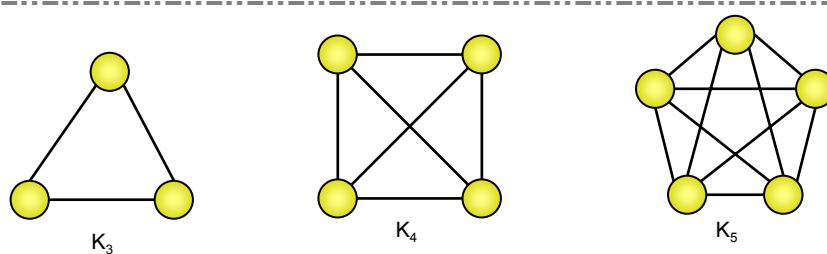
Với thuật toán liệt kê các thành phần liên thông như thế này, thì độ phức tạp tính toán của nó đúng bằng độ phức tạp tính toán của thuật toán tìm kiếm trên đồ thị trong thủ tục Scan.

## 4.3. ĐỒ THỊ ĐẦY ĐỦ VÀ THUẬT TOÁN WARSHALL

### 4.3.1. 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 cạnh nối.

Đồ thị đầy đủ  $K_n$  có đúng:  $\binom{n}{2} = \frac{n(n-1)}{2}$  cạnh và bậc của mọi đỉnh đều bằng  $n - 1$ .



Hình 63: Đồ thị đầy đủ

#### 4.3.2. Bao đóng đồ thị:

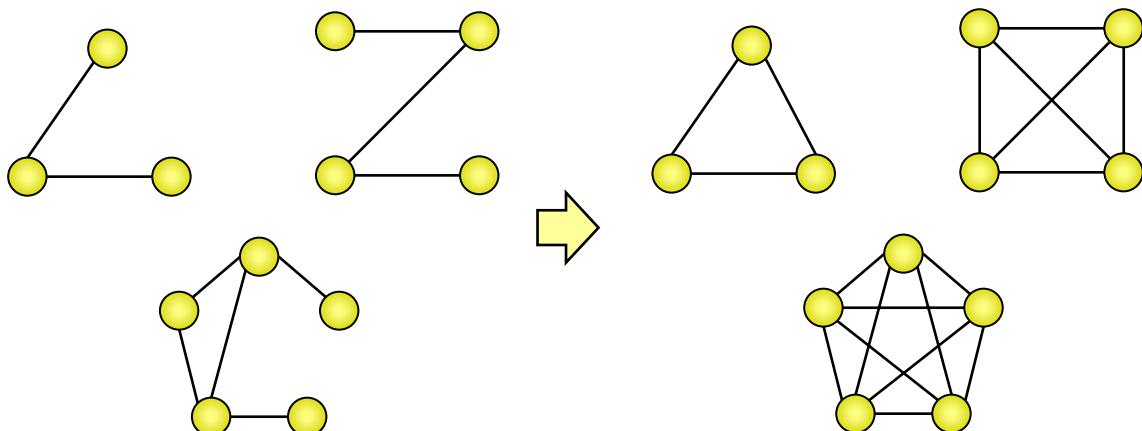
Với đồ thị  $G = (V, E)$ , người ta xây dựng đồ thị  $G' = (V, E')$  cũng gồm những đỉnh của  $G$  còn các cạnh xây dựng như sau: (ở đây quy ước giữa  $u$  và  $v$  luôn có đường đi)

Giữa đỉnh  $u$  và  $v$  của  $G'$  có cạnh nối  $\Leftrightarrow$  Giữa đỉnh  $u$  và  $v$  của  $G$  có đường đi

Đồ thị  $G'$  xây dựng như vậy được gọi là bao đóng của đồ thị  $G$ .

Từ định nghĩa của đồ thị đầy đủ, ta dễ dàng suy ra một đồ thị đầy đủ bao giờ cũng liên thông và từ định nghĩa đồ thị liên thông, ta cũng dễ dàng suy ra được:

- ❖ 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 đầy đủ.



Hình 64: Đơn 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 và một trong những thuật toán đó là:

#### 4.3.3. Thuật toán Warshall

Thuật toán Warshall - 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 Roy cũng đã mô tả thuật toán này vào năm 1959. Thuật toán đó có thể mô tả rất gọn:

Với đơn đồ thị vô hướng  $G$ , với mọi đỉnh  $k$  xét theo thứ tự từ 1 tới  $n$ , ta xét tất cả các cặp đỉnh  $(u, v)$ : nếu có cạnh nối  $(u, k)$  và cạnh nối  $(k, v)$  thì ta tự nối thêm cạnh  $(u, v)$  nếu nó chưa có.

Tư tưởng này dựa trên một quan sát đơn giản như sau: Nếu từ  $u$  có đường đi tới  $k$  và từ  $k$  lại có đường đi tới  $v$  thì tất nhiên từ  $u$  sẽ có đường đi tới  $v$ .

Với  $n$  là số đỉnh của đồ thị và  $A = \{a[i, j]\}$  là ma trận kề biểu diễn đồ thị, ta có thể viết thuật toán Warshall như sau:

```
for k := 1 to n do
    for u := 1 to n do
        for v := 1 to n do
            a[u, v] := a[u, v] or a[u, k] and a[k, v];
```

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. Có nhận xét rằng tuy thuật toán Warshall rất dễ cài đặt nhưng độ phức tạp tính toán của thuật toán này khá lớn ( $O(n^3)$ ).

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

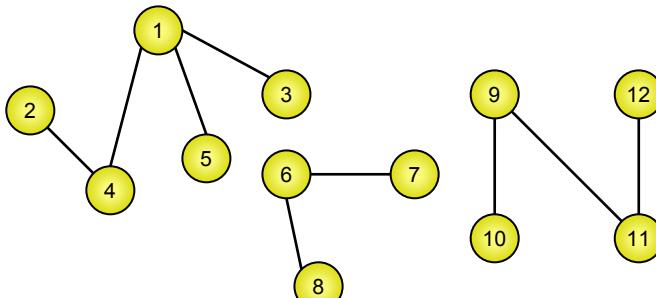
#### Việc cài đặt thuật toán sẽ qua những bước sau:

- ❖ Nhập ma trận kề  $A$  của đồ thị (Lưu ý ở đây  $A[v, v]$  luôn được coi là True với  $\forall v$ )
- ❖ Dùng thuật toán Warshall tìm bao đóng, khi đó  $A$  là ma trận kề của bao đóng đồ thị
- ❖ Dựa vào ma trận kề  $A$ , đỉnh 1 và những đỉnh kề với nó sẽ thuộc thành phần liên thông thứ nhất; với đỉnh  $u$  nào đó không kề với đỉnh 1, 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 1 và đỉnh  $u$ , thì  $v$  cùng với những đỉnh kề nó sẽ thuộc thành phần liên thông thứ ba v.v...

**Input:** file văn bản CONNECT.INP

- ❖ Dòng 1: Chứa số đỉnh  $n$  ( $\leq 100$ ) và số cạnh  $m$  của đồ thị cách nhau ít nhất một dấu cách
- ❖  $m$  dòng tiếp theo, mỗi dòng chứa một cặp số  $u$  và  $v$  cách nhau ít nhất một dấu cách tượng trưng cho một cạnh ( $u, v$ )

**Output:** file văn bản CONNECT.OUT, liệt kê các thành phần liên thông



CONNECT.INP	CONNECT.OUT
12 9	Connected Component 1:
1 3	1, 2, 3, 4, 5,
1 4	Connected Component 2:
1 5	6, 7, 8,
2 4	Connected Component 3:
6 7	9, 10, 11, 12,
6 8	
9 10	
9 11	
11 12	

#### P\_4\_04\_1.PAS \* Thuật toán Warshall liệt kê các thành phần liên thông

```
{$MODE DELPHI} (*This program uses 32-bit Integer [-231..231 - 1]*)  
program Connectivity;  
const  
  InputFile = 'CONNECT.INP';  
  OutputFile = 'CONNECT.OUT';
```