

CÁC THUẬT TOÁN TRÊN ĐỒ THỊ

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 kí hiệu, đó là đồ thị: một mô hình toán học gồm các đỉnh biểu diễn các đối tượng và các cạnh biểu diễn mối quan hệ giữa các đối tượng.

Những ý tưởng cơ bản của đồ thị được đưa ra từ thế kỷ thứ XVIII bởi nhà toán học Thụy Sĩ Leonhard Euler, năm 1736, ông đã dùng mô hình đồ thị để giải bài toán về bảy cây cầu Königsberg (*Seven Bridges of Königsberg*). Bài toán này cùng với bài toán mã đi tuần (*Knight Tour*) được coi là những bài toán đầu tiên của lí thuyết đồ thị.

Rất nhiều bài toán của lí thuyết đồ thị đã trở thành nổi tiếng và thu hút được sự quan tâm lớn của cộng đồng nghiên cứu. Ví dụ bài toán bốn màu, bài toán đằng cầu đồ thị, bài toán người du lịch, bài toán người đưa thư Trung Hoa, bài toán đường đi ngắn nhất, luồng cực đại trên mạng v.v... Trong phạm vi một chuyên đề, không thể trình bày tất cả những gì đã phát triển trong suốt gần 300 năm, chúng ta sẽ xem xét lí 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.



Leonhard Euler
1707-1783

1. Các khái niệm cơ bản

1.1. Đồ thị

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

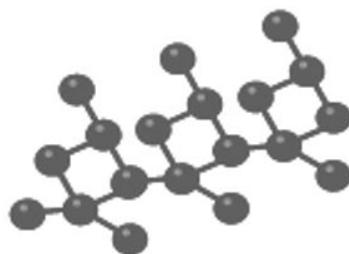
$$\begin{aligned} \text{Graph} &= \text{Objects} + \text{Connections} \\ G &= (V, E) \end{aligned}$$

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.

Một số hình ảnh của đồ thị:



Sơ đồ giao thông



Cấu trúc phân tử



Mạng máy tính

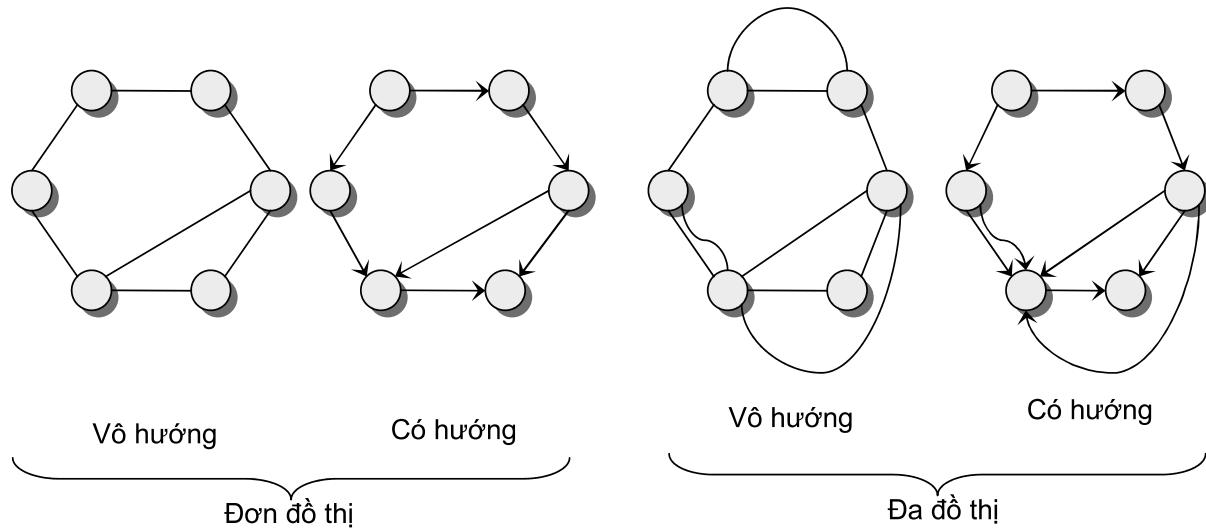
Hình 5-1. Ví dụ về mô hình đồ thị

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

- G được gọi là *đơn đồ thị* (hay gọi tắt là 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 u và 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*)
- 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 kì 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*). Đồ thị vô hướng cũng có thể coi là đồ thị có hướng nếu như ta coi cạnh nối hai đỉnh u, v bất kì tương đương với hai cung (u, v) và (v, u) .

Hình 5-2 là ví dụ về đơn đồ thị/đa đồ thị có hướng/vô hướng.



Hình 5-2. 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 là tập không quá đếm được, vì vậy ta có thể đánh số thứ tự 1, 2, 3... cho các phần tử của tập V và E và đồng nhất các phần tử của tập V và E với số thứ tự của chúng. 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.

a) 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 vô hướng, ta định nghĩa *bậc* (*degree*) của v , kí hiệu $\deg(v)$ là số cạnh liên thuộc với v . Trên đơn đồ thị thì số cạnh liên thuộc với v cũng là số đỉnh kề với v .

Định lí 5-1

|| Giả sử $G = (V, E)$ là đồ thị vô hướng, khi đó tổng tất cả các bậc đỉnh trong V sẽ bằng hai lần số cạnh:

$$\sum_{v \in V} \deg(v) = 2|E| \quad (1)$$

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: *Bán bậc ra (out-degree)* của v kí hiệu $\deg^+(v)$ là số cung đi ra khỏi nó; *bán bậc vào (in-degree)* kí hiệu $\deg^-(v)$ là số cung đi vào đỉnh đó.

Định lí 5-2

|| Giả sử $G = (V, E)$ là đồ thị có hướng, khi đó tổng tất cả các bán bậc ra của các đỉnh bằng tổng tất cả các bán bậc vào và bằng số cung của đồ thị

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

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.

b) 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 (path)*, đường đi này gồm $k + 1$ đỉnh p_0, p_1, \dots, p_k và k cạnh $(p_0, p_1), (p_1, p_2), \dots, (p_{k-1}, p_k)$. 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 , kí hiệu $p_0 \sim p_k$. Đỉnh p_0 được gọi là đỉnh đầu và đỉnh p_k gọi là đỉnh cuối của đường đi P . Các đỉnh p_1, p_2, \dots, p_{k-1} được gọi là *đỉnh trong* của đường đi P

Một đường đi gọi là *đơn giản* (*simple*) hay *đường đi đơn* nếu tất cả các đỉnh trên đường đi là hoàn toàn phân biệt (dù nhiên khi đó các cạnh trên đường đi cũng hoàn toàn phân biệt). Đường đi $P = \langle p_0, p_1, \dots, p_k \rangle$ trở thành *chu trình* (*circuit*) nếu $p_0 = p_k$. Trên đồ thị có hướng, chu trình P được gọi là *chu trình đơn* nếu nó có ít nhất một cung và các đỉnh p_1, p_2, \dots, p_k hoàn toàn phân biệt. Trên đồ thị vô hướng, chu trình P được gọi là chu trình đơn nếu $k \geq 3$ và các đỉnh p_1, p_2, \dots, p_k hoàn toàn phân biệt.

c) Một số khái niệm khác

Đẳng cấu

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 số cung nối u với v trên E bằng số cung nối $f(u)$ với $f(v)$ trên E' .

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 .

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) .

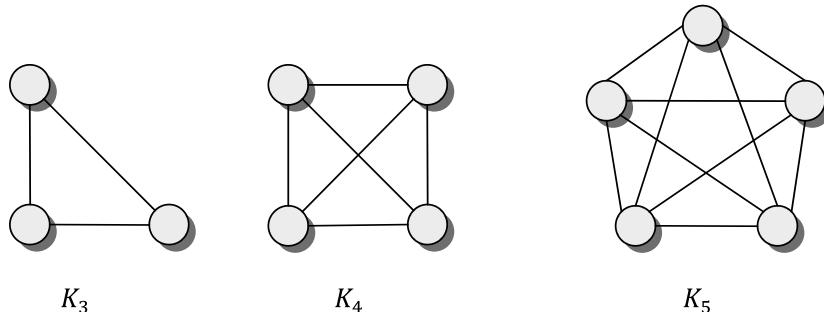
Với một đồ thị có hướng $G = (V, E)$, ta gọi *phiên bản vô hướng* (*undirected version*) của G là một đồ thị vô hướng $G' = (V, E')$ tạo thành bằng cách thay mỗi cung (u, v) bằng cạnh vô hướng (u, v) . Nói cách khác, G' tạo thành từ G bằng cách bỏ đi chiều của cung.

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

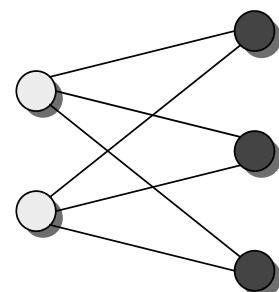
Một đồ thị vô hướng được gọi là *đầy đủ* (*complete*) nếu mọi cặp đỉnh đều là kề nhau, đồ thị đầy đủ gồm n đỉnh kí hiệu là K_n . Hình 5-3 là ví dụ về các đồ thị đầy đủ K_3 , K_4 và K_5 .



Hình 5-3. Đồ thị đầy đủ

Đồ thị hai phia

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 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 phia đó được gọi là đồ thị hai phia đầy đủ, kí hiệu $K_{m,n}$. Hình 5-4 là ví dụ về đồ thị hai phia đầy đủ $K_{2,3}$.



Hình 5-4. Đồ thị hai phia đầy đủ

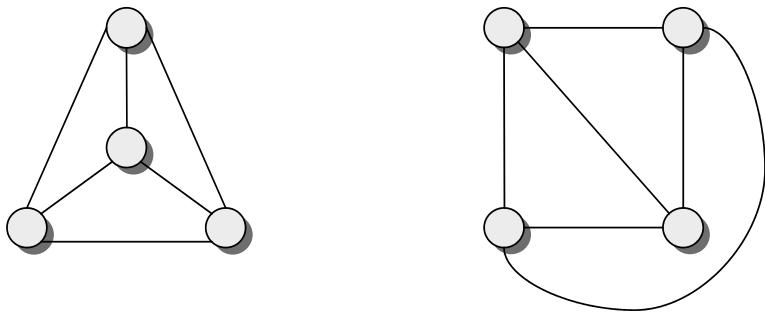
Đồ thị phẳng

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 toạ độ.
- Mỗi cạnh tương ứng với một đoạn đường liên tục nối hai đỉnh, các điểm nằm trên hai cạnh bất kì 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 5-5



Hình 5-5. Hai cách vẽ đồ thị phẳng của K_4

Định lí 5-3 (Định lí 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 lí 5-4 (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 lí 5-5

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 lí 5-5 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...

2. Biểu diễn đồ thị

Khi lập trình giải các bài toán được mô hình hóa 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.

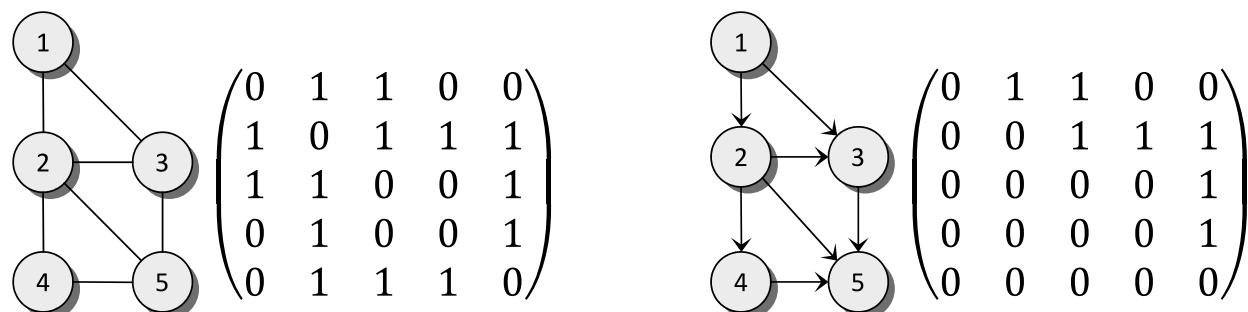
2.1. Ma trận kề

Với $G = (V, E)$ là một đơn đồ thị có hướng trong đó $|V| = n$, ta có thể đánh số các đỉnh từ 1 tới n 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 $A = \{a_{ij}\}_{n \times n}$. Trong đó:

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

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 .



Hình 5-6. Ma trận kề biểu diễn đồ thị

Trong trường hợp G là đơn đồ thị, ta có thể biểu diễn ma trận kề A tương ứng là các phần tử lôgic:

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

Có một cách khác biểu diễn đồ thị vô hướng G bằng ma trận $A = \{a_{ij}\}_{n \times n}$ như sau:

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

Cách biểu diễn này có ứng dụng trong một số bài toán đồ thị, gọi là biểu diễn bằng ma trận Laplace (*Laplacian matrix* hay *Kirchhoff matrix*)

Ma trận kè có một số tính chất:

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

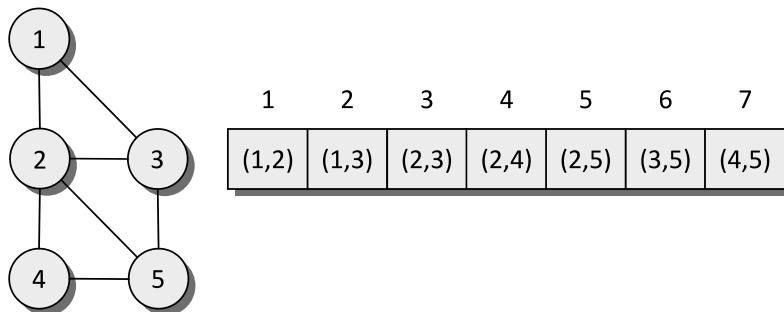
Ưu điểm của ma trận kè:

- 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 phép so sánh: $a_{uv} \neq 0$

Nhược điểm của ma trận kè

- Bất kè 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.
- Một số bài toán yêu cầu thao tác liệt kê tất cả các đỉnh v kè với một đỉnh u cho trước. Trên ma trận kè 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 kè với đỉnh nào) hoặc *đỉnh treo* (chỉ kè với 1 đỉnh) ta cũng buộc phải xét tất cả các đỉnh v và kiểm tra giá trị tương ứng a_{uv} .

2.2. Danh sách cạnh



Hình 5-7. 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 mốc nối.

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. (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 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).

2.3. 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 v của đồ thị, ta cho tương ứng với nó một danh sách các đỉnh kề với v .

Với đồ thị có hướng $G = (V, E)$. V gồm n đỉnh và E gồm m cung. 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 : $adj[v] = \{u: (u, v) \in E\}$

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.

Việc biểu diễn đồ thị vô hướng được quy về việc biểu diễn phiên bản có hướng tương ứng: thay mỗi cạnh (u, v) bởi hai cung có hướng ngược nhau: (u, v) và (v, u) .

Bất cứ cấu trúc dữ liệu nào có khả năng biểu diễn danh sách (mảng, danh sách mốc nối, cây...) đều có thể sử dụng để biểu diễn danh sách kè, nhưng mảng và danh sách mốc nối được sử dụng phổ biến nhất.

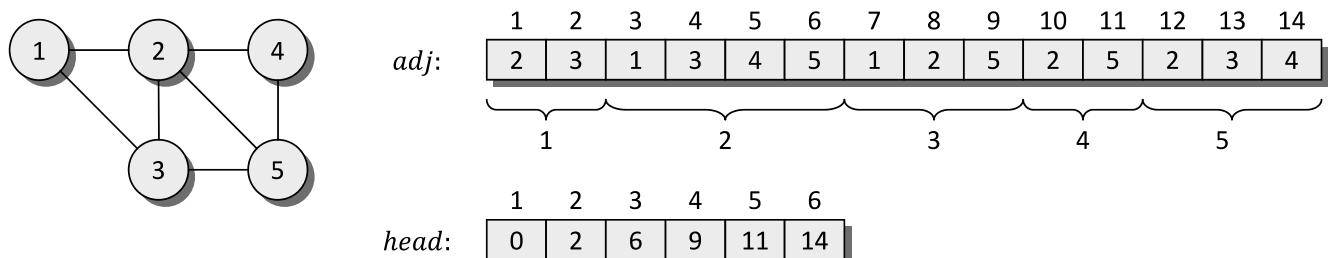
a) Biểu diễn danh sách kè bằng mảng

Dùng một mảng $adj[1 \dots m]$ chứa các đỉnh, mảng được chia làm n đoạn, đoạn thứ u trong mảng lưu danh sách các đỉnh kè với đỉnh u . Để biết một đoạn nằm từ chỉ số nào đến chỉ số nào, ta có một mảng $head[1 \dots n + 1]$ đánh dấu vị trí phân đoạn: $head[u]$ sẽ bằng chỉ số đứng liền trước đoạn thứ u , quy ước $head[n + 1] = m$. Khi đó các phần tử trong đoạn:

$$adj[head[u] + 1 \dots head[u + 1]]$$

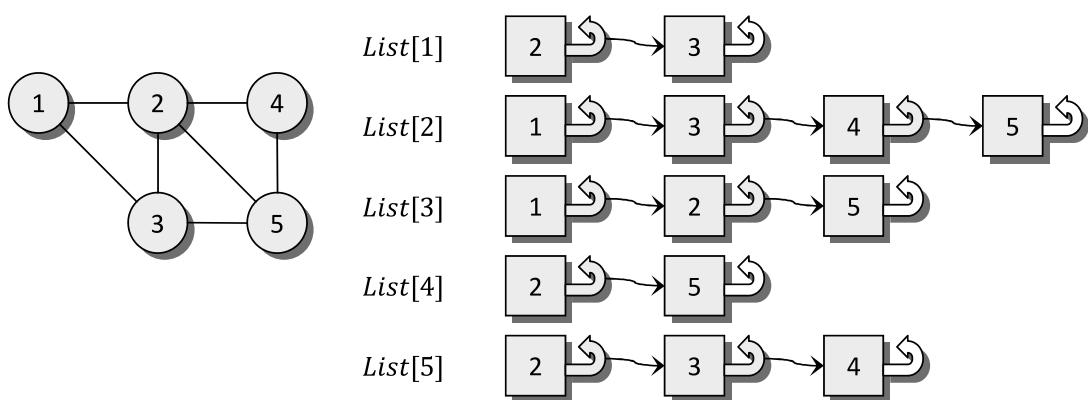
là các đỉnh kè với đỉnh u .

Nhắc lại rằng khi sử dụng danh sách kè để biểu diễn đồ thị vô hướng, ta quy nó về đồ thị có hướng và số cung m được nhân đôi (Hình 5-8).



Hình 5-8. Dùng mảng biểu diễn danh sách kè

b) Biểu diễn danh sách kè bằng các danh sách mốc nối



Hình 5-9. Biểu danh sách kè bởi các danh sách mốc nối

Trong cách biểu diễn này, ta cho tương ứng mỗi đỉnh u của đồ thị với $List[u]$ là chốt của một danh sách mốc gồm các đỉnh kề với u .

Ưu điểm của danh sách kề

- Đối với danh sách kề, việc duyệt tất cả các đỉnh kề với một đỉnh v cho trước là hết sức dễ dàng, cái tên “danh sách kề” 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 kề nó.

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 .

2.4. 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.

Đặc biệt trong trường hợp đồ thị có hướng, ta có thể xây dựng danh sách liên thuộc từ danh sách cạnh tương đối dễ dàng bằng cách bổ sung các con trỏ liên kết. Giả sử đồ thị có hướng $G = (V, E)$ có n đỉnh và m cung được biểu diễn bởi danh sách cạnh $e[1 \dots m]$. Vì đồ thị có hướng, nếu ta cho tương ứng mỗi đỉnh u một danh sách các cung đi ra khỏi u (forward star) thì sẽ có tổng cộng n danh sách liên thuộc và mỗi cung chỉ xuất hiện trong đúng một danh sách liên thuộc. Vì vậy ta có thể bổ sung hai mảng $head[1 \dots n]$ và $link[1 \dots m]$ trong đó:

- $head[u]$ là chỉ số cung đầu tiên trong danh sách liên thuộc của đỉnh u . Nếu danh sách liên thuộc đỉnh u là \emptyset , $head[u]$ được gán bằng 0.
- $link[i]$ là chỉ số cung kế tiếp cung $e[i]$ trong danh sách liên thuộc chứa cung $e[i]$. Trường hợp $e[i]$ là cung cuối cùng của một danh sách liên thuộc, $link[i]$ được gán bằng 0

Để duyệt tất cả những cung đi ra khỏi một đỉnh u nào đó, ta có thể thực hiện dễ dàng bằng thuật toán sau:

```
i := head[u];
while i ≠ 0 do
    begin
```

```

«Xử lí cung e[i]»;
i := link[i];
end;

```

2.5. 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.

Ta xét bài toán chuyển đổi các cách biểu diễn đơn đồ thị có hướng $G = (V, E)$ có n đỉnh và m cạnh. Có thể biểu diễn đồ thị này bởi:

Ma trận kè:

```

var
  a: array[1..n, 1..n] of Boolean;

```

Danh sách cạnh:

```

type
  TEdge = record
    x, y: Integer;
  end;
var
  e: array[1..m] of TEdge;

```

Danh sách kè (forward star) (biểu diễn bằng mảng):

```

var
  adj: array[1..2 * m] of Integer;
  head: array[1..n + 1] of Integer;

```

Danh sách liên thuộc (forward star) (biểu diễn bằng cấu trúc liên kết)

```

type
  TEdge = record
    x, y: Integer;
  end;
var
  e: array[1..m] of TEdge; //Danh sách cạnh
  link: array[1..m] of Integer; //link[i]: chỉ số cạnh kế tiếp trong danh sách
  liên thuộc
  head: array[1..n] of Integer; //head[i]: chỉ số cạnh đầu
  tiên trong danh sách liên thuộc

```

a) Chuyển đổi giữa ma trận kè và danh sách cạnh

Nếu đồ thị G được cho bởi ma trận kè $A = \{a_{ij}\}_{n \times n}$ trong đó $a_{ij} = \text{True} \Leftrightarrow (i, j) \in E$, ta có thể xây dựng danh sách cạnh tương ứng bằng cách duyệt tất cả các cặp (i, j) , nếu $a_{ij} = \text{True}$ thì đưa cặp này vào danh sách cạnh e .

```
k := 0;
for i := 1 to n do
    for j := 1 to n do
        if a[i, j] then
            begin
                k := k + 1;
                e[k].x := i; e[k].y := j;
            end;
```

Ngược lại, nếu đồ thị G cho bởi danh sách cạnh e , ta có thể xây dựng ma trận kè A bằng cách khởi tạo các phần tử của A là False rồi duyệt danh sách cạnh, mỗi khi duyệt qua cung (x, y) , ta đặt $a_{xy} := \text{True}$.

```
for i := 1 to n do
    for j := 1 to n do a[i, j] := False;
for k := 1 to m do
    with e[k] do
        a[x, y] := True;
```

b) Chuyển đổi giữa ma trận kè và danh sách kè

Từ ma trận kè $A = \{a_{ij}\}_{n \times n}$, ta có thể xây dựng hai mảng $adj[1 \dots m]$ và $head[1 \dots n + 1]$ sao cho các phần tử trong mảng adj từ chỉ số $head[u] + 1$ tới chỉ số $head[u + 1]$ chứa danh sách kè của đỉnh u . Hai mảng này là danh sách kè dạng forward star của đồ thị.

```
head[n + 1] := m;
for i := n downto 1 do
    begin
        head[i] := head[i + 1];
        for j := n downto 1 do
            if a[i, j] then
                begin
                    adj[head[i]] := j;
                    head[i] := head[i] - 1;
                end;
    end;
```

Ngược lại, chúng ta có thể xây dựng ma trận kè từ danh sách kè theo cách: Đặt các phần tử của ma trận kè A bằng $False$, sau đó với mỗi đỉnh u , duyệt các đỉnh v thuộc danh sách kè của nó và đặt $a_{uv} := True$.

```

for i := 1 to n do
    for j := 1 to n do a[i, j] := False;
for u := 1 to n do
    for k := head[u] + 1 to head[u + 1] do
        a[u, adj[k]] := True;

```

c) Chuyển đổi giữa danh sách cạnh và danh sách kè

Từ danh sách cạnh e , ta có thể xây dựng hai mảng adj và $head$ tương ứng với danh sách kè dạng forward star bằng thuật toán đếm phân phôi.

Trước hết, ta tính các $head[u]$ là bậc của đỉnh u ($\forall u \in V$):

```

for u := 1 to n do head[u] := 0;
for i := 1 to m do
    with e[i] do
        head[x] := head[x] + 1;

```

Sau đó, ta chia mảng adj thành n đoạn, đoạn thứ u sẽ chứa các đỉnh kè với đỉnh u . Để xác định vị trí các đoạn này, ta đặt mỗi $head[u]$ trở tới vị trí cuối đoạn thứ u :

```

for u := 2 to n do
    head[u] := head[u - 1] + head[u];

```

Tiếp theo là duyệt lại danh sách cạnh, mỗi khi duyệt tới cạnh (x, y) ta đưa y vào mảng adj tại vị trí $head[x]$, đưa x vào mảng adj tại vị trí $head[y]$ đồng thời giảm hai con trỏ $head[x]$ và $head[y]$ đi 1.

```

for i := m downto 1 do
    with e[i] do
        begin
            adj[head[x]] := y; head[x] := head[x] - 1;
        end;

```

Đến đây, chúng ta có mảng adj phân làm n đoạn, trong đó $head[u]$ là vị trí đứng liền trước đoạn thứ u . Việc cuối cùng là đặt:

```
head[n + 1] := m;
```

Việc chuyển đổi từ danh sách kè sang danh sách cạnh được thực hiện đơn giản hơn: Với mỗi đỉnh u , ta xét các đỉnh v thuộc danh sách kè của nó và đưa (u, v) vào danh sách cạnh e .

```

i := 0;
for u := 1 to n do
    for k := head[u] + 1 to head[u + 1] do
        begin
            v := adj[k];
            i := i + 1;
            e[i].x := u; e[i].y := v;
        end;

```

d) Chuyển đổi giữa danh sách cạnh và danh sách liên thuộc

Bởi danh sách liên thuộc được đặc tả đã bao gồm danh sách cạnh, ta chỉ quan tâm tới vấn đề chuyển đổi từ danh sách cạnh thành danh sách liên thuộc.

Trước hết với mọi đỉnh u ta đặt $head[u] := 0$ để khởi tạo danh sách liên thuộc của u bằng \emptyset .

```
for u := 1 to n do head[u] := 0;
```

Tiếp theo ta duyệt danh sách cạnh, mỗi khi duyệt qua một cung (x, y) ta mốc nối cung này vào danh sách liên thuộc các cung đi ra khỏi x :

```

for i := m downto 1 do
    with e[i] do
        begin
            link[i] := head[x];
            head[x] := i;
        end;

```

Với các bài toán mà chúng ta sẽ khảo sát, cũng có một số thuật toán không phụ thuộc nhiều và cách biểu diễn đồ thị, trong trường hợp này tôi sẽ chọn cấu trúc dữ liệu dễ cài đặt và trình bày nhất để việc đọc hiểu thuật toán/chương trình được thuận tiện hơn.

Bài tập

- 5.1. Cho một đồ thị có hướng n đỉnh, m cạnh được biểu diễn bằng danh sách kè, 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
- 5.2. 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 kè; G và G^T được biểu diễn bằng danh sách kè.

- 5.3. Cho đa đồ thị vô hướng $G = (V, E)$ được biểu diễn bằng danh sách kè, hãy tìm thuật toán $O(|V| + |E|)$ để xây dựng đơn đồ thị $G' = (V, E')$ và biểu diễn G' bằng danh sách kè, 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' .
- 5.4. Cho đa đồ thị G được biểu diễn bằng ma trận kè $A = \{a_{ij}\}$ trong đó a_{ij} là số cạnh nối từ đỉnh i tới đỉnh j . Hãy chứng minh rằng A^k là ma trận $B = \{b_{ij}\}$ trong đó b_{ij} là số đường đi từ đỉnh i tới đỉnh j qua đúng k cạnh. Gợi ý: Sử dụng chứng minh quy nạp.
- 5.5. 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 kè, tìm thuật toán $O(|E| + |V|^2)$ để xây dựng G^2 từ G trong trường hợp cả G và G^2 được biểu diễn bằng danh sách kè.

- 5.6. Xây dựng cấu trúc dữ liệu để biểu diễn đồ thị vô hướng và các thao tác:
 - Liệt kê các đỉnh kè với một đỉnh cho trước trong thời gian $O(|E|)$
 - Kiểm tra hai đỉnh có kè nhau hay không trong thời gian $O(1)$
 - Loại bỏ một cạnh trong thời gian $O(1)$
- 5.7. Với đồ thị $G = (V, E)$ được biểu diễn bằng ma trận kè, 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 chúa” (*universal sink*) trong đồ thị: bồn chúa 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 chúa trong đồ thị có hướng.
- 5.8. Người ta còn có thể biểu diễn đồ thị bằng *ma trận liên thuộc* (*incidence matrix*): Với đồ thị có hướng $G = (V, E)$ có n đỉnh và m cung, ma trận liên thuộc $B = \{b_{ij}\}$ của G kích thước $m \times n$, trong đó:

$$b_{ij} = \begin{cases} -1, & \text{nếu cung thứ } j \text{ đi ra khỏi đỉnh } i \\ 1, & \text{nếu cung thứ } j \text{ đi vào đỉnh } i \\ 0, & \text{nếu cung thứ } j \text{ không liên thuộc với đỉnh } i \end{cases}$$

Xét B^T là ma trận chuyển vị của ma trận B , hãy cho biết ý nghĩa của ma trận tích BB^T

3. Các thuật toán tìm kiếm trên đồ thị

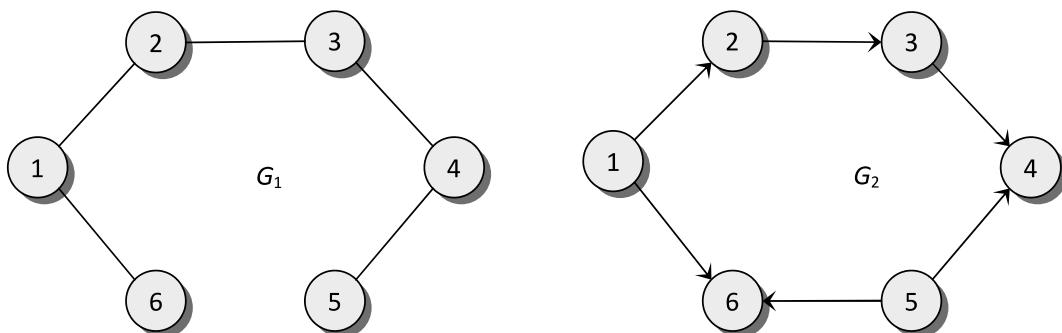
3.1. Bài toán tìm đường

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 : $s \sim t$.



Hình 5-10: Đồ thị và đường đi

Trên cả hai đồ thị ở Hình 5-10, $\langle 1,2,3,4 \rangle$ là đường đi từ đỉnh 1 tới đỉnh 4. $\langle 1,6,5,4 \rangle$ không phải đường đi vì không có cạnh (cung) $(6,5)$.

Một bài toán quan trọng trong lí 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 kì 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.

Trong những chương trình cài đặt dưới đây, ta giả thiết rằng đồ thị được cho là đồ thị có hướng, số đỉnh không quá 10^5 , số cung không quá 10^6 , các đỉnh được đánh

số từ 1 tới n và đồng nhất với số hiệu của chúng. Khuôn dạng Input/Output quy định cụ thể như sau:

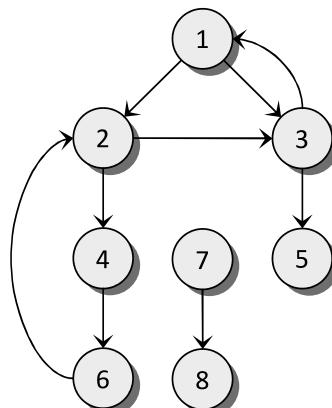
Input

- Dòng 1 chứa số đỉnh n , đỉnh xuất phát s và đỉnh cần đến t .
- n dòng tiếp theo, dòng thứ i chứa một danh sách các đỉnh, mỗi đỉnh j trong danh sách tương ứng với một cung (i, j) của đồ thị, ngoài ra có thêm một số 0 ở cuối dòng để báo hiệu kết thúc.

Output

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

Sample Input	Sample Output
8 1 5	Reachable vertices from 1: 1, 2, 3, 5, 4, 6,
2 3 0	The path from 1 to 5: 5<-3<-2<-1
3 4 0	
1 5 0	
6 0	
0	
2 0	
8 0	
0	



3.2. Biểu diễn đồ thị

Đồ thị được biểu diễn bằng danh sách kè dạng forward star, mỗi đỉnh u sẽ được cho tương ứng với một danh sách các đỉnh nối từ u . Nếu đồ thị có n đỉnh thì có tổng cộng n danh sách kè, gọi m là tổng số phần tử trên tất cả các danh sách kè. Khi đó $m = |E|$, như đã quy ước, $m \leq 10^6$.

Cấu trúc dữ liệu được cài đặt bằng mảng $adj[1 \dots m]$ mảng này được chia làm n đoạn liên tiếp, đoạn thứ u chứa danh sách các đỉnh nối từ u . Vị trí của các đoạn được xác định bởi mảng $head[0 \dots n]$ trong đó $head[u]$ là vị trí cuối đoạn thứ u , quy ước $head[0] = 0$. Như vậy các đỉnh nối từ u sẽ nằm liên tiếp trong mảng adj từ chỉ số $head[u - 1] + 1$ tới chỉ số $head[u]$.

3.3. Thuật toán tìm kiếm theo chiều sâu

a) 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 thủ tục 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 .

Kĩ thuật đánh dấu được sử dụng để tránh việc liệt kê lặp các đỉnh: Khoi tạo $avail[v] := True$, $\forall v \in V$, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại ($avail[v] := False$) 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 thủ tục $DFSVisit(u)$, 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:

$$\langle p_1 = t \leftarrow p_2 = trace[p_1] \leftarrow p_3 = trace[p_2] \leftarrow \dots \leftarrow s \rangle$$

```
procedure DFSVisit (u∈V); //Thuật toán tìm kiếm theo chiều sâu từ đỉnh u
begin
    avail[u] := False; //avail[u] = False ⇔ u đã thăm
    Output ← u; //Liệt kê u
    for ∀v∈V: (u, v)∈E do //Duyệt mọi đỉnh v chưa thăm nối từ u
        if avail[v] then
            begin
                trace[v] := u; //Lưu vết đường đi, đỉnh liền trước v trên đường đi
                từ s tới v là u
                DFSVisit(v); //Gọi đệ quy để tìm kiếm theo chiều sâu từ đỉnh v
            end;
    end;
    begin //Chương trình chính
        Input → Đồ thị G, đỉnh xuất phát s, đỉnh đích t;
        for ∀v∈V do avail[v] := True; //Đánh dấu mọi đỉnh đều chưa thăm
        DFSVisit(s);
        if avail[t] then //s đi tới được t
            «Truy theo vết từ t để tìm đường đi từ s tới t»;
    end.
```

b) Cài đặt

DFS.PAS ✓ Tìm đường bằng DFS

```
{$MODE OBJFPC}
{$M 4000000}
program DepthFirstSearch;
const
  maxN = 100000;
  maxM = 1000000;
var
  adj: array[1..maxM] of Integer; //Các danh sách kề
  head: array[0..maxN] of Integer; //Mảng đánh dấu vị trí cắt đoạn trong
  adj
  avail: array[1..maxN] of Boolean;
  trace: array[1..maxN] of Integer;
  n, s, t: Integer;
procedure Enter; //Nhập dữ liệu
var
  u, v, i: Integer;
begin
  ReadLn(n, s, t);
  i := 0;
  for u := 1 to n do
    begin //Đọc danh sách kề của u
      repeat
        read(v);
        if v <> 0 then //Thêm v vào mảng adj
          begin
            Inc(i); adj[i] := v;
          end;
      until v = 0;
      head[u] := i; //Đọc hết một dòng, đánh dấu vị trí cắt đoạn thứ u
      ReadLn;
    end;
    head[0] := 0; //Cầm canh
  end;
procedure DFSSvisit(u: Integer); //Thuật toán tìm kiếm theo chiều sâu bắt
  đầu từ u
var
  i: Integer;
begin
  avail[u] := False;
```

```

Write(u, ', ') ; //Liệt kê u
for i := head[u - 1] + 1 to head[u] do //Duyệt các đỉnh adj[i] nối từ u
    if avail[adj[i]] then
        begin
            trace[adj[i]] := u;
            DFSVisit(adj[i]);
        end;
    end;
procedure PrintPath; //In đường đi từ s tới t
begin
    if avail[t] then //Từ s không có đường tới t
        WriteLn(' There is no path from ', s, ' to ', t)
    else
        begin
            WriteLn('The path from ', s, ' to ', t, ':');
            while t <> s do //Truy vết ngược từ t về s
                begin
                    Write(t, '->');
                    t := trace[t];
                end;
            WriteLn(s);
        end;
    end;
begin
    Enter;
    FillChar(avail[1], n * SizeOf(avail[1]), True);
    WriteLn('Reachable vertices from ', s, ': ');
    DFSVisit(s);
    WriteLn;
    PrintPath;
end.

```

Có thể không cần mảng đánh dấu $avail[1 \dots n]$ mà dùng luôn mảng $trace[1 \dots n]$ để đánh dấu: Khởi tạo các phần tử mảng $trace[1 \dots n]$ là:

$$\begin{cases} trace[s] \neq 0 \\ trace[v] = 0, \forall v \neq s \end{cases}$$

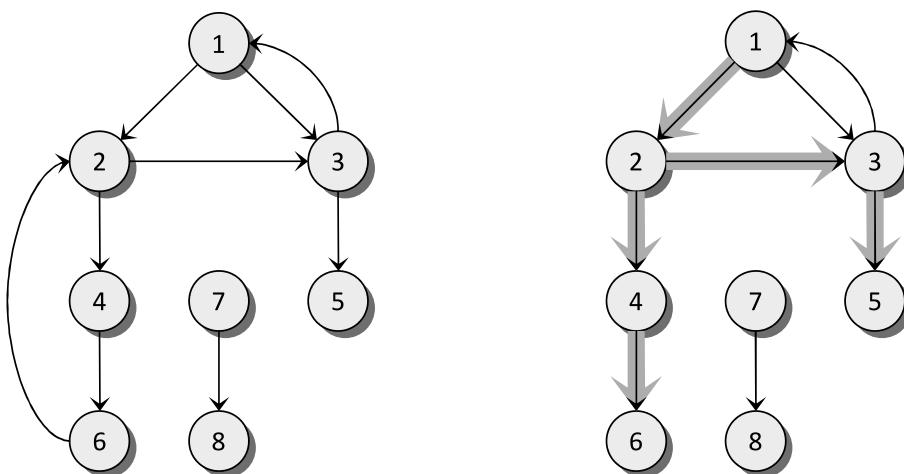
Khi đó điều kiện để một đỉnh v chưa thăm là $trace[v] = 0$, mỗi khi từ đỉnh u thăm đỉnh v , phép gán $trace[v] := u$ sẽ kiêm luôn công việc đánh dấu v đã thăm ($trace[v] \neq 0$).

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 ($DFSVisit(u)$ gọi $DFSVisit(v)$) thì u là nút cha của nút v . Hình 5-11 là đồ thị và cây DFS tương ứng với đỉnh xuất phát $s = 1$.



Hình 5-11: Đồ thị và cây DFS

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:

```
procedure DFSVisit (u∈V); //Thuật toán tìm kiếm theo chiều sâu từ đỉnh u
begin
    Time := Time + 1;
    d[u] := Time;
    Output ← u; //Liệt kê u
    for ∀v∈V: (u, v)∈E do //Duyệt mọi đỉnh v nối từ u
        if d[v] = 0 then DFSVisit (v); //Nếu v chưa thăm, gọi đệ quy để tìm
        kiêm theo chiều sâu từ đỉnh v
        Time := Time + 1;
        f[u] := Time;
    end;
begin //Chương trình chính
Input → Đồ thị G
```

```

for ∀v∈V do d[v] := 0; //Mọi đỉnh đều chưa được duyệt đến
Time := 0;
for ∀v∈V do
    if d[v] = 0 then DFSVisit(v);
end.

```

Thuật toán này sẽ thăm tất cả các đỉnh và các cạnh của đồ thị và thứ tự thăm được gọi là thứ tự duyệt DFS. Như ví dụ ở đồ thị trong bài, thứ tự thăm DFS với các đỉnh là:

1, 2, 3, 5, 4, 6, 7, 8

Thứ tự thăm DFS với các cạnh là:

(1,2); (2,3); (3,1); (3,5); (2,4); (4,6); (6,2); (1,3); (7,8)

Thời gian thực hiện giải thuật của DFS có thể đánh giá bằng số lần gọi thủ tục *DFSVisit* ($|V|$ lần) cộng với số lần thực hiện của vòng lặp for bên trong thủ tục *DFSVisit*. Chính vì vậy:

- Nếu đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, vòng lặp for bên trong thủ tục *DFSVisit* (xét tổng thể cả chương trình) sẽ duyệt qua tất cả các cạnh của đồ thị (mỗi cạnh hai lần nếu là đồ thị vô hướng, mỗi cạnh một lần nếu là đồ thị có hướng). Trong trường hợp này, thời gian thực hiện giải thuật DFS là $\Theta(|V| + |E|)$
- Nếu đồ thị được biểu diễn bằng ma trận kề, vòng lặp for bên trong mỗi thủ tục *DFSVisit* sẽ phải duyệt qua tất cả các đỉnh 1 ... n . Trong trường hợp này thời gian thực hiện giải thuật DFS là $\Theta(|V| + |V|^2) = \Theta(|V|^2)$.
- Nếu đồ thị được biểu diễn bằng danh sách cạnh , vòng lặp for bên trong thủ tục *DFSVisit* sẽ phải duyệt qua tất cả danh sách cạnh mỗi lần thực hiện thủ tục. Trong trường hợp này thời gian thực hiện giải thuật DFS là $\Theta(|V||E|)$.

Thứ tự duyệt đến và duyệt xong

Hãy để ý thủ tục *DFSVisit*(u):

- Khi bắt đầu vào thủ tục ta nói đỉnh u được *duyệt đến* hay được *thăm* (*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 thủ tục để lùi về , ta nói đỉnh u được *duyệt 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, chúng ta sử dụng một biến đếm *Time* để xác định thời điểm duyệt đến d_u và thời điểm duyệt xong f_u của mỗi đỉnh u . Thứ tự

duyệt 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 lí 5-6

Với hai đỉnh phân biệt u, v :

- *Đỉnh v được duyệt 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 duyệt đến (hay duyệt xong) trong thời gian từ d_u đến f_u chính là thủ tục $DFSVisit(v)$ được gọi (hay thoát) khi mà thủ tục $DFSVisit(u)$ đã bắt đầu nhưng chưa kết thúc, nghĩa là thủ tục $DFSVisit(v)$ 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 chung nhau. Hai đoạn $[d_u, f_u]$ và $[d_v, f_v]$ chung 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 lí 5-6, ta có ĐPCM.

Định lí 5-7

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

$$(u, v) \in E \Rightarrow d_v < f_u \quad (0.1)$$

Chứng minh

Đây là một tính chất quan trọng của thuật toán DFS. Hãy để ý thủ tục $DFSVisit(u)$, trước khi thoát (duyệt 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 v phải được duyệt đến trước khi u được duyệt xong: $d_v < f_u$.

Định lí 5-8 (định lí 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 lí 5-6, ta có $d_u < d_w$, tức là vào thời điểm d_u , tất cả các đỉnh w đó đều chưa được thăm

“ \Leftarrow ”

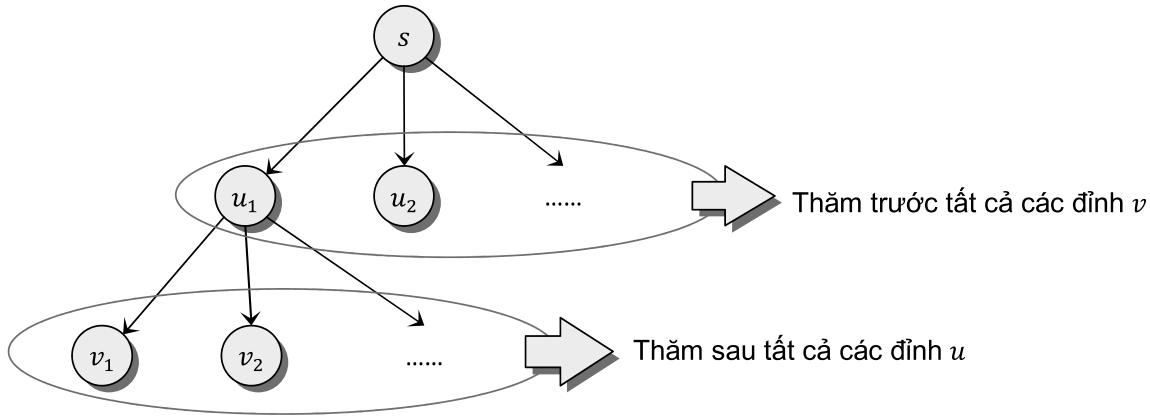
Nếu tại thời điểm d_u , tồn tại một đường đi từ u tới v mà ngoại trừ đỉnh u , tất cả các đỉnh khác trên đường đi đều chưa được thăm, ta sẽ chứng minh rằng mọi đỉnh trên đường đi này đều là hậu duệ của u . Thật vậy, giả sử phản chứng rằng v^* là đỉnh đầu tiên trên đường đi này mà không phải hậu duệ của u , tức là tồn tại đỉnh w liền trước v^* trên đường đi là hậu duệ của u . Theo Định lí 5-7, v^* phải được thăm trước khi duyệt xong w : $d_{v^*} < f_w$; w lại là hậu duệ của u nên theo Định lí 5-6, ta có $f_w \leq f_u$, vậy $d_{v^*} < f_u$. Mặt khác theo giả thiết rằng tại thời điểm d_u thì v^* chưa được thăm, tức là $d_u < d_{v^*}$, kết hợp lại ta có $d_{v^*} \in [d_u, f_u]$, vậy v^* là hậu duệ của u theo Định lí 5-6, trái với giả thiết phản chứng.

Tên gọi “định lí 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 duyệt 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 lí 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.

3.4. Thuật toán tìm kiếm theo chiều rộng

a) Ý tưởng

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_1, u_2, \dots nối từ s (những đỉnh gần s nhất). Tiếp theo ta thăm đỉnh u_1 , khi thăm đỉnh u_1 sẽ lại phát sinh yêu cầu thăm những đỉnh v_1, v_2, \dots nối từ u_1 . 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: $s, u_1, u_2, \dots, v_1, v_2, \dots$



Hình 5-12: 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): 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 BFS có thể viết như sau:

```

Queue := (s); //Khởi tạo hàng đợi chỉ gồm một đỉnh s
for ∀v∈V do
    avail[v] := True;
    avail[s] := False; //Đánh dấu chỉ có đỉnh s được xếp hàng
repeat //Lặp tới khi hàng đợi rỗng
    u := Pop; //Lấy từ hàng đợi ra một đỉnh u
    Output ← u; //Liệt kê u
    for ∀v∈V:avail[v] and (u, v)∈E do //Xét những đỉnh v kề u
        chưa được đẩy vào hàng đợi
        begin
            trace[v] := u; //Lưu vết đường đi
            Push(v); //Đẩy v vào hàng đợi
            avail[v] := False; //Đánh dấu v đã xếp hàng
        end;
until Queue = ∅;
if avail[t] then //s đi tới được t
    «Truy theo vết từ t để tìm đường đi từ s tới t»;
```

b) Cài đặt

BFS.PAS ✓ Tìm đường bằng BFS

```
{$MODE OBJFPC}
program Breadth_First_Search;
const
  maxN = 100000;
  maxM = 1000000;
var
  adj: array[1..maxM] of Integer; //Các danh sách kề
  head: array[0..maxN] of Integer; //Mảng đánh dấu vị trí cắt đoạn trong adj
  avail: array[1..maxN] of Boolean;
  trace: array[1..maxN] of Integer;
  n, s, t: Integer;
  Queue: array[1..maxN] of Integer;
  front, rear: Integer;
procedure Enter; //Nhập dữ liệu
var
  u, v, i: Integer;
begin
  ReadLn(n, s, t);
  i := 0;
  for u := 1 to n do
    begin //Đọc danh sách kề của u
      repeat
        read(v);
        if v <> 0 then //Thêm v vào mảng adj
          begin
            Inc(i); adj[i] := v;
          end;
      until v = 0;
      head[u] := i; //Đọc hết một dòng, đánh dấu vị trí cắt đoạn thứ u
      ReadLn;
    end;
  head[0] := 0; //Cầm canh
end;
procedure BFS; //Thuật toán tìm kiếm theo chiều rộng
var
  u, i: 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 có mỗi một đỉnh s
```

```

FillChar(avail[1], n * SizeOf(avail[1]), True); //Các đỉnh đều
chưa xếp hàng
avail[s] := False; //ngoại trừ đỉnh s đã xếp hàng
repeat
    u := Queue[front]; Inc(front); //Lấy từ hàng đợi ra một đỉnh u
    Write(u, ', ', );
    for i := head[u - 1] + 1 to head[u] do //Duyệt những đỉnh adj[i]
nối từ u
        if avail[adj[i]] then //Nếu đỉnh đó chưa thăm
            begin
                Inc(rear); Queue[rear] := adj[i]; //Đẩy vào hàng đợi
                avail[adj[i]] := False;
                trace[adj[i]] := u; //Lưu vết đường đi
            end;
        until front > rear;
end;
procedure PrintPath; //In đường đi từ s tới t
begin
    if avail[t] then //Từ s không có đường tới t
        WriteLn(' There is no path from ', s, ' to ', t)
    else
        begin
            WriteLn('The path from ', s, ' to ', t, ':');
            while t <> s do //Truy vết ngược từ t về s
                begin
                    Write(t, '<-');
                    t := trace[t];
                end;
            WriteLn(s);
        end;
end;
begin
    Enter;
    WriteLn('Reachable vertices from ', s, ': ');
    BFS;
    WriteLn;
    PrintPath;
end.

```

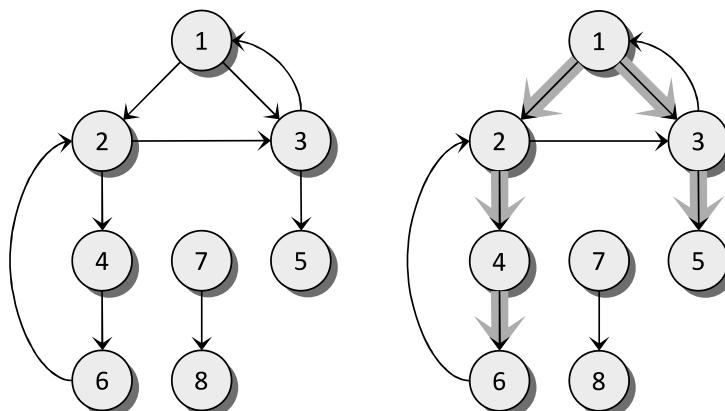
Tương tự như thuật toán tìm kiếm theo chiều sâu, ta có thể dùng mảng $Trace[1 \dots n]$ kiêm luôn chức năng đánh dấu.

c) Một vài tính chất của BFS

Cây BFS

Nếu ta sắp xếp các danh sách kề của mỗi đỉnh theo thứ tự tăng dần thì thuật toán BFS luôn trả về đường đi qua ít cạnh nhất trong số tất cả các đường đi từ s tới t . Nếu có nhiều đường đi từ s tới t đều qua ít cạnh nhất thì thuật toán BFS sẽ trả về đường đi có thứ tự từ điển nhỏ nhất trong số những đường đi đó.

Quá trình tìm kiếm theo chiều rộng cho ta một cây BFS gốc s . Khi thuật toán kết thúc $Trace[v]$ chính là nút cha của nút v trên cây. Hình 5-13 là đồ thị và cây BFS tương ứng với đỉnh xuất phát $s = 1$.



Hình 5-13: Đồ thị và cây BFS

Mô hình duyệt đồ thị theo BFS

Tương tự như thuật toán DFS, trên thực tế, thuật toán BFS cũng dùng để xác định một thứ tự trên các đỉnh của đồ thị và được viết theo mô hình sau:

```

procedure BFSVisit (s∈V);
begin
    Queue := (s); //Khởi tạo hàng đợi chỉ gồm một đỉnh s
    Time := Time + 1;
    d[s] := Time; //Duyệt đến đỉnh s
    repeat //Lặp tới khi hàng đợi rỗng
        u := Pop; //Lấy từ hàng đợi ra một đỉnh u
        Time := Time + 1;
        f[u] := Time; //Ghi nhận thời điểm duyệt xong đỉnh u
        Output ← u; //Liệt kê u
        for ∀v∈V: (u, v)∈E do //Xét những đỉnh v kề u
            if d[v] = 0 then //Nếu v chưa duyệt đến
                begin
                    Push(v); //Đẩy v vào hàng đợi
    
```

```

        Time := Time + 1;
        d[v] := Time; //Ghi nhận thời điểm duyệt đến đỉnh v
    end;
    until Queue = Ø;
end;
begin //Chương trình chính
    Input → Đồ thị G;
    for ∀v∈V do d[v] := 0; //Mọi đỉnh đều chưa được duyệt đến
    Time := 0;
    for ∀v∈V do
        if avail[v] then BFSVisit(v);
    end.

```

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 duyệt đến và duyệt xong

Tương tự như thuật toán DFS, đối với thuật toán BFS người ta cũng quan tâm tới thứ tự duyệt đến và duyệt xong: Khi một đỉnh được đẩy vào hàng đợi, ta nói đỉnh đó được duyệt 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 duyệt đến d_u và thời điểm duyệt xong d_v

Vì cách hoạt động của hàng đợi: đỉnh nào duyệt đến trước sẽ phải duyệt xong trước, chính vì vậy, việc liệt kê các đỉnh có thể thực hiện khi chúng được duyệt 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 duyệt 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 duyệt đến/chưa duyệt đến bằng duyệt xong/chưa duyệt xong:

```

Input → Đồ thị G;
for ∀v∈V do avail[v] := True; //Đánh dấu mọi đỉnh đều chưa duyệt xong
Queue := Ø;
for ∀v∈V do Push(v); //Khởi tạo hàng đợi chứa tất cả các đỉnh
repeat //Lặp tới khi hàng đợi rỗng
    u := Pop; //Lấy từ hàng đợi ra một đỉnh u
    if avail[u] then //Nếu u chưa duyệt xong
        begin

```

```

Output ← u; //Liệt kê u
avail[u] := False; //Đánh dấu u đã duyệt xong
for ∀v∈V: avail[v] and ((u, v) ∈ E) do //Xét những đỉnh v kề u chưa
    duyệt xong
        begin
            trace[v] := u; //Lưu vết đường đi
            Push(v); //Đẩy v vào hàng đợi
        end;
until 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 duyệt đến/chưa duyệt 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 $\sum_{v \in V} (1 + \deg(v)) = 2m + n$ ô nhớ cho *Queue*. Con số này đối với đồ thị có hướng là $m + n$ 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 nếu thay cấu trúc hàng đợi bởi cấu trúc ngăn xếp trong cách cài đặt thứ hai, ta sẽ được thứ tự duyệt đỉnh DFS. 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

- 5.9.** Viết chương trình cài đặt thuật toán DFS không đệ quy.
- 5.10.** 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 duyệt đến trước v , thì v nằm trong nhánh DFS gốc u .
- 5.11.** Cho đồ thị vô hướng $G = (V, E)$, tìm thuật toán $O(|V|)$ để phát hiện một chu trình đơn trong G .
- 5.12.** 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.
- 5.13.** Mê cung hình chữ nhật kích thước $m \times n$ gồm các ô vuông đơn vị ($m, n \leq 1000$). 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.

4. Tính liên thông của đồ thị

4.1. Định nghĩa

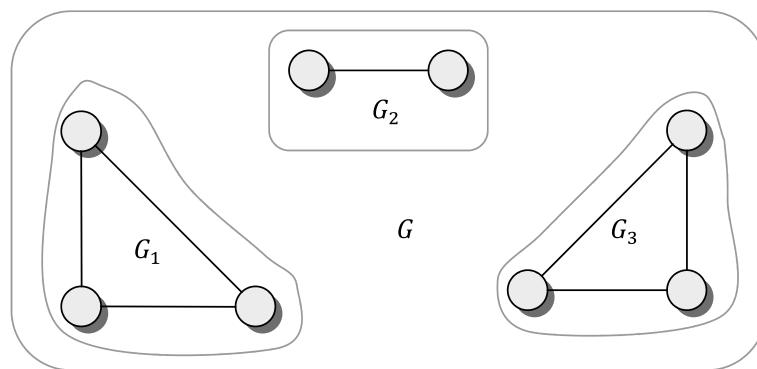
a) Tính liên thông trên đồ thị vô hướng

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).

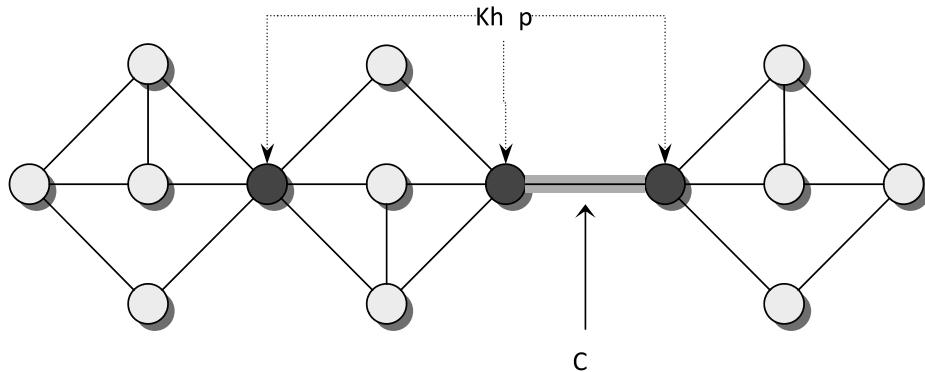
(Ta cũng đồng nhất khái niệm thành phần liên thông U với thành phần liên thông $G_U = (U, E_U)$).



Hình 5-14: Đồ thị và các thành phần liên thông

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 5-14 là ví dụ về 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* (*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*).

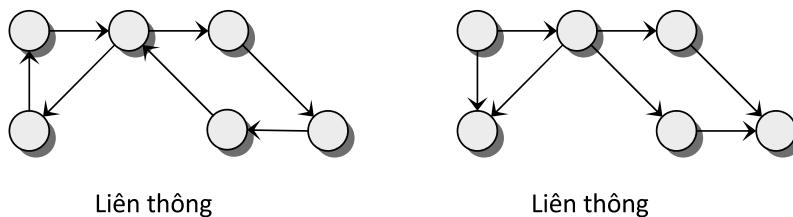


Hình 5-15: Khớp và cầu

b) 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 kì của đồ thị, G gọi là *liên thông yếu* (*weakly connected*) nếu phiên bản vô hướng của nó là đồ thị liên thông.



Hình 5-16: Liên thông mạnh và liên thông yếu

4.2. Bài toán xác định các thành phần liên thông

Một bài toán quan trọng trong lí 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 bất kì, ta liệt kê những đỉnh đến được từ đỉnh đó vào một thành phần liên thông, sau đó loại tất cả các đỉnh đã liệt kê ra

khỏi đồ thị và lặp lại, thuật toán sẽ kết thúc 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:

```

procedure Scan (u∈V)
begin
    «Dùng BFS hoặc DFS liệt kê và đánh dấu những đỉnh có thể
đến được từ u»;
end;
begin
    for ∀u∈V do
        «Khởi tạo v chưa bị đánh dấu»;
    Count := 0;
    for ∀u∈V do
        if «u chưa bị đánh dấu» then
            begin
                Count := Count + 1;
                Output ← «Thông báo thành phần liên thông thứ Count
gồm các đỉnh :»;
                Scan (u);
            end;
    end.

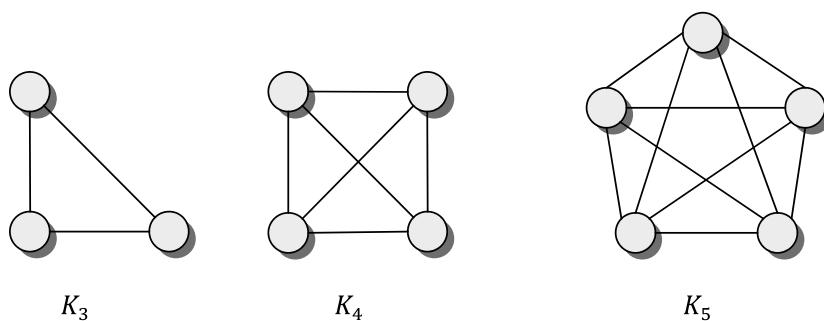
```

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

4.3. Bao đóng của đồ thị vô hướng

a) Định nghĩa

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



Hình 5-17: Đồ thị đầy đủ

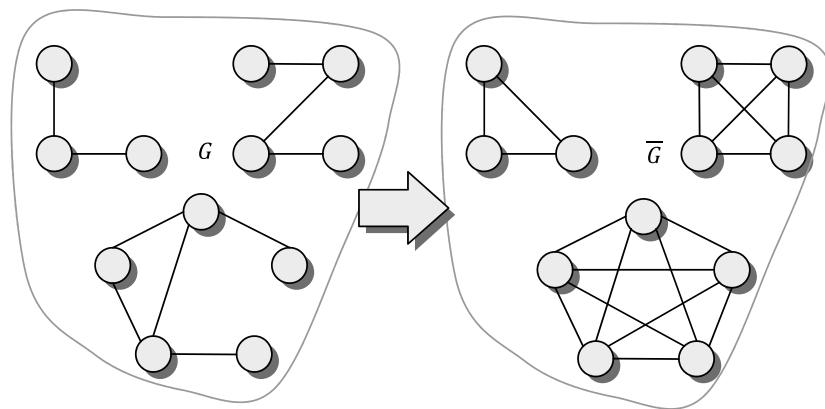
b) Bao đóng đồ thị

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

Giữa hai đỉnh u, v của \bar{G} có cạnh nối \Leftrightarrow Giữa hai đỉnh u, v của G có đường đi
Đồ thị \bar{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 đầy đủ.



Hình 5-18: 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:

c) 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 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 1 tới n , 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 .

Thuật toán Warshall yêu cầu đồ thị phải được biểu diễn bằng ma trận kè $A = \{a_{ij}\}$, trong đó $a_{ij} = \text{True} \Leftrightarrow (i, j) \in E$. Mô hình cài đặt thuật toán khá đơn giản:

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

Việc chứng minh tính đúng đắn của thuật toán đòi hỏi phải lật lại các lí thuyết về bao đóng bắc cầu và quan hệ liên thông, ta sẽ không trình bày ở đây. 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 kè A biểu diễn đồ thị, quy ước rằng $a_{ij} = \text{True}, \forall i$
- Dùng thuật toán Warshall tìm bao đóng, khi đó A là ma trận kè của đồ thị bao đóng
- Dựa vào ma trận kè A , đỉnh 1 và những đỉnh kè với nó sẽ thuộc thành phần liên thông thứ nhất; với đỉnh u nào đó không kè với đỉnh 1, thì u cùng với những đỉnh kè nó sẽ thuộc thành phần liên thông thứ hai; với đỉnh v nào đó không kè với cả đỉnh 1 và đỉnh u , thì v cùng với những đỉnh kè nó sẽ thuộc thành phần liên thông thứ ba v.v...

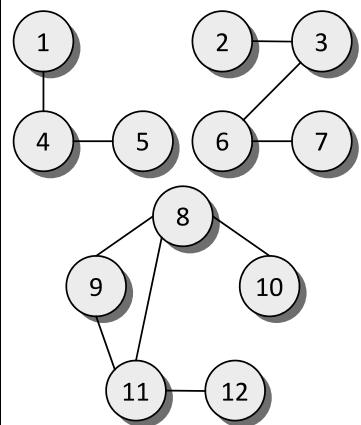
Input

- 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 1 4 2 3 3 6 4 5 6 7 8 9 8 10 9 11 11 8 11 12	Connected Component 1: 1, 4, 5, Connected Component 2: 2, 3, 6, 7, Connected Component 3: 8, 9, 10, 11, 12,



WARSHALL.PAS ✓ Thuật toán Warshall liệt kê các thành phần liên thông

```

{$MODE OBJFPC}
program WarshallAlgorithm;
const
  maxN = 200;
var
  a: array[1..maxN, 1..maxN] of Boolean; //Ma trận kè của đồ thị
  n: Integer;
procedure Enter; //Nhập đồ thị
var
  i, j, k, m: Integer;
begin
  ReadLn(n, m);
  for i := 1 to n do
    begin
      FillChar(a[i][1], n * SizeOf(a[i][1]), False);
      a[i, i] := True;
    end;
  for k := 1 to m do
    begin
      ReadLn(i, j);
      a[i, j] := True;
      a[j, i] := True; //Đồ thị vô hướng: (i, j) = (j, i)
    end;
end;
procedure ComputeTransitiveClosure; //Thuật toán Warshall
var

```

```

k, i, j: Integer;
begin
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        a[i, j] := a[i, j] or a[i, k] and a[k, j];
end;
procedure PrintResult;
var
  Count: Integer;
  avail: array[1..maxN] of Boolean; //avail[v] = True  $\leftrightarrow$  v chưa được liệt kê
  vào thành phần liên thông nào
  u, v: Integer;
begin
  FillChar(avail, n * SizeOf(Boolean), True); //Mọi đỉnh đều chưa
  được liệt kê vào thành phần liên thông nào
  Count := 0;
  for u := 1 to n do
    if avail[u] then //Với một đỉnh u chưa được liệt kê vào thành phần liên thông
    nào
      begin //Liệt kê thành phần liên thông chứa u
        Inc(Count);
        Write('Connected Component ', Count, ': ');
        for v := 1 to n do
          if a[u, v] then //Xét những đỉnh v kề u (trên bao đóng)
            begin
              Write(v, ', ');
              avail[v] := False; //Liệt kê đỉnh đó vào thành phần liên thông
              chứa u
            end;
        WriteLn;
      end;
    end;
  begin
    Enter;
    ComputeTransitiveClosure;
    PrintResult;
  end.

```

4.4. 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:

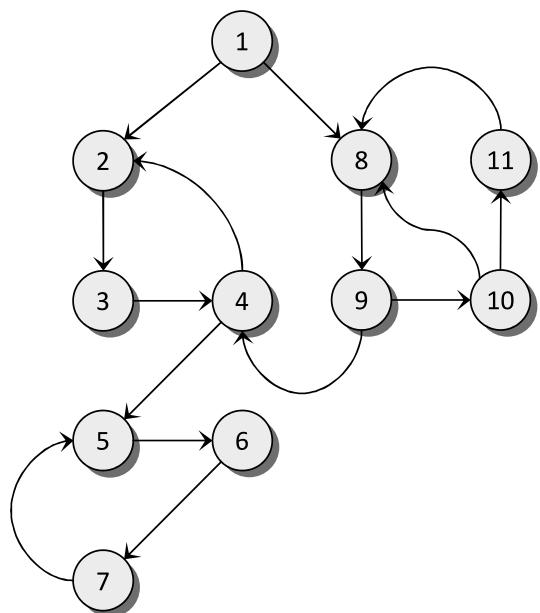
Input

- Dòng đầu: Chứa số đỉnh $n \leq 10^5$ 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 1:
1 2	7, 6, 5,
1 8	Strongly Connected Component 2:
2 3	4, 3, 2,
3 4	Strongly Connected Component 3:
4 2	11, 10, 9, 8,
4 5	Strongly Connected Component 4:
5 6	1,
6 7	
7 5	
8 9	
9 4	
9 10	
10 8	
10 11	
11 8	



a) Phân tích

Xét thuật toán tìm kiếm theo chiều sâu:

```

procedure DFSVisit (u∈V);
begin
    «Thêm u vào cây T»
    for ∀v∈V: (u, v)∈E do
        if v∉T then
            begin

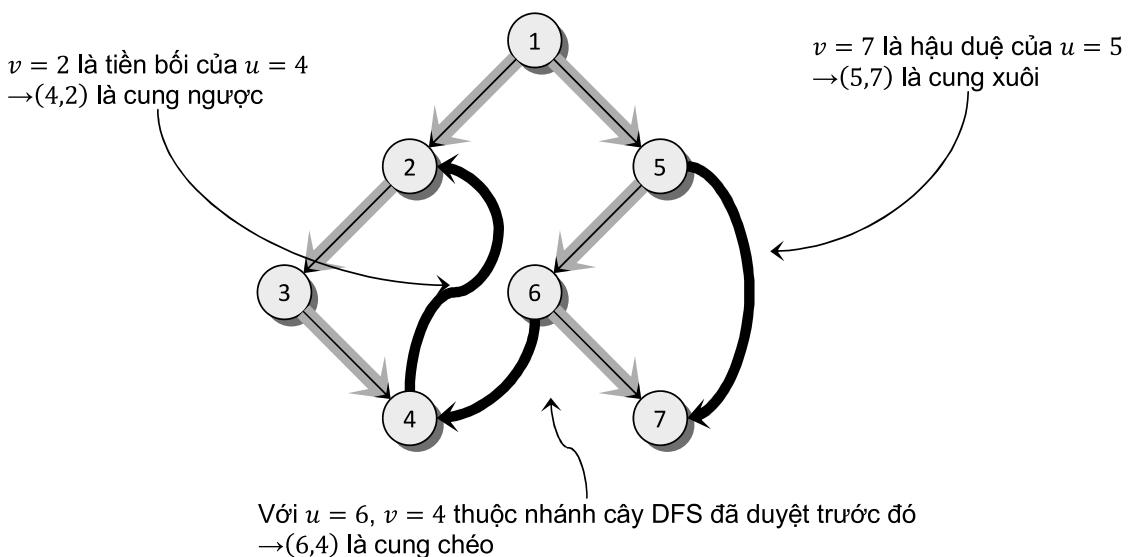
```

```

    «Thêm v và cung (u, v) vào cây T»;
    DFSVisit(v);
end;
begin
Input → Đồ thị G;
for ∀v∈V do avail[v] := True;
for ∀v∈V do
if avail[v] then
begin
«Tạo ra một cây rỗng, gọi là T»
DFSVisit(v);
end;
end.

```

Để ý thủ tục thăm đỉnh đệ quy $DFSVisit(u)$. Thủ tục này xét tất cả những đỉnh v nối từ u :



Hình 5-19: Ba dạng cung ngoài cây DFS

- 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 thủ tục $DFSVisit(u)$ do dây chuyền đệ quy từ thủ tục $DFSVisit(v)$ 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 thủ tục $DFSVisit(u)$ sau khi tiến đệ quy theo một hướng khác đã gọi $DFSVisit(v)$ rồi. Nên khi dây chuyền đệ quy lùi lại về thủ tục $DFSVisit(u)$ 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)*

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 (Hình 5-19).

b) *Cây DFS và các thành phần liên thông mạnh*

Định lí 5-9

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

Vì x và y là hai đỉnh thuộc C nên có một đường đi từ x tới y và một đường đi khác từ y tới 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.

Định lí 5-10

Với một thành phần liên thông mạnh C bất kì, 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 kì 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 Định lí 5-9, tất cả các đỉnh x_1, x_2, \dots, x_k đều thuộc C , lại do cách chọn r nên chúng sẽ phải thăm sau đỉnh r . Lại từ Định lí 5-8 (định lí 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 lí – đỉnh thăm trước tất cả các đỉnh khác trong C – gọi là *chốt* của thành phần liên thông mạnh C . Mỗi thành phần liên thông mạnh có duy nhất một chốt. Xét về vị trí trong cây DFS, chốt của một thành phần liên

thông mạnh 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.

Định lí 5-11

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

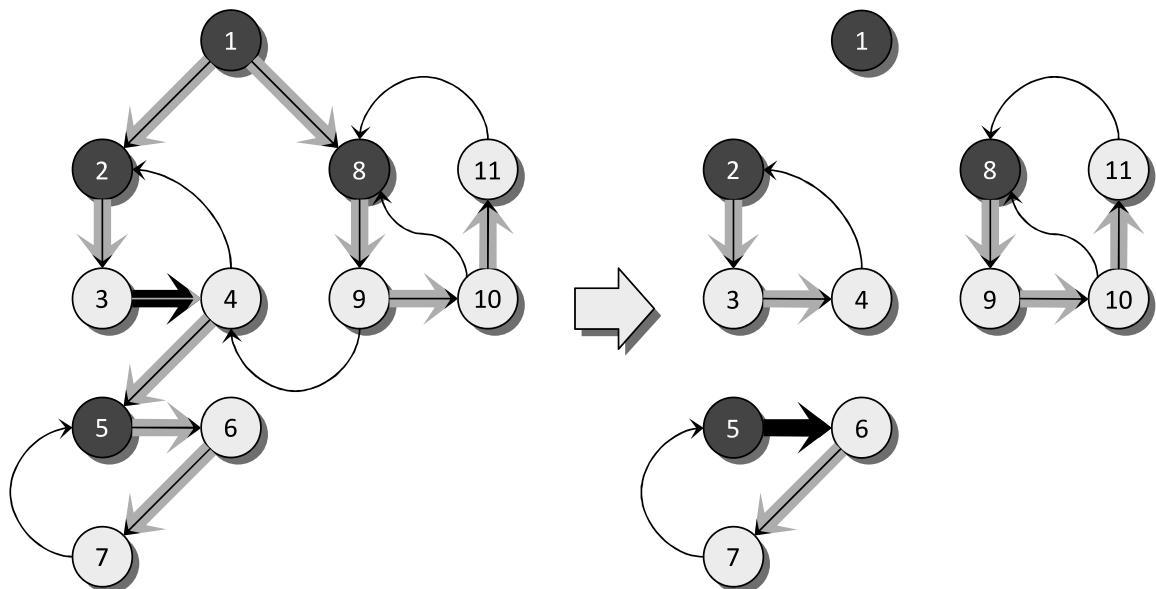
Chứng minh

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 Định lí 5-10, 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 kì chốt nào khác nên r phải là hậu duệ của s . Ta có đường đi $s \rightarrow r \rightarrow v$, mà s và v thuộc cùng một thành phần liên thông mạnh nên theo Định lí 5-9, 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 $r = s$.

Theo Định lí 5-10, ta đã có 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 .

c) Thuật toán Tarjan

Ý tưởng



Hình 5-20: Thuật toán Tarjan “bé” cây DFS

Thuật toán Tarjan [40] 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 tại vị trí các chốt để được các nhánh rời rạc, mỗi nhánh là một thành phần liên thông mạnh.

Mô hình cài đặt của thuật toán Tarjan:

```

procedure DFSVisit ( $u \in V$ ) ;
begin
    «Đánh dấu  $u$  đã thăm»
    for  $\forall v \in V : (u, v) \in E$  do
        if « $v$  chưa thăm» then DFSVisit ( $v$ );
        if « $u$  là chốt» then
            begin
                «Liệt kê thành phần liên thông mạnh tương ứng với chốt  $u$ »
                «Loại bỏ các đỉnh đã liệt kê khỏi đồ thị và cây DFS»
            end;
    end;
    begin
        «Đánh dấu mọi đỉnh đều chưa thăm»
        for  $\forall v \in V$  do
            if « $v$  chưa thăm» then DFSVisit ( $v$ );
    end.

```

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 lí 5-12

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 trong nhánh DFS gốc r không có cung tới 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 thăm trước r thì cung đó phải là cung ngược hoặc cung chéo.

Nếu cung (u, v) là cung ngược thì v là tiền bối của u , mà r cũng là tiền bối của u nhưng 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.

Nếu cung (u, v) là cung chéo, ta gọi s là chốt của thành phần liên thông chứa v . Tại thời điểm thủ tục $DFSVisit(u)$ xét tới cung (u, v) , đỉnh r đã được duyệt đến nhưng chưa duyệt xong (do r là tiền bối của u), đỉnh s cũng đã duyệt đến (s được thăm trước v do s là chốt của thành phần liên thông chứa v , v được thăm trước r theo giả thiết, r được thăm trước u vì r là chốt của thành phần liên thông mạnh chứa u) nhưng chưa 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 trong đó có đỉnh v ra khỏi đồ thị nên cung (u, v) sẽ không được tính đến nữa), điều này chỉ ra rằng khi $DFSVisit(u)$ được gọi, hai thủ tục $DFSVisit(r)$ và $DFSVisit(s)$ 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 lí 5-12, 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 hết sức thông minh, nội dung của nó là đánh số thứ tự các đỉnh theo thứ tự duyệt đến. Định nghĩa $Number[u]$ là số thứ tự của đỉnh u theo cách đánh số đó. Ta tính thêm $Low[u]$ là giá trị $Number[.]$ 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 $Low[u]$ như sau:

Trong thủ tục $DFSVisit(u)$, trước hết ta đánh số thứ tự thăm cho đỉnh u : $Number[u]$ và khởi tạo $Low[u] := +\infty$. Sau đó xét các đỉnh v nối từ u , có hai khả năng:

Nếu v đã thăm thì ta cực tiểu hoá $Low[u]$ theo công thức:

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

Nếu v chưa thăm thì ta gọi đệ quy $DFSVisit(v)$, 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 thủ tục $DFSVisit(u)$), ta so sánh $Low[u]$ và $Number[u]$, nếu như $Low[u] \geq Number[u]$ 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 định nghĩa một danh sách *Stack* được tổ chức dưới dạng ngăn xếp và dùng ngăn xếp này để lấy ra các đỉnh thuộc một nhánh nào đó. Khi duyệt 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 *Stack* ngay sau u . Nếu u là chốt, ta chỉ việc lấy các đỉnh ra khỏi ngăn xếp *Stack* 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 đầy đủ của thuật toán Tarjan

```

procedure DFSVisit (u $\in$ V) ;
begin
    Count := Count + 1;
    Number[u] := Count; //Đánh số u theo thứ tự duyệt đến
    Low[u] := + $\infty$ ;
    Push(u); //Đẩy u vào ngăn xếp
    for  $\forall v \in V : (u, v) \in E$  do
        if Number[v] > 0 then //v đã thăm
            Low[u] := min(Low[u], Number[v])
        else //v chưa thăm
            begin
                DFSVisit(v); //Đi thăm v
                Low[u] := min(Low[u], Low[v]);
            end;
    //Đến đây u được duyệt xong
    if Low[u]  $\geq$  Number[u] then //Nếu u là chốt
        begin
            «Thông báo thành phần liên thông mạnh với chốt u gồm có
            các đỉnh:»;
            repeat
                v := Pop; //Lấy từ ngăn xếp ra một đỉnh v
                Output  $\leftarrow$  v;
                «Xoá đỉnh v khỏi đồ thị: v := V - {v}»;
            until v = u;
        end;
    end;
    begin
        Count := 0;
        Stack :=  $\emptyset$ ; //Khởi tạo một ngăn xếp rỗng
        for  $\forall v \in V$  do Number[v] := 0; //Number[v] = 0  $\leftrightarrow$  v chưa thăm
    end;
end;

```

```

for ∀v∈V do
    if Number[v] = 0 then DFSVisit(v);
end.

```

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

Cài đặt

Chương trình cài đặt dưới đây biểu diễn đồ thị bởi danh sách liên thuộc kiểu forward star: Mỗi đỉnh u sẽ được cho tương ứng với một danh sách các cung đi ra khỏi u , như vậy mỗi cung sẽ xuất hiện trong đúng một danh sách liên thuộc. Nếu các cung được lưu trữ trong mảng $e[1 \dots m]$, danh sách liên thuộc được xây dựng bằng hai mảng.

- $head[u]$ là chỉ số cung đầu tiên trong danh sách liên thuộc của đỉnh u . Nếu danh sách liên thuộc đỉnh u là \emptyset , $head[u]$ được gán bằng 0.
- $link[i]$ là chỉ số cung kế tiếp cung $e[i]$ trong danh sách liên thuộc chứa cung $e[i]$. Trường hợp $e[i]$ là cung cuối cùng của một danh sách liên thuộc, $link[i]$ được gán bằng 0

TARJAN.PAS ✓ Thuật toán Tarjan

```

{$MODE OBJFPC}
{$M 4000000}
program StronglyConnectedComponents;
const
  maxN = 100000;
  maxM = 1000000;
type
  TStack = record
    Items: array[1..maxN] of Integer;
    Top: Integer;
  end;
  TEdge = record //Cấu trúc cung
    x, y: Integer; //Hai đỉnh đầu mút
  end;
var
  e: array[1..maxM] of TEdge; //Danh sách cạnh
  link: array[1..maxM] of Integer; //link[i]: chỉ số cung tiếp theo e[i] trong
  danh sách liên thuộc

```

```

head: array[1..maxN] of Integer; //head[u]: chỉ số cung đầu tiên trong
danh sách liên thuộc các cung đi ra khỏi u
avail: array[1..maxN] of Boolean;
Number, Low: array[1..maxN] of Integer;
Stack: TStack;
n, Count, SCC: Integer;
procedure Enter; //Nhập dữ liệu
var
  i, u, v, m: Integer;
begin
  ReadLn(n, m);
  for i := 1 to m do //Đọc danh sách cạnh
    with e[i] do ReadLn(x, y);
  FillChar(head[1], n * SizeOf(head[1]), 0); //Khởi tạo các danh sách
liên thuộc rỗng
  for i := m downto 1 do //Xây dựng các danh sách liên thuộc
    with e[i] do
      begin
        link[i] := head[x]; //Mởc nối e[i] = (x, y) vào danh sách liên thuộc
những cung đi ra khỏi x
        head[x] := i;
      end;
  end;
  procedure Init; //Khởi tạo
begin
  FillChar(Number, n * SizeOf(Number[1]), 0); //Mọi đỉnh đều chưa
thăm
  FillChar(avail, n * SizeOf(avail[1]), True); //Chưa đỉnh nào bị
loại
  Stack.Top := 0; //Ngăn xếp rỗng
  Count := 0; //Biến đếm số thứ tự duyệt đến, dùng để đánh số
  SCC := 0; //Biến đánh số các thành phần liên thông
end;
procedure Push(v: Integer); //Đẩy một đỉnh v vào ngăn xếp
begin
  with Stack do
    begin
      Inc(Top); Items[Top] := v;
    end;
end;
function Pop: Integer; //Lấy một đỉnh v khỏi ngăn xếp, trả về trong kết quả
hàm
begin

```

```

with Stack do
begin
    Result := Items[Top]; Dec(Top);
end;
end;
//Hàm cực tiểu hoá: Target := Min(Target, Value)
procedure Minimize(var Target: Integer; Value: Integer);
begin
    if Value < Target then Target := Value;
end;
procedure DFSVisit(u: Integer); //Thuật toán tìm kiếm theo chiều sâu bắt đầu từ u
var
    i, v: Integer;
begin
    Inc(Count); Number[u] := Count; //Trước hết đánh số cho u
    Low[u] := maxN + 1; //khởi tạo Low[u]:=+∞ rồi sau cực tiểu hoá dần
    Push(u); //Đẩy u vào ngăn xếp
    i := head[u]; //Duyệt từ đầu danh sách liên thuộc các cung đi ra khỏi u
    while i <> 0 do
        begin
            v := e[i].y; //Xét những đỉnh v nối từ u
            if avail[v] then //Nếu v chưa bị loại
                if Number[v] <> 0 then //Nếu v đã thăm
                    Minimize(Low[u], Number[v]) //cực tiểu hoá Low[u] theo công thức này
                else //Nếu v chưa thăm
                    begin
                        DFSVisit(v); //Tiếp tục tìm kiếm theo chiều sâu bắt đầu từ v
                        Minimize(Low[u], Low[v]); //Rồi cực tiểu hoá Low[u] theo công thức này
                    end;
                i := link[i]; //Chuyển sang xét cung tiếp theo trong danh sách liên thuộc
            end;
        //Đến đây thì đỉnh u được duyệt xong, tức là các đỉnh thuộc nhánh DFS gốc u đều đã thăm
        if Low[u] >= Number[u] then //Nếu u là chốt
            begin //Liệt kê thành phần liên thông mạnh có chốt u
                Inc(SCC);
                WriteLn('Strongly Connected Component ', SCC, ': ');
                repeat
                    v := Pop; //Lấy dần các đỉnh ra khỏi ngăn xếp
                    Write(v, ', ', ' ');
                    //Liệt kê các đỉnh đó
                    avail[v] := False; //Rồi loại luôn khỏi đồ thị
                until Low[u] < Number[u];
            end;
        end;
    end;

```

```

        until v = u; //Cho tới khi lấy tới đỉnh u
        Writeln;
    end;
end;
procedure Tarjan; //Thuật toán Tarjan
var
    v: Integer;
begin
    for v := 1 to n do
        if avail[v] then DFSVisit(v);
end;
begin
    Enter;
    Init;
    Tarjan;
end.

```

d) Thuật toán Kosaraju-Sharir

Mô hình

Có một thuật toán khác để liệt kê các thành phần liên thông mạnh là thuật toán Kosaraju-Sharir (1981). Thuật toán này thực hiện qua hai bước:

- Bước 1: Dùng thuật toán tìm kiếm theo chiều sâu với thủ tục *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.

Định lí 5-13

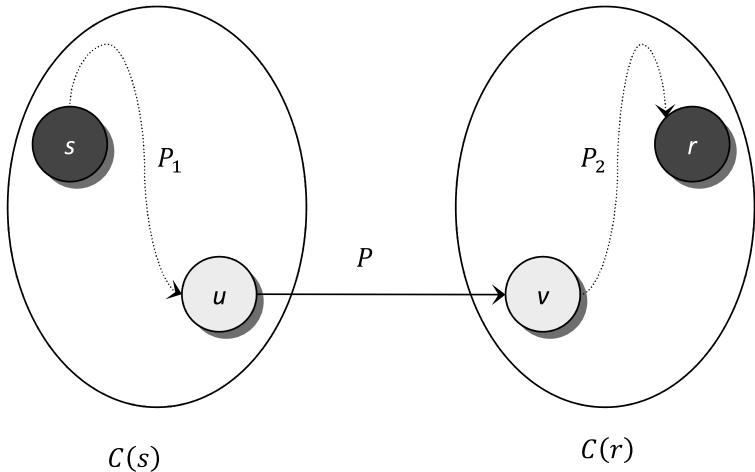
Với r là đỉnh duyệt xong sau cùng thì r là chốt của một thành phần liên thông mạnh không có cung đi vào.

Chứng minh

Dễ thấy rằng đỉnh r duyệt xong sau cùng phải là gốc của một cây DFS nên r sẽ là chốt của một thành phần liên thông mạnh, kí 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 $C(r)$. Do tính chất của chốt, s được thăm trước mọi đỉnh khác trên đường P_1 và r được thăm trước mọi đỉnh khác trên đường P_2 . 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 5-21)



Hình 5-21

Có hai khả năng xảy ra:

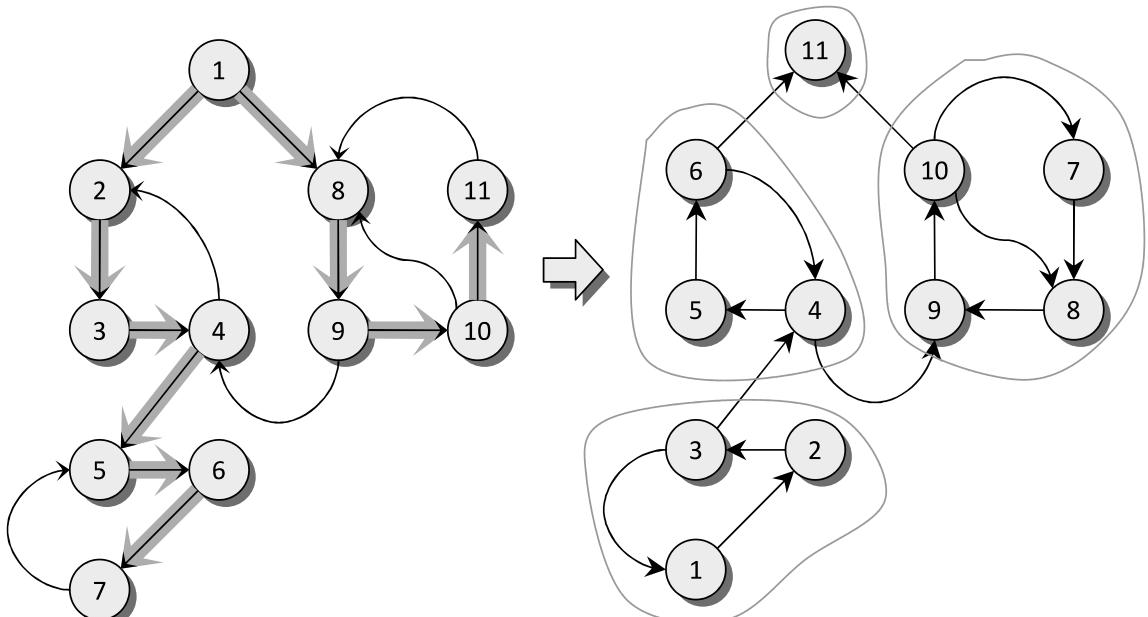
- Nếu s được thăm trước r thì vào thời điểm s được thăm, mọi đỉnh khác trên đường đi P chưa thăm. Theo Định lí 5-8 (định lí đường đi trắng), s sẽ là tiền bối của r và phải được duyệt xong sau r . Trái với giả thiết r là đỉnh duyệt xong sau cùng.
- Nếu s được thăm sau r , nghĩa là vào thời điểm r được duyệt đến thì s chưa duyệt đến, lại do r được duyệt xong sau cùng nên vào thời điểm r duyệt xong thì s đã duyệt xong. Theo Định lí 5-13, 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.

Định lí được chứng minh.

Định lí 5-13 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 5-22)

Ví dụ:



Hình 5-22. Đánh số lại, đảo chiều các cung và thực hiện thuật toán tìm kiếm trên đồ thị với cách chọn các đỉnh xuất phát ngược lại với thứ tự duyệt xong (thứ tự 11, 10... 3, 2, 1)

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 (cơ chế tương tự như ngăn xếp)

Để liệt kê các thành phần liên thông mạnh của đơn đồ thị có hướng bằng thuật toán Tarjan cũng như thuật toán Kosaraju-Sharir, cách biểu diễn đồ thị tốt nhất là sử dụng danh sách kề hoặc danh sách liên thuộc. Tuy nhiên với thuật toán Kosaraju-Sharir, việc cài đặt bằng dach sách liên thuộc là hợp lí hơn bởi nó cho phép chuyển từ cách biểu diễn forward star sang cách biểu diễn reverse star một cách dễ dàng bằng cách chỉnh lại mảng *link* và *head*. Cấu trúc forward star được sử dụng ở pha đánh số lại các đỉnh, còn cấu trúc reverse star được sử dụng khi liệt kê các thành phần liên thông mạnh (bởi cần thực hiện trên đồ thị đảo chiều)



KOSARAJUSHARIR.PAS ✓ Thuật toán Kosaraju–Sharir

```
{ $MODE OBJFPC }
{$M 4000000}
program StronglyConnectedComponents;
const
  maxN = 100000;
  maxM = 1000000;
```

```

type
  TEdge = record //Cấu trúc cung
    x, y: Integer; //Hai đỉnh đầu mút
  end;
var
  e: array[1..maxM] of TEdge; //Danh sách cạnh
  link: array[1..maxM] of Integer; //link[i]: Chỉ số cung kế tiếp e[i] trong
  danh sách liên thuộc
  head: array[1..maxN] of Integer; //head[u]: Chỉ số cung đầu tiên trong
  danh sách liên thuộc
  avail: array[1..maxN] of Boolean;
  List: array[1..maxN] of Integer;
  Top: Integer;
  n, m, v, SCC: Integer;
procedure Enter; //Nhập dữ liệu
var
  i, u, v: Integer;
begin
  ReadLn(n, m);
  for i := 1 to m do
    with e[i] do
      ReadLn(x, y);
end;
procedure Numbering; //Liệt kê các đỉnh theo thứ tự duyệt xong vào danh sách List
var
  i, u: Integer;
procedure DFSVisit(u: Integer); //Thuật toán DFS từ u
var
  i, v: Integer;
begin
  avail[u] := False;
  i := head[u];
  while i <> 0 do //Xét các cung e[i] đi ra khỏi u
    begin
      v := e[i].y;
      if avail[v] then DFSVisit(v);
      i := link[i];
    end;
  Inc(Top); List[Top] := u; //u duyệt xong, đưa u vào cuối danh sách List
end;

```

```

begin
    //Xây dựng danh sách liên thuộc dạng forward star: Mỗi đỉnh u tương ứng với danh sách
    //các cung đi ra khỏi u
    FillChar(head[1], n * SizeOf(head[1]), 0);
    for i := m downto 1 do
        with e[i] do
            begin
                link[i] := head[x];
                head[x] := i;
            end;
    FillChar(avail[1], n * SizeOf(avail[1]), True);
    Top := 0; //Khởi tạo danh sách List rỗng
    for u := 1 to n do
        if avail[u] then DFSVisit(u);
    end;
procedure KosarajuSharir;
var
    i, u: Integer;
procedure Enum(u: Integer); //Thuật toán DFS từ u trên đồ thị đảo chiều
var
    i, v: Integer;
begin
    avail[u] := False;
    Write(u, ', ');
    i := head[u];
    while i <> 0 do //Xét các cung e[i] đi vào u
        begin
            v := e[i].x;
            if avail[v] then Enum(v);
            i := link[i];
        end;
    end;
begin
    //Xây dựng danh sách liên thuộc dạng reverse star: mỗi đỉnh u tương ứng với danh sách
    //các cung đi vào u
    FillChar(head[1], n * SizeOf(head[1]), 0);
    for i := m downto 1 do
        with e[i] do
            begin
                link[i] := head[y];
                head[y] := i;
            end;

```

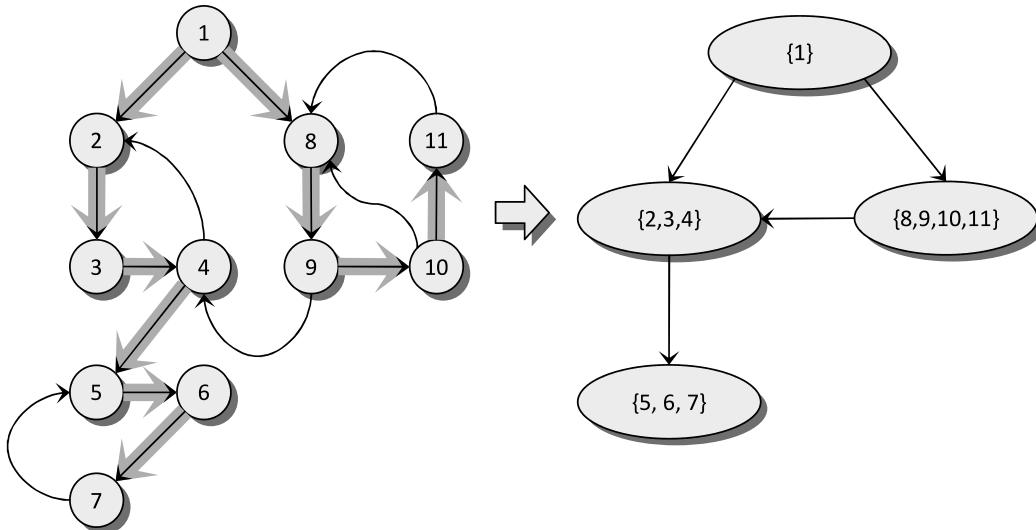
```

FillChar(avail[1], n * SizeOf(avail[1]), True);
SCC := 0;
for u := n downto 1 do
  if avail[List[u]] then //Liệt kê thành phần liên thông chót List[u]
    begin
      Inc(SCC);
      WriteLn('Strongly Connected Component ', SCC, ': ');
      Enum(List[u]);
      WriteLn;
    end;
  end;
begin
  Enter;
  Numbering;
  KosarajuSharir;
end.

```

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.

4.5. Sắp xếp tô pô



Hình 5-23. Đồ thị có hướng và đồ thị các thành phần liên thông mạnh

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

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 trên đồ thị G , mâu thuẫn với tính tối đại của một thành phần liên thông mạnh.

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} từ 1 trở đi 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 tố 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

- 5.14. 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.
- 5.15. 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).
- 5.16. Trên mặt phẳng với hệ toạ 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 toạ 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 kì trong một nhóm bất kì có thể đi được sang nhau sau một số hữu hạn các bước di chuyển giữa hai đường tròn thông nhau.
- 5.17. 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 kì thuộc thuộc hai nhóm khác nhau thì không giống nhau:

1	1	1	0	1	1	0	0	1
1	0	0	0	1	0	0	1	1
1	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1
1	0	1	0	0	1	1	0	1
1	1	1	1	1	0	0	1	1

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

- 5.18. 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 .
- 5.19. 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 .
- 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)
 - Tìm thuật toán và viết chương trình kiểm tra tính nửa liên thông của đồ thị.

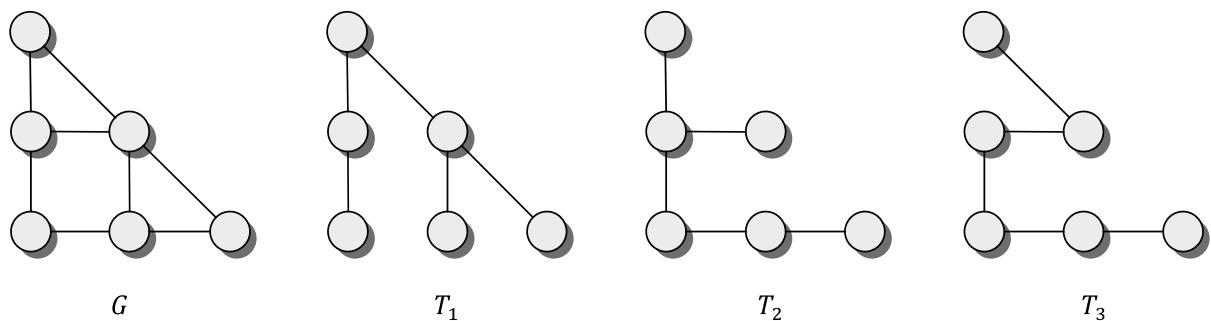
5. Vài ứng dụng của DFS và BFS

5.1. Xây dựng cây khung của đồ thị

Cây là đồ thị vô hướng, liên thông, không có chu trình đơn. Đồ thị vô hướng không có chu trình đơn gọi là rừng (hợp của nhiều cây). Như vậy mỗi thành phần liên thông của rừng là một cây.

Xét đồ thị $G = (V, E)$ và $T = (V, E_T)$ là một đồ thị con của đồ thị G ($E_T \subseteq E$), nếu T là một cây thì ta gọi T là *cây khung* hay *cây bao trùm* (*spanning tree*) của đồ thị G . Điều kiện cần và đủ để một đồ thị vô hướng có cây khung là đồ thị đó phải liên thông.

Dễ thấy rằng với một đồ thị vô hướng liên thông có thể có nhiều cây khung (Hình 5-24).



Hình 5-24: Đồ thị và một số ví dụ cây khung

Định lí 5-14 (Daisy Chain Theorem)

Giả sử $T = (V, E)$ là đồ thị vô hướng với n đỉnh. Khi đó các mệnh đề sau là tương đương:

1. T là cây
2. T không chứa chu trình đơn và có $n - 1$ cạnh
3. T liên thông và mỗi cạnh của nó đều là cầu
4. Giữa hai đỉnh bất kỳ của T đều tồn tại đúng một đường đi đơn
5. T không chứa chu trình đơn nhưng nếu thêm vào một cạnh ta thu được một chu trình đơn.
6. T liên thông và có $n - 1$ cạnh

Chứng minh:

1⇒2:

Từ T là cây, theo định nghĩa T không chứa chu trình đơn. Ta sẽ chứng minh cây T có n đỉnh thì phải có $n - 1$ cạnh bằng quy nạp theo số đỉnh n . Rõ ràng khi $n = 1$ thì cây có 1 đỉnh sẽ chứa 0 cạnh. Nếu $n = 1$, gọi $P = \langle v_1, v_2, \dots, v_k \rangle$ là đường đi dài nhất (qua nhiều cạnh nhất) trong T . Đỉnh v_1 không thể kề với đỉnh nào trong số các đỉnh v_3, v_4, \dots, v_k , bởi nếu có cạnh (v_1, v_p) ($3 \leq p \leq k$), ta sẽ thiết lập được chu trình đơn $\langle v_1, v_2, \dots, v_p, v_1 \rangle$. Mặt khác, đỉnh v_1 cũng không thể kề với đỉnh nào khác ngoài các đỉnh trên đường đi P trên bởi nếu có cạnh $(v_0, v_1) \in E$, $v_0 \notin P$ thì ta thiết lập được đường đi $\langle v_0, v_1, v_2, \dots, v_k \rangle$ dài hơn P . Vậy đỉnh v_1 chỉ có đúng một cạnh nối với v_2 , nói cách khác, v_1 là đỉnh treo. Loại bỏ v_1 và cạnh (v_1, v_2) khỏi T , ta được đồ thị mới cũng là cây và có $n - 1$ đỉnh, cây này theo giả thiết quy nạp có $n - 2$ cạnh. Vậy cây T có $n - 1$ cạnh.

2⇒3:

Giả sử T có k thành phần liên thông T_1, T_2, \dots, T_k . Vì T không chứa chu trình đơn nên các thành phần liên thông của T cũng không chứa chu trình đơn, tức là các T_1, T_2, \dots, T_k đều là cây. Gọi n_1, n_2, \dots, n_k lần lượt là số đỉnh của T_1, T_2, \dots, T_k thì cây T_1 có $n_1 - 1$ cạnh, cây T_2 có $n_2 - 1$ cạnh..., cây T_k có $n_k - 1$ cạnh. Cộng lại ta có số cạnh của T là $n - k$ cạnh. Theo giả thiết, cây T có $n - 1$ cạnh, suy ra $k = 1$, đồ thị chỉ có một thành phần liên thông là đồ thị liên thông.

Bây giờ khi T đã liên thông, kết hợp với giả thiết T không có chu trình nên nếu bỏ đi một cạnh bất kì thì đồ thị mới vẫn không chứa chu trình. Đồ thị mới này không thể liên thông vì nếu không nó sẽ phải là một cây và theo chứng minh trên, đồ thị mới sẽ có $n - 1$ cạnh, tức là T có n cạnh. Mâu thuẫn này chứng tỏ tất cả các cạnh của T đều là cầu.

3⇒4:

Gọi x và y là 2 đỉnh bất kì trong T , vì T liên thông nên sẽ có một đường đi đơn từ x tới y . Nếu tồn tại một đường đi đơn khác từ x tới y thì nếu ta bỏ đi một cạnh (u, v) nằm trên đường đi thứ nhất nhưng không nằm trên đường đi thứ hai thì từ u vẫn có thể đến được v bằng cách: đi từ u đi theo chiều tới x theo các cạnh thuộc đường thứ nhất, sau đó đi từ x tới y theo đường thứ hai, rồi lại đi từ y tới v theo các cạnh thuộc đường đi thứ nhất. Điều này chỉ ra việc bỏ đi cạnh (u, v) không ảnh hưởng tới việc có thể đi lại được giữa hai đỉnh bất kì. Mâu thuẫn với giả thiết (u, v) là cầu.

4⇒5:

Thứ nhất T không chứa chu trình đơn vì nếu T chứa chu trình đơn thì chu trình đó qua ít nhất hai đỉnh (u, v) . Rõ ràng đọc theo các cạnh trên chu trình đó thì từ u có hai đường đi đơn tới v . Vô lí.

Giữa hai đỉnh (u, v) bất kì của T có một đường đi đơn nối u với v , vậy khi thêm cạnh (u, v) vào đường đi này thì sẽ tạo thành chu trình.

5⇒6:

Gọi u và v là hai đỉnh bất kì trong T , thêm vào T một cạnh (u, v) nữa thì theo giả thiết sẽ tạo thành một chu trình chứa cạnh (u, v) . Loại bỏ cạnh này đi thì phần còn lại của chu trình sẽ là một đường đi từ u tới v . Mọi cặp đỉnh của T đều có một đường đi nối chúng tức là T liên thông, theo giả thiết T không chứa chu trình đơn nên T là cây và có $n - 1$ cạnh.

6⇒1:

Giả sử T không là cây thì T có chu trình, huỷ bỏ một cạnh trên chu trình này thì T vẫn liên thông, nếu đồ thị mới nhận được vẫn có chu trình thì lại huỷ một cạnh trong chu trình mới. Cứ như thế cho tới khi ta nhận được một đồ thị liên thông không có chu trình. Đồ thị này là cây nhưng lại có $< n - 1$ cạnh (vô lí). Vậy T là cây.

Định lí 5-15

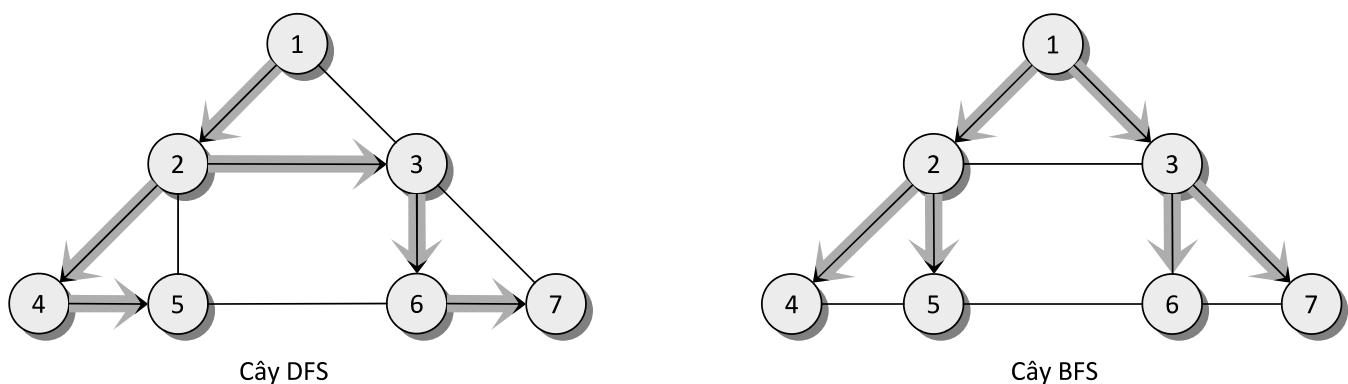
Số cây khung của đồ thị đầy đủ K_n là n^{n-2} .

Ta sẽ khảo sát hai thuật toán tìm cây khung trên đồ thị vô hướng liên thông $G = (V, E)$.

a) Xây dựng cây khung bằng thuật toán hợp nhất

Trước hết, đặt $T = (V, \emptyset)$; T không chứa cạnh nào thì có thể coi T gồm $|V|$ cây rời rạc, mỗi cây chỉ có 1 đỉnh. Sau đó xét lần lượt các cạnh của G , nếu cạnh đang xét nối hai cây khác nhau trong T thì thêm cạnh đó vào T , đồng thời hợp nhất hai cây đó lại thành một cây. Cứ làm như vậy cho tới khi kết nạp đủ $|V| - 1$ cạnh vào T thì ta được T là cây khung của đồ thị. Trong việc xây dựng cây khung bằng thuật toán hợp nhất, một cấu trúc dữ liệu biểu diễn các tập rời nhau thường được sử dụng để tăng tốc phép hợp nhất hai cây cũng như phép kiểm tra hai đỉnh có thuộc hai cây khác nhau không.

b) Xây dựng cây khung bằng các thuật toán tìm kiếm trên đồ thị.



Hình 5-25: Cây khung DFS và cây khung BFS trên cùng một đồ thị (mũi tên chỉ chiều đi thăm các đỉnh)

Áp dụng thuật toán BFS hay DFS bắt đầu từ đỉnh s nào đó, tại mỗi bước từ đỉnh u tới thăm đỉnh v , ta thêm vào thao tác ghi nhận luôn cạnh (u, v) vào cây khung. Do đồ thị liên thông nên thuật toán sẽ xuất phát từ s và tới thăm tất cả các đỉnh còn lại, mỗi đỉnh đúng một lần, tức là quá trình duyệt sẽ ghi nhận được đúng $|V| - 1$ cạnh. Tất cả những cạnh đó không tạo thành chu trình đơn bởi thuật toán không

thăm lại những đỉnh đã thăm. Theo mệnh đề tương đương thứ hai, ta có những cạnh ghi nhận được tạo thành một cây khung của đồ thị.

5.2. Tập các chu trình cơ sở của đồ thị

Xét một đồ thị vô hướng liên thông $G = (V, E)$; gọi $T = (V, E_T)$ là một cây khung của nó. Các cạnh của cây khung được gọi là các cạnh trong, còn các cạnh khác là các cạnh ngoài cây.

Nếu thêm một cạnh ngoài $e \in E - E_T$ vào cây khung T , thì ta được đúng một chu trình đơn trong T , kí hiệu chu trình này là C_e . Chu trình C_e chỉ chứa duy nhất một cạnh ngoài cây còn các cạnh còn lại đều là cạnh trong cây T

Tập các chu trình:

$$\Psi = \{C_e | e \in E - E_T\}$$

được gọi là tập các chu trình cơ sở của đồ thị G .

Các tính chất quan trọng của tập các chu trình cơ sở:

- Tập các chu trình cơ sở là phụ thuộc vào cây khung, hai cây khung khác nhau có thể cho hai tập chu trình cơ sở khác nhau.
- Cây khung của đồ thị liên thông $G = (V, E)$ luôn chứa $|V| - 1$ cạnh, còn lại $|E| - |V| + 1$ cạnh ngoài. Tương ứng với mỗi cạnh ngoài có một chu trình cơ sở, vậy số chu trình cơ sở của đồ thị liên thông là $|E| - |V| + 1$.
- Tập các chu trình cơ sở là tập nhiều nhất các chu trình thoả mãn: Mỗi chu trình có đúng một cạnh riêng, cạnh đó không nằm trong bất cứ một chu trình nào khác. Điều này có thể chứng minh được bằng cách lấy trong đồ thị liên thông một tập gồm k chu trình thoả mãn điều đó thì việc loại bỏ cạnh riêng của một chu trình sẽ không làm mất tính liên thông của đồ thị, đồng thời không ảnh hưởng tới sự tồn tại của các chu trình khác. Như vậy nếu loại bỏ tất cả các cạnh riêng thì đồ thị vẫn liên thông và còn $|E| - k$ cạnh. Đồ thị liên thông thì không thể có ít hơn $|V| - 1$ cạnh nên ta có $|E| - k \geq |V| - 1$ hay $k \leq |E| - |V| + 1$.
- Mọi cạnh trong một chu trình đơn bất kì đều phải thuộc một chu trình cơ sở. Bởi nếu có một cạnh (u, v) không thuộc một chu trình cơ sở nào, thì khi ta bỏ cạnh đó đi đồ thị vẫn liên thông và không ảnh hưởng tới sự tồn tại của các chu trình cơ sở. Lại bỏ tiếp $|E| - |V| + 1$ cạnh ngoài của các chu trình cơ sở thì đồ thị vẫn liên thông và còn lại $|V| - 2$ cạnh. Điều này vô lí.

Đối với đồ thị $G = (V, E)$ có k thành phần liên thông, ta có thể xét các thành phần liên thông và xét rùng các cây khung của các thành phần đó. Khi đó có thể mở

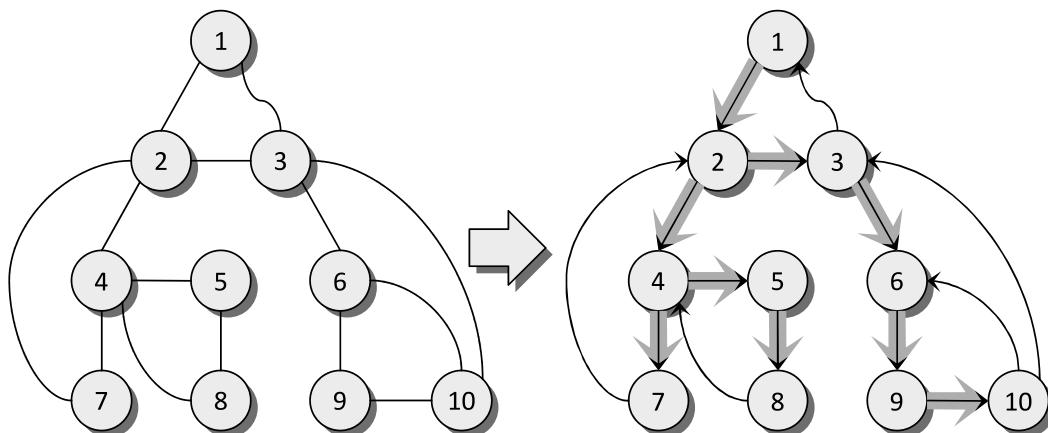
rộng khái niệm tập các chu trình cơ sở cho đồ thị vô hướng tổng quát: Mỗi khi thêm một cạnh không nằm trong các cây khung vào rừng, ta được đúng một chu trình đơn, tập các chu trình đơn tạo thành bằng cách ghép các cạnh ngoài như vậy gọi là tập các chu trình cơ sở của đồ thị G . Số các chu trình cơ sở là $|E| - |V| + k$.

5.3. Bài toán định chiều đồ thị

Bài toán đặt ra là cho một đồ thị vô hướng liên thông $G = (V, E)$, hãy thay mỗi cạnh của đồ thị bằng một cung định hướng để được một đồ thị có hướng liên thông mạnh. Nếu có phương án định chiều như vậy thì G được gọi là đồ thị định chiều được. Bài toán định chiều đồ thị có ứng dụng rõ nhất trong sơ đồ giao thông đường bộ. Chẳng hạn như trả lời câu hỏi: Trong một hệ thống đường phố, liệu có thể quy định các đường phố đó thành đường một chiều mà vẫn đảm bảo sự đi lại giữa hai nút giao thông bất kì hay không.

Có thể tổng quát hóa bài toán định chiều đồ thị: Với đồ thị vô hướng $G = (V, E)$ hãy tìm cách thay mỗi cạnh của đồ thị bằng một cung định hướng để được đồ thị mới có ít thành phần liên thông mạnh nhất. Dưới đây ta xét một tính chất hữu ích của thuật toán thuật toán tìm kiếm theo chiều sâu để giải quyết bài toán định chiều đồ thị

Xét mô hình duyệt đồ thị bằng thuật toán tìm kiếm theo chiều sâu, tuy nhiên trong quá trình duyệt, mỗi khi xét qua cạnh (u, v) thì ta định chiều luôn cạnh đó thành cung (u, v) . Nếu coi một cạnh của đồ thị tương đương với hai cung có hướng ngược chiều nhau thì việc định chiều cạnh (u, v) thành cung (u, v) tương đương với việc loại bỏ cung (v, u) của đồ thị. Ta có một phép định chiều gọi là phép định chiều DFS.



Hình 5-26. Phép định chiều DFS

Thuật toán thực hiện phép định chiều DFS có thể viết như sau:

```

procedure DFSVisit(u ∈ V);
begin
    «Thông báo thăm u và đánh dấu u đã thăm»;
    for ∀v: (u, v) ∈ E do
        begin
            «Định chiều cạnh (u, v) thành cung (u, v) ⇔ xoá cung
            (v, u) khỏi đồ thị»;
            if «v chưa thăm» then
                DFSVisit(v);
            end;
        end;
    begin
        «Đánh dấu mọi đỉnh đều chưa thăm»;
        for ∀v ∈ V do
            if «v chưa thăm» then DFSVisit(v);
    end;

```

Thuật toán DFS sẽ cho ta một rừng các cây DFS và các cung ngoài cây. Ta có các tính chất sau:

Định lí 5-16

Sau quá trình duyệt DFS và định chiều, đồ thị sẽ chỉ còn cung DFS và cung ngược.

Chứng minh

Xét một cạnh (u, v) bất kì, không giảm tính tổng quát, giả sử rằng u được duyệt đến trước v . Theo *Định lí 5-8 (định lí đường đi trắng)*, ta có v là hậu duệ của u . Nhìn vào mô hình cài đặt thuật toán, có nhận xét rằng việc định chiều cạnh (u, v) chỉ có thể được thực hiện trong thủ tục $DFSVisit(u)$ hoặc trong thủ tục $DFSVisit(v)$.

Nếu cạnh (u, v) được định chiều trước khi đỉnh v được duyệt đến, nghĩa là việc định chiều được thực hiện trong thủ tục $DFSVisit(u)$, và ngay sau khi cạnh (u, v) được định chiều thành cung (u, v) thì đỉnh v sẽ được thăm. Điều đó chỉ ra rằng cung (u, v) là cung DFS.

Nếu cạnh (u, v) được định chiều sau khi đỉnh v được duyệt đến, nghĩa là khi thủ tục $DFSVisit(v)$ được gọi thì cạnh (u, v) chưa định chiều. Vòng lặp bên trong thủ tục $DFSVisit(v)$ chắc chắn sẽ quét vào cạnh này và định chiều thành cung ngược (v, u) .

Trong đồ thị vô hướng ban đầu, cạnh bị định hướng thành cung ngược chính là cạnh ngoài của cây DFS. Chính vì vậy, mọi chu trình cơ sở của cây DFS trong đồ thị vô hướng ban đầu vẫn sẽ là chu trình trong đồ thị có hướng tạo ra. Đây là một phương pháp hiệu quả để liệt kê các chu trình cơ sở của cây khung DFS: Vừa duyệt DFS vừa định chiều, nếu duyệt phải cung ngược (u, v) thì truy vết đường đi của DFS để tìm đường từ v đến u , sau đó nối thêm cung ngược (u, v) để được một chu trình cơ sở.

Định lí 5-17

Điều kiện cần và đủ để một đồ thị vô hướng liên thông có thể định chiều được là mỗi cạnh của đồ thị nằm trên ít nhất một chu trình đơn (hay nói cách khác mọi cạnh của đồ thị đều không phải là cầu).

Chứng minh

Gọi $G = (V, E)$ là một đồ thị vô hướng liên thông.

" \Rightarrow "

Nếu G là định chiều được thì sau khi định hướng sẽ được đồ thị liên thông mạnh G' . Với một cạnh (u, v) được định chiều thành cung (u, v) thì sẽ tồn tại một đường đi đơn trong G' theo các cạnh định hướng từ v về u . Đường đi đó nối thêm cung (u, v) sẽ thành một chu trình đơn có hướng trong G' . Tức là trên đồ thị ban đầu, cạnh (u, v) nằm trên một chu trình đơn.

" \Leftarrow "

Nếu mỗi cạnh của G đều nằm trên một chu trình đơn, ta sẽ chứng minh rằng: phép định chiều DFS sẽ tạo ra đồ thị G' liên thông mạnh.

Lấy một cạnh (u, v) của G , vì (u, v) nằm trong một chu trình đơn, mà mọi cạnh của một chu trình đơn đều phải thuộc một chu trình cơ sở của cây DFS, nên sẽ có một chu trình cơ sở chứa cạnh (u, v) . Có thể nhận thấy rằng chu trình cơ sở của cây DFS qua phép định chiều DFS vẫn là chu trình trong G' nên theo các cung đã định hướng của chu trình đó ta có thể đi từ u tới v và ngược lại.

Lấy x và y là hai đỉnh bất kì của G , do G liên thông, tồn tại một đường đi

$$\langle x = v_0, v_1, \dots, v_k = y \rangle$$

Vì (v_i, v_{i+1}) là cạnh của G nên theo chứng minh trên, từ v_i có thể đi đến được v_{i+1} trên G' , $\forall i: 1 \leq i < k$, tức là từ x vẫn có thể đi đến y bằng các cung định hướng của G' . Suy ra G' là đồ thị liên thông mạnh

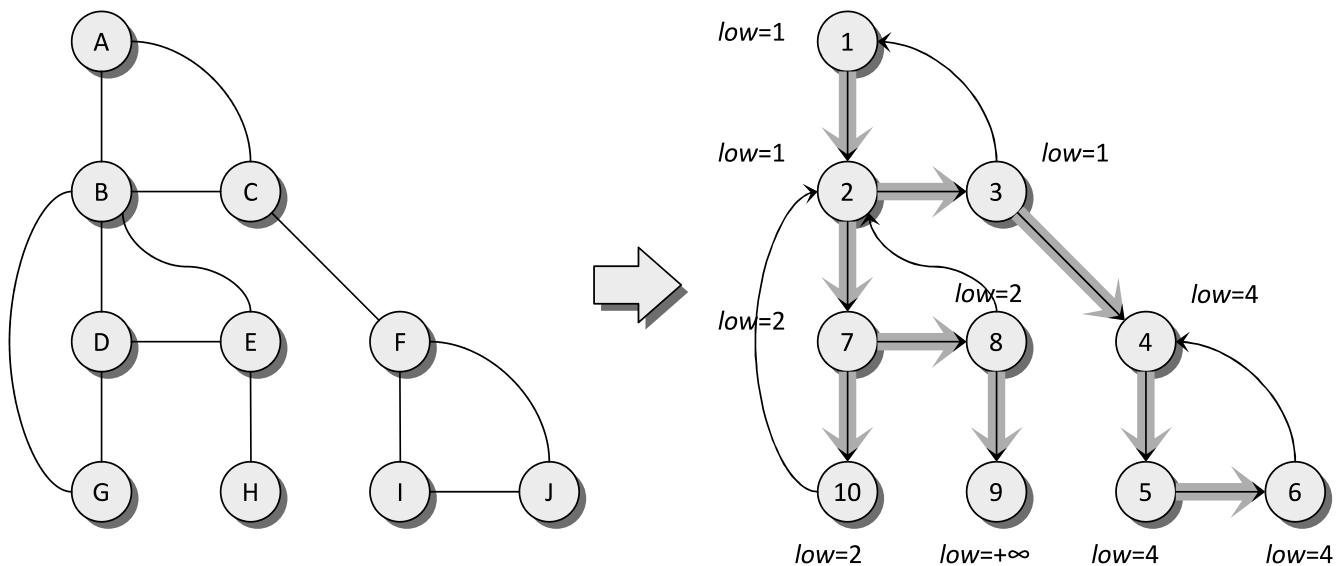
Với những kết quả đã chứng minh trên, ta còn suy ra được: Nếu đồ thị liên thông và mỗi cạnh của nó nằm trên ít nhất một chu trình đơn thì phép định chiều DFS sẽ cho một đồ thị liên thông mạnh. Còn nếu không, thì phép định chiều DFS sẽ cho một đồ thị định hướng có ít thành phần liên thông mạnh nhất, một cạnh không nằm trên một chu trình đơn nào (cầu) của đồ thị ban đầu sẽ được định hướng thành cung nối giữa hai thành phần liên thông mạnh.

5.4. Liệt kê các khớp và cầu của đồ thị

Nếu trong quá trình định chiều ta thêm vào đó thao tác đánh số các đỉnh theo thứ tự duyệt đến của thuật toán DFS, gọi $Number[u]$ là số thứ tự của đỉnh u theo cách đánh số đó. Định nghĩa thêm $Low[u]$ là giá trị $Number[.]$ nhỏ nhất của những đỉnh đến được từ nhánh DFS gốc u bằng một cung ngược. Tức là nếu nhánh DFS gốc u có nhiều cung ngược hướng lên phía gốc thì ta ghi nhận lại cung ngược hướng lên cao nhất. Nếu nhánh DFS gốc u không chứa cung ngược thì ta cho

$Low[u] := +\infty$. Cách tính các giá trị $Number[.]$ và $Low[.]$ tương tự như trong thuật toán Tarjan: Trong thủ tục $DFSVisit(u)$, trước hết ta đánh số thứ tự thăm cho đỉnh u ($Number[u]$) và khởi tạo $Low[u] := +\infty$, sau đó xét tất cả những đỉnh v kề u , định chiều cạnh (u, v) thành cung (u, v) . Có hai khả năng xảy ra:

- Nếu v chưa thăm thì ta gọi $DFSVisit(v)$ để thăm v , khi thủ tục $DFSVisit(v)$ thoát có nghĩa là đã xây dựng được nhánh DFS gốc v nằm trong nhánh DFS gốc u , những cung ngược đi từ nhánh DFS gốc v cũng là cung ngược đi từ nhánh DFS gốc $u \Rightarrow$ ta cực tiểu hoá $Low[u]$ theo công thức: $Low[u]_{\text{mới}} := \min(Low[u]_{\text{cũ}}, Low[v])$
- Nếu v đã thăm thì (u, v) là một cung ngược đi từ nhánh DFS gốc $u \Rightarrow$ ta cực tiểu hoá $Low[u]$ theo công thức: $Low[u]_{\text{mới}} := \min(Low[u]_{\text{cũ}}, Number[v])$



Hình 5-27. Cách đánh số và ghi nhận cung ngược lên cao nhất

Hãy để ý một cung DFS (u, v) (u là nút cha của nút v trên cây DFS)

- Nếu từ nhánh DFS gốc v không có cung nào ngược lên phía trên v có nghĩa là từ một đỉnh thuộc nhánh DFS gốc v đi theo các cung định hướng chỉ đi được tới những đỉnh nội bộ trong nhánh DFS gốc v mà thôi chứ không thể tới được u , suy ra (u, v) là một cầu. Cũng dễ dàng chứng minh được điều ngược lại. Vậy (u, v) là cầu nếu và chỉ nếu $Low[v] \geq Number[v]$. Như ví dụ ở Hình 5-27, ta có (C, F) và (E, H) là cầu.
- Nếu từ nhánh DFS gốc v không có cung nào ngược lên phía trên u , tức là nếu bỏ u đi thì từ v không có cách nào lên được các tiền bối của u . Điều này chỉ ra rằng nếu u không phải là nút gốc của một cây DFS thì u là khớp. Cũng không khó khăn để chứng minh điều ngược lại. Vậy nếu u không là gốc của

một cây DFS thì u là khớp nếu và chỉ nếu $Low[v] \geq Number[u]$. Như ví dụ ở Hình 5-27, ta có B, C, E và F là khớp.

- Gốc của một cây DFS thì là khớp nếu và chỉ nếu nó có từ hai nhánh con trở lên. Như ví dụ ở Hình 5-27, gốc A không là khớp vì nó chỉ có một nhánh con. Đến đây ta đã có đủ điều kiện để giải bài toán liệt kê các khớp và cầu của đồ thị: đơn giản là dùng phép định chiều DFS đánh số các đỉnh theo thứ tự thăm và ghi nhận cung ngược lên cao nhất xuất phát từ một nhánh cây DFS, sau đó dùng ba nhận xét kể trên để liệt kê ra tất cả các cầu và khớp của đồ thị.

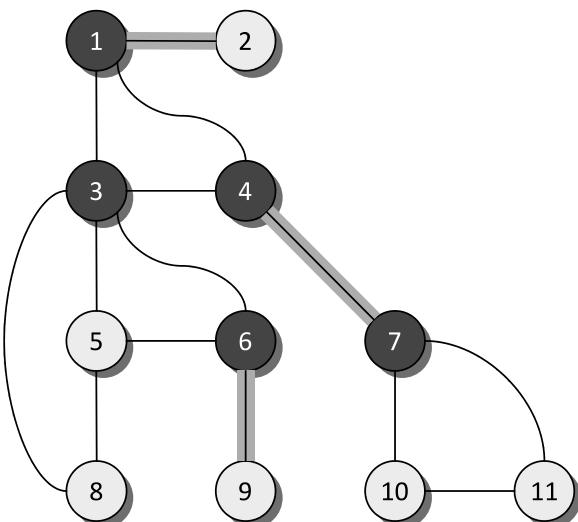
Input

- Dòng 1: Chứa số đỉnh $n \leq 1000$, số cạnh m của đồ thị vô hướng G .
- m dòng tiếp theo, mỗi dòng chứa hai số u, v tương ứng với một cạnh (u, v) của G

Output

Các khớp và cầu của G

Sample Input	Sample Output
11 14	Bridges:
1 2	(1, 2)
1 3	(4, 7)
1 4	(6, 9)
3 4	Articulations:
3 5	1
3 6	3
3 8	4
4 7	6
5 6	7
5 8	
6 9	
7 10	
7 11	
10 11	



Về kĩ thuật cài đặt, ngoài các mảng đã được nói tới khi trình bày thuật toán, có thêm một mảng $Parent[1 \dots n]$, trong đó $Parent[v]$ chỉ ra nút cha của nút v trên cây DFS, nếu v là gốc của một cây DFS thì $Parent[v]$ được đặt bằng -1 . Công dụng của mảng $Parent[1 \dots n]$ là để duyệt tất cả các cung DFS và kiểm tra một đỉnh có phải là gốc của cây DFS hay không.

CUTVE.PAS ✓ Liệt kê các khớp và cầu của đồ thị

```
{ $MODE OBJFPC }
program ArticulationsAndBridges;
```

```

const
  maxN = 1000;
var
  a: array[1..maxN, 1..maxN] of Boolean;
  Number, Low, Parent: array[1..maxN] of Integer;
  n, Count: Integer;
procedure Enter; //Nhập dữ liệu
var
  i, m, u, v: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m);
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
end;
//Hàm cực tiểu hóa: Target := min(Target, Value)
procedure Minimize(var Target: Integer; Value: Integer);
begin
  if Value < Target then Target := Value;
end;
procedure DFSVisit(u: Integer); //Thuật toán tìm kiếm theo chiều sâu bắt đầu
từ u
var
  v: Integer;
begin
  Inc(Count);
  Number[u] := Count; //Đánh số u theo thứ tự duyệt đến
  Low[u] := maxN + 1; //Đặt Low[u] := +∞
  for v := 1 to n do
    if a[u, v] then //Xét các đỉnh v kề u
      begin
        a[v, u] := False; //Định chiều cạnh (u, v) thành cung (u, v)
        if Parent[v] = 0 then //Nếu v chưa thăm
          begin
            Parent[v] := u; //cung (u, v) là cung DFS
            DFSVisit(v); //Di thăm v
            Minimize(Low[u], Low[v]); //Cực tiểu hóa Low[u] theo Low[v]
          end
      end
end;

```

```

        else
            Minimize (Low[u] , Number[v]) ; //Cực tiểu hóa Low[u] theo
Number[v]
        end;
end;
procedure Solve;
var
    u, v: Integer;
begin
    Count := 0; //Khởi tạo bộ đếm
    FillChar(Parent, SizeOf(Parent), 0); //Các đỉnh đều chưa thăm
    for u := 1 to n do
        if Parent[u] = 0 then
            begin
                Parent[u] := -1;
                DFSVisit(u);
            end;
    end;
    procedure PrintResult; //In kết quả
    var
        u, v: Integer;
        nChildren: array[1..maxN] of Integer;
        IsArticulation: array[1..maxN] of Boolean;
    begin
        WriteLn('Bridges: '); //Liệt kê các cầu
        for v := 1 to n do
            begin
                u := Parent[v];
                if (u <> -1) and (Low[v] >= Number[v]) then
                    WriteLn('(' , u, ', ', v, ')');
            end;
        WriteLn('Articulations:'); //Liệt kê các khớp
        FillChar(nChildren, n * SizeOf(Integer), 0);
        for v := 1 to n do
            begin
                u := Parent[v];
                if u <> -1 then Inc(nChildren[u]);
            end;
        //Đánh dấu các gốc cây có nhiều hơn 1 nhánh con
        for u := 1 to n do
            IsArticulation[u] := (Parent[u] = -1) and (nChildren[u]
>= 2);
    end;

```

```

for v := 1 to n do
begin
    u := Parent[v];
    if (u <> -1) and (Parent[u] <> -1) and (Low[v] >=
Number[u]) then
        IsArticulation[u] := True; //Đánh dấu các khớp không phải gốc
cây
    end;
    for u := 1 to n do //Liệt kê
        if IsArticulation[u] then
            WriteLn(u);
    end;
begin
    Enter;
    Solve;
    PrintResult;
end.

```

Trong bài toán liệt kê các khớp và cầu của đồ thị, ta biểu diễn đồ thị bằng ma trận kè để tiện lợi cho thao tác định chiều. Nếu đồ thị có số đỉnh n lớn (không thể biểu diễn được bằng ma trận kè) và số cạnh m nhỏ (đồ thị thừa), chúng ta phải tìm một cấu trúc dữ liệu khác để biểu diễn đồ thị để chi phí về bộ nhớ và thời gian phụ thuộc chủ yếu vào m thay vì n^2 như ma trận kè. Trong các cấu trúc dữ liệu biểu diễn đồ thị phổ biến, chỉ có danh sách kè và danh sách liên thuộc cho phép thực hiện điều này, tuy nhiên việc thực hiện định chiều cạnh vô hướng thành cung có hướng sẽ trở nên khá phức tạp.

Error! Reference source not found. yêu cầu bạn sửa đổi thuật toán để bỏ đi thao tác định chiều, từ đó có thể biểu diễn đồ thị thừa bởi danh sách kè mà không còn gặp khó khăn trong việc định chiều đồ thị nữa.

5.5. Các thành phần song liên thông

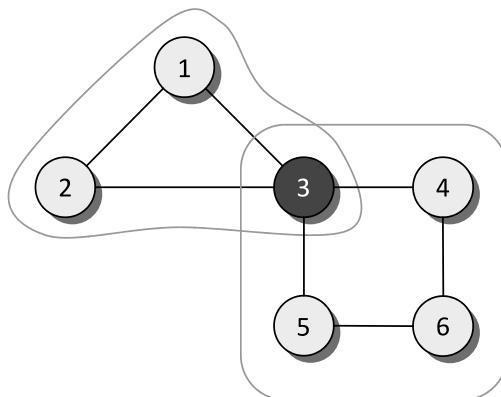
a) Các khái niệm và thuật toán

Đồ thị vô hướng liên thông được gọi là đồ thị song liên thông nếu nó không có khớp, tức là việc bỏ đi một đỉnh bất kì của đồ thị không ảnh hưởng tới tính liên thông của các đỉnh còn lại. Ta quy ước rằng đồ thị chỉ gồm một đỉnh và không có cạnh nào cũng là một đồ thị song liên thông.

Cho đồ thị vô hướng $G = (V, E)$, xét một tập con $V' \subset V$. Gọi G' là đồ thị G hạn chế trên V' . Đồ thị G' được gọi là một thành phần song liên thông của đồ thị G nếu G' song liên thông và không tồn tại đồ thị con song liên thông nào khác của G .

nhận G' làm đồ thị con. Ta cũng đồng nhất khái niệm G' là thành phần song liên thông với khái niệm V' là thành phần song liên thông.

Cần phân biệt hai khái niệm đồ thị định chiều được (không có cầu) và đồ thị song liên thông (không có khớp). Nếu như đồ thị G không định chiều được thì *tập đỉnh* của G có thể phân hoạch thành các tập con rời nhau để đồ thị G hạn chế trên các tập con đó là các đồ thị định chiều được. Còn nếu đồ thị G không phải đồ thị song liên thông thì *tập cạnh* của G có thể phân hoạch thành các tập con rời nhau để trên mỗi tập con, các cạnh và các đỉnh đầu mút của chúng trở thành một đồ thị song liên thông. Hai thành phần song liên thông có thể có chung một điểm khớp nhưng không có cạnh nào chung



Hình 5-28. Đồ thị và hai thành phần song liên thông có chung khớp

Xét mô hình định chiều đồ thị đánh số đỉnh theo thứ tự duyệt đến và ghi nhận cung ngược lên cao nhất...

```

procedure DFSVisit (u $\in$ V) ;
begin
    Count := Count + 1;
    Number[u] := Count; //Đánh số u theo thứ tự duyệt đến
    Low[u] := + $\infty$ ;
    for  $\forall v \in V : (u, v) \in E$  do
        begin
            «Định chiều cạnh (u, v) thành cung (u, v)»;
            if Number[v] > 0 then //v đã thăm
                Low[u] := min(Low[u], Number[v])
            else //v chưa thăm
                begin
                    DFSVisit(v); //Đi thăm v
                    Low[u] := min(Low[u], Low[v]); //Cực tiểu hóa Low[u]
                end;
        end;
end;
```

```

end;
begin
    Count := 0;
    for  $\forall v \in V$  do Number[v] := 0; //Number[v] = 0  $\Leftrightarrow$  v chưa thăm
    for  $\forall v \in V$  do
        if Number[v] = 0 then DFSVisit(v);
end.

```

Trong thủ tục $DFSVisit(u)$, mỗi khi xét các đỉnh v kề u chưa được thăm, thuật toán sẽ gọi $DFSVisit(v)$ để đi thăm v sau đó cực tiểu hoá $Low[u]$ theo $Low[v]$. Tại thời điểm này, nếu $Low[v] \geq Number[u]$ thì hoặc u là khớp hoặc u là gốc của một cây DFS. Để tiện, trong trường hợp này ta gọi cung (u, v) là *cung chót* của thành phần song liên thông.

Thuật toán tìm kiếm theo chiều sâu không chỉ duyệt qua các đỉnh mà còn duyệt và định chiều các cung nữa. Ta sẽ quan tâm tới cả thời điểm một cạnh được duyệt đến, duyệt xong, cũng như thứ tự tiền bối–hậu duệ của các cung DFS: Cung DFS (u, v) được coi là tiền bối thực sự của cung DFS (u', v') (hay cung (u', v') là hậu duệ thực sự của cung (u, v)) nếu cung (u', v') nằm trong nhánh DFS gốc v . Xét về vị trí trên cây, cung (u', v') nằm dưới cung (u, v) .

Có thể nhận thấy rằng nếu (u, v) là một cung chót thỏa mãn: Khi $DFSVisit(u)$ gọi $DFSVisit(v)$ và quá trình tìm kiếm theo chiều sâu tiếp tục từ v không thăm tiếp bất cứ một cung chót nào (tức là nhánh DFS gốc v không chứa cung chót nào) thì cung (u, v) hợp với tất cả các cung hậu duệ của nó sẽ tạo thành một nhánh cây mà mọi đỉnh thuộc nhánh cây đó là một thành phần song liên thông. Chính vì vậy thuật toán liệt kê các thành phần song liên thông có tư tưởng khá giống với thuật toán Tarjan tìm thành phần liên thông mạnh. Việc cài đặt thuật toán liệt kê các thành phần song liên thông chính là sự sửa đổi đối ngẫu của thuật toán Tarjan: Thay khái niệm “chốt” bằng “cung chót” và thay vì dùng ngăn xếp chứa chốt và các đỉnh hậu duệ của chốt để liệt kê các thành phần liên thông mạnh, chúng ta sẽ dùng ngăn xếp chứa cung chốt và các hậu duệ của cung chốt để liệt kê các thành phần song liên thông.

Vấn đề rắc rối duy nhất gặp phải là quy ước một đỉnh cô lập của đồ thị cũng là một thành phần song liên thông. Nếu thực hiện thuật toán trên, thành phần song liên thông chỉ gồm duy nhất một đỉnh sẽ không có cung chốt nào cả và như vậy sẽ bị sót khi liệt kê. Ta sẽ phải xử lý các đỉnh cô lập như trường hợp riêng khi liệt kê các thành phần song liên thông của đồ thị.

```
procedure DFSVisit ( $u \in V$ );
```

```

begin
    Count := Count + 1;
    Number[u] := Count; //Đánh số u theo thứ tự duyệt đến
    Low[u] := +∞;
    for ∀v∈V: (u, v) ∈ E do
        begin
            «Định chiều cạnh (u, v) thành cung (u, v)»;
            if Number[v] > 0 then //v đã thăm
                Low[u] := min(Low[u], Number[v])
            else //v chưa thăm
                begin
                    Push((u, v)); //Đẩy cung (u, v) vào ngăn xếp
                    DFSVisit(v); //Đi thăm v
                    Low[u] := min(Low[u], Low[v]); //Cực tiểu hóa Low[u]
                    if Low[v] ≥ Number[u] then //(u, v) là cung chốt
                        begin
                            «Thông báo thành phần song liên thông với cung
                            chốt (u, v):»;
                            repeat
                                (p, q) := Pop; //Lấy từ ngăn xếp ra một cung (p, q)
                                Output ← q; //Liệt kê các đỉnh nên chỉ cần xuất ra một đầu mút
                            until (p, q) = (u, v);
                            Output ← u; //Còn thiếu đỉnh u, liệt kê nó
                        end;
                end;
            end;
        end;
    end;
begin
    Count := 0;
    for ∀v∈V do Number[v] := 0; //Number[v] = 0 ⇔ v chưa thăm
    Stack := ∅;
    for ∀v∈V do
        if Number[v] = 0 then
            begin
                DFSVisit(v);
                if «v là đỉnh cô lập» then
                    «Liệt kê thành phần song liên thông chỉ gồm một
                    đỉnh v»
            end;
    end.

```

b) Cài đặt

Về kĩ thuật cài đặt không có gì mới, có một chú ý nhỏ là chúng ta chỉ dùng ngăn xếp *Stack* để chứa các cung DFS, vì vậy *Stack* không bao giờ phải chứa quá $n - 1$ cung

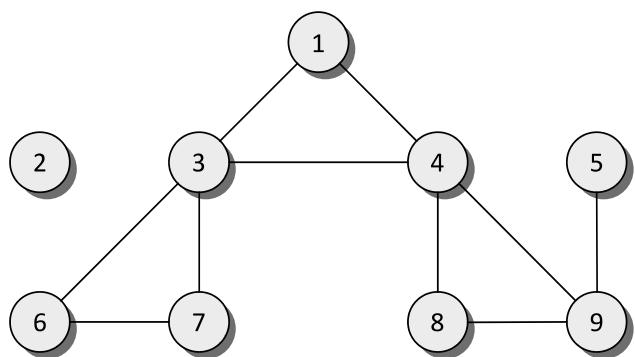
Input

- Dòng 1: Chứa số đỉnh $n \leq 1000$ và số cạnh m của một đồ thị vô hướng
- m dòng tiếp theo, mỗi dòng chứa hai số u, v tương ứng với một cạnh (u, v) của đồ thị.

Output

Các thành phần song liên thông của đồ thị

Sample Input	Sample Output
9 10	Biconnected component: 1
1 3	5, 9
1 4	Biconnected component: 2
3 4	9, 8, 4
3 6	Biconnected component: 3
3 7	7, 6, 3
4 8	Biconnected component: 4
4 9	4, 3, 1
5 9	Biconnected component: 5
6 7	
8 9	



BCC.PAS ✓ Liệt kê các thành phần song liên thông

```
{$MODE OBJFPC}
program BiconnectedComponents;
const
  maxN = 1000;
type
  TStack = record
    x, y: array[1..maxN - 1] of Integer;
    Top: Integer;
  end;
var
  a: array[1..maxN, 1..maxN] of Boolean;
```

```

Number, Low: array[1..maxN] of Integer;
Stack: TStack;
BCC, PrevCount, Count, n, u: Integer;
procedure Enter; //Nhập dữ liệu
var
  i, m, u, v: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m);
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
end;
procedure Push(u, v: Integer); //Đẩy một cung (u, v) vào ngăn xếp
begin
  with Stack do
    begin
      Inc(Top);
      x[Top] := u;
      y[Top] := v;
    end;
end;
procedure Pop(var u, v: Integer); //Lấy một cung (u, v) khỏi ngăn xếp
begin
  with Stack do
    begin
      u := x[Top];
      v := y[Top];
      Dec(Top);
    end;
end;
//Hàm cực tiểu hóa: Target := min(Target, Value)
procedure Minimize(var Target: Integer; Value: Integer);
begin
  if Value < Target then Target := Value;
end;
procedure DFSVisit(u: Integer); //Thuật toán tìm kiếm theo chiều sâu
var
  v, p, q: Integer;

```

```

begin
  Inc(Count);
  Number[u] := Count;
  Low[u] := maxN + 1;
  for v := 1 to n do
    if a[u, v] then //Xét mọi cạnh (u, v)
      begin
        a[v, u] := False; //Định chiều luôn
        if Number[v] <> 0 then //v đã thăm
          Minimize(Low[u], Number[v])
        else //v chưa thăm
          begin
            Push(u, v); //Đẩy cung DFS (u, v) vào Stack
            DFSVisit(v); //Tiếp tục quá trình DFS từ v
            Minimize(Low[u], Low[v]);
            if Low[v] >= Number[u] then //Nếu (u, v) là cung chốt
              begin //Liệt kê thành phần song liên thông với cung chốt (u, v)
                Inc(BCC);
                WriteLn('Biconnected component: ', BCC);
                repeat
                  Pop(p, q); //Lấy một cung DFS (p, q) khỏi Stack
                  Write(q, ' ', ' ');
                  until (p = u) and (q = v); //Đến khi lấy ra cung (u, v)
                thi dừng
                  WriteLn(u); //In nốt ra đỉnh u
                end;
              end;
            end;
          end;
    end;
begin
  Enter;
  FillChar(Number, n * SizeOf(Integer), 0);
  Stack.Top := 0;
  Count := 0;
  BCC := 0;
  for u := 1 to n do
    if Number[u] = 0 then
      begin
        PrevCount := Count;
        DFSVisit(u);
        if Count = PrevCount + 1 then //u là đỉnh cô lập
          begin

```

```

Inc(BCC);
WriteLn('Biconnected component: ', BCC);
WriteLn(u);
end;
end;
end.

```

Bài tập

- 5.20.** Hãy sửa đổi thuật toán liệt kê khốp và cầu của đồ thị, sửa đổi thuật toán liệt kê các thành phần song liên thông sao cho không cần phải thực hiện việc định chiều đồ thị nữa (Bởi vì việc định chiều một đồ thị tỏ ra khá công kềnh và không hiệu quả nếu đồ thị được biểu diễn bằng danh sách kè hay danh sách cạnh)
- 5.21.** Tìm thuật toán đếm số cây khung của đồ thị (Hai cây khung gọi là khác nhau nếu chúng có ít nhất một cạnh khác nhau)

6. Đồ thị Euler và đồ thị Hamilton

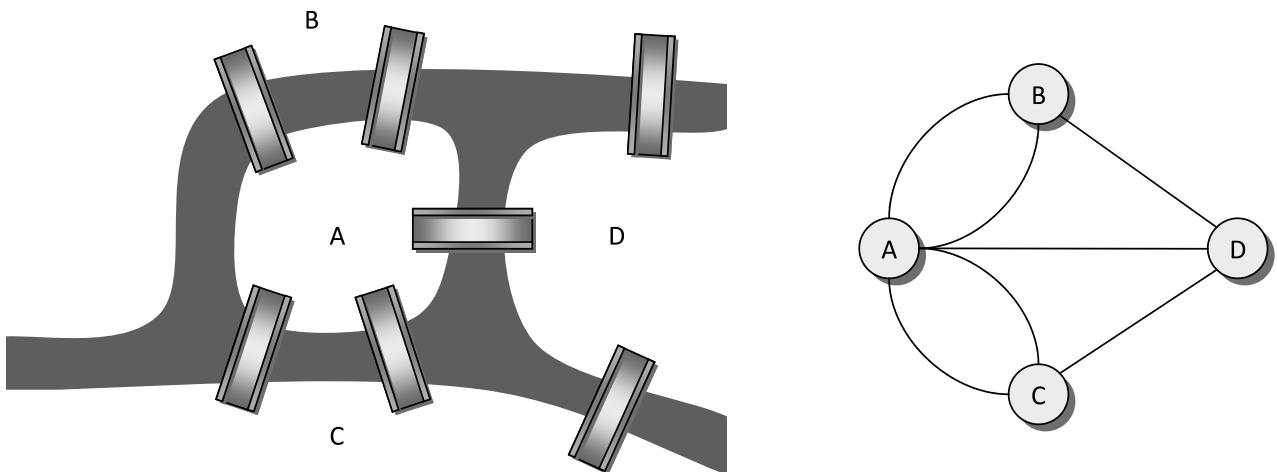
6.1. Đồ thị Euler

a) Bài toán

Bài toán về đồ thị Euler được coi là bài toán đầu tiên của lí thuyết đồ thị. Bài toán này xuất phát từ một bài toán nổi tiếng: Bài toán bảy cây cầu ở Königsberg:

Thành phố Königsberg thuộc Đức (nay là Kaliningrad thuộc Cộng hoà Nga), được chia làm 4 vùng bằng các nhánh sông Pregel. Các vùng này gồm 2 vùng bên bờ sông (B, C), đảo Kneiphof (A) và một miền nằm giữa hai nhánh sông Pregel (D). Vào thế kỷ XVIII, người ta đã xây 7 chiếc cầu nối những vùng này với nhau. Người dân ở đây tự hỏi: Liệu có cách nào xuất phát tại một địa điểm trong thành phố, đi qua 7 chiếc cầu, mỗi chiếc đúng 1 lần rồi quay trở về nơi xuất phát không ?

Nhà toán học Thụy sĩ Leonhard Euler đã giải bài toán này và có thể coi đây là ứng dụng đầu tiên của Lí thuyết đồ thị, ông đã mô hình hoá sơ đồ 7 cái cầu bằng một đa đồ thị, bốn vùng được biểu diễn bằng 4 đỉnh, các cầu là các cạnh. Bài toán tìm đường qua 7 cầu, mỗi cầu đúng một lần có thể tổng quát hoá bằng bài toán: Có tồn tại chu trình trong đa đồ thị đi qua tất cả các cạnh và mỗi cạnh đúng một lần.



Hình 5-29: Mô hình đồ thị của bài toán báy cái cầu

Chu trình qua tất cả các cạnh của đồ thị, mỗi cạnh đúng một lần được gọi là *chu trình Euler* (*Euler circuit/Euler circle/Euler tour*). Đường đi qua tất cả các cạnh của đồ thị, mỗi cạnh đúng một lần gọi là *đường đi Euler* (*Euler path/Euler trail/Euler walk*). Một đồ thị có chu trình Euler được gọi là *đồ thị Euler* (*Eulerian graph/unicursal graph*). Một đồ thị có đường đi Euler được gọi là *đồ thị nửa Euler* (*Semi-Eulerian graph/Traversable graph*).

b) Các định lí và thuật toán

Định lí 5-18 (Euler)

Một đồ thị vô hướng liên thông $G = (V, E)$ có chu trình Euler khi và chỉ khi mọi đỉnh của nó đều có bậc chẵn.

Chứng minh

Nếu G có chu trình Euler thì khi đi dọc chu trình đó, mỗi khi đi qua một đỉnh thì bậc của đỉnh đó tăng lên 2 (một lần vào + một lần ra). Chu trình Euler lại đi qua tất cả các cạnh nên suy ra mọi đỉnh của đồ thị đều có bậc chẵn.

Ngược lại nếu G liên thông và mọi đỉnh đều có bậc chẵn, ta sẽ chỉ ra thuật toán xây dựng chu trình Euler trên G .

Xuất phát từ một đỉnh bất kỳ, ta đi sang một đỉnh tùy ý kề nó, đi qua cạnh nào xoá luôn cạnh đó cho tới khi không đi được nữa, có thể nhận thấy rằng sau mỗi bước đi, chỉ có đỉnh đầu và đỉnh cuối của đường đi có bậc lẻ còn mọi đỉnh khác trong đồ thị đều có bậc chẵn. Cạnh cuối cùng đi qua chắc chắn là đi tới một đỉnh bậc lẻ, vì nếu là cạnh đi tới một đỉnh bậc chẵn thì đỉnh này sẽ có ít nhất 2 cạnh liên thuộc, và như vậy khi đi tới đỉnh này và xoá cạnh vào ta vẫn còn một cạnh để ra, quá trình đi chưa kết thúc. Điều này chỉ ra rằng cạnh cuối cùng bắt buộc phải đi về nơi xuất phát tức là chúng ta có một chu trình C . Cũng dễ dàng nhận thấy rằng khi quá trình này kết thúc, mọi đỉnh của G vẫn có bậc chẵn.

Nếu G còn lại cạnh liên thuộc với một đỉnh v nào đó trên C thì lại bắt đầu từ v , ta đi một cách tùy ý theo các cạnh còn lại của G ta sẽ được một chu trình C' bắt đầu từ v và kết thúc

tại v . Thay thế một bước đi qua đỉnh v trên C bằng cả chu trình C' , ta sẽ được một chu trình mới lớn hơn. Quy trình được lặp lại cho tới khi C không còn đỉnh nào có cạnh liên thuộc nằm ngoài C . Do tính liên thông của G , điều này có nghĩa là C chứa tất cả các cạnh của G hay C là chu trình Euler trên đồ thị ban đầu.

Hệ quả

Một đồ thị vô hướng liên thông $G = (V, E)$ có đường đi Euler khi và chỉ khi nó có đúng 2 đỉnh bậc lẻ.

Chứng minh

Nếu G có đường đi Euler thì chỉ có đỉnh bắt đầu và đỉnh kết thúc đường đi có bậc lẻ còn mọi đỉnh khác đều có bậc chẵn. Ngược lại nếu đồ thị liên thông có đúng 2 đỉnh bậc lẻ thì ta thêm vào một cạnh giả nối hai đỉnh bậc lẻ đó và tìm chu trình Euler. Loại bỏ cạnh giả khỏi chu trình, chúng ta sẽ được đường đi Euler.

Định lí 5-19

Một đồ thi có hướng liên thông yếu $G = (V, E)$ có chu trình Euler thì mọi đỉnh của nó có bán bậc ra bằng bán bậc vào: $\deg^+(v) = \deg^-(v), \forall v \in V$; Ngược lại, nếu G liên thông yếu và mọi đỉnh của nó có bán bậc ra bằng bán bậc vào, thì G có chu trình Euler (suy ra G sẽ là liên thông mạnh).

Chứng minh

Tương tự như phép chứng minh Định lí 5.18.

Hệ quả

Một đồ thi có hướng liên thông yếu $G = (V, E)$ có đường đi Euler nhưng không có chu trình Euler nếu tồn tại đúng hai đỉnh $s, t \in V$ sao cho:

$$\deg^+(s) - \deg^-(s) = \deg^-(t) - \deg^+(t) = 1$$

còn tất cả những đỉnh còn lại của đồ thi đều có bán bậc ra bằng bán bậc vào.

Việc chứng minh *Định lí 5-18 (Euler)* cho ta một thuật toán hữu hiệu để chỉ ra chu trình Euler trên đồ thi Euler. Thuật toán này hoạt động dựa trên một ngăn xếp *Stack* và được mô tả cụ thể như sau: Bắt đầu từ đỉnh 1, ta đi thoái mái theo các cạnh của đồ thi cho tới khi không đi được nữa, đi tới đỉnh nào ta đẩy đỉnh đó vào ngăn xếp và đi qua cạnh nào thì ta xoá cạnh đó khỏi đồ thi. Khi không đi được nữa thì ngăn xếp sẽ chứa các đỉnh trên một chu trình C bắt đầu và kết thúc ở đỉnh 1. Sau đó chúng ta lấy lần lượt các đỉnh ra khỏi ngăn xếp tương đương với việc đi ngược chu trình C . Nếu đỉnh được lấy ra (u) không có cạnh nào còn lại liên thuộc với nó thì u sẽ được ghi ra chu trình Euler, ngược lại, nếu u vẫn còn có cạnh liên thuộc thì ta lại đi tiếp từ u theo cách trên và đẩy thêm vào ngăn xếp một chu trình

C' bắt đầu và kết thúc tại u , để khi lấy các đỉnh ra khỏi ngăn xếp sẽ tương đương với việc đi ngược lại chu trình C' rồi tiếp tục đi ngược phần còn lại của chu trình C trong ngăn xếp... Có thể hình dung là thuật toán lần ngược chu trình C , khi đến đỉnh u thì thay u bằng cả một chu trình C' ...

Khi cài đặt thuật toán, chúng ta cần trang bị ba phép toán trên ngăn xếp *Stack*:

- *Push(v)*: Đẩy một đỉnh v vào *Stack*
- *Pop*: Lấy ra một đỉnh khỏi *Stack*
- *Get*: Đọc phần tử ở đỉnh *Stack*

```

Stack := (1); //Ngăn xếp ban đầu chỉ chứa một đỉnh bất kì,
chẳng hạn đỉnh 1
repeat
    u := Get; //Đọc phần tử ở đỉnh ngăn xếp
    if ∃(u, v) ∈ E then //Từ u còn đi tiếp được
        begin
            Push(v);
            E := E - { (u, v) }; //Xoá cạnh (u, v) khỏi đồ thị
        end;
    else //Từ u không đi đâu được nữa
        begin
            u := Pop; //Lấy u khỏi ngăn xếp
            Output ← u; //In ra u
        end;
    until Stack = ∅; //Lặp tới khi ngăn xếp rỗng

```

c) Cài đặt

Dưới đây chúng ta sẽ cài đặt thuật toán tìm chu trình Euler trên đa đồ thị Euler vô hướng $G = (V, E)$. Dữ liệu vào luôn đảm bảo đồ thị liên thông, có ít nhất một đỉnh và mọi đỉnh đều có bậc chẵn.

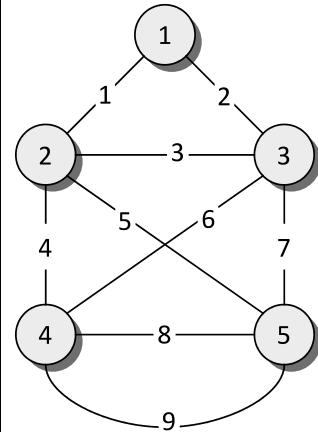
Input

- Dòng 1 chứa số đỉnh $n \leq 10^5$ và số cạnh $m \leq 10^6$
- m dòng tiếp, mỗi dòng chứa số hiệu hai đầu mút của một cạnh.

Output

Chu trình Euler

Sample Input	Sample Output
5 9	1 2 4 5 4 3 5 2 3 1
1 2	
1 3	
2 3	
2 4	
2 5	
3 4	
3 5	
4 5	
4 5	



Ngoài các thao tác đối với ngăn xếp, thuật toán tìm chu trình Euler còn yêu cầu cài đặt hai thao tác sau đây một cách hiệu quả:

- Với mỗi đỉnh kiểm tra xem có tồn tại cạnh liên thuộc với nó hay không, nếu có thì chỉ ra một cạnh liên thuộc.
- Loại bỏ một cạnh khỏi đồ thị

Các cạnh của đồ thị được đánh số từ 1 tới m , sau đó mỗi cạnh vô hướng (x, y) sẽ được thay thế bởi hai cung có hướng ngược chiều: (x, y) và (y, x) . Mỗi cung là một bản ghi gồm hai đỉnh đầu mút và chỉ số cạnh vô hướng tương ứng.

```

const
  maxM = 1000000;
type
  TArc = record
    x, y: Integer; //cung (x, y)
    edge: Integer; //chỉ số cạnh vô hướng tương ứng
  end;
var
  a: array[1..2 * maxM] of TArc;

```

Danh sách liên thuộc được xây dựng theo kiểu reverse star: Mỗi đỉnh u cho tương ứng với một danh sách các cung đi vào u . Các danh sách này được cho bởi hai mảng $head[1 \dots n]$ và $link[1 \dots 2m]$ trong đó:

- $head[u]$ là chỉ số cung đầu tiên trong danh sách liên thuộc các cung đi vào u , trường hợp đỉnh u không còn cung đi vào, $head[u]$ được gán bằng 0.
- $link[i]$ là chỉ số cung kế tiếp cung a_i trong cùng danh sách liên thuộc chứa cung a_i , trường hợp a_i là cung cuối cùng trong một danh sách liên thuộc, $link[i]$ được gán bằng 0.

Để thực hiện thao tác xoá cạnh, ta duy trì một mảng đánh dấu $deleted[1 \dots m]$ trong đó $deleted[i] = \text{True}$ nếu cạnh vô hướng thứ i đã bị xoá. Mỗi khi cạnh vô hướng bị xoá, cả hai cung có hướng tương ứng đều không còn tồn tại, việc kiểm tra một cung có hướng a_i còn tồn tại hay không có thể thực hiện bằng việc kiểm tra: $deleted[a_i.\text{edge}] = ?$ False.

Chúng ta sẽ cài đặt các thao tác sau trên cấu trúc dữ liệu:

- Hàm *Get*: Trả về phần tử nằm ở đỉnh ngăn xếp.
- Hàm *Pop*: Trả về phần tử nằm ở đỉnh ngăn xếp và rút phần tử đó khỏi ngăn xếp.
- Thủ tục *Push(v)*: Đẩy một đỉnh v vào ngăn xếp.

Tất cả các thao tác trên ngăn xếp có thể cài đặt để thực hiện trong thời gian $O(1)$. Thuật toán tìm chu trình Euler có thể viết cụ thể hơn:

```

Stack := (1); //Khởi tạo ngăn xếp chỉ chứa một đỉnh
repeat
    u := Get; //Đọc đỉnh u từ ngăn xếp
    i := head[u]; //Xét cung a[i] đứng đầu danh sách liên thuộc
    các cung đi vào u
    while (i > 0) and (deleted[a[i].edge]) do //cung a[i] ứng
    với cạnh vô hướng đã xoá
        i := link[i]; //Dịch sang cung kế tiếp
        head[u] := i; //Những cung đã duyệt qua bị loại ngay, cập
        nhật lại chỉ số đầu danh sách liên thuộc
        if i > 0 then //u còn cung đi vào ứng với cạnh vô hướng
        chưa xoá
            begin
                Push(a[i].x); //Đẩy đỉnh nối tới u vào ngăn xếp (đi
                ngược cung a[i])
                Deleted[a[i].edge] := True; //Xoá ngay cạnh vô hướng
                ứng với cung a[i]
            end
        else
            Output ← Pop;
    until Top = 0; //Lặp tới khi ngăn xếp rỗng

```

Xét vòng lặp repeat...until, mỗi bước lặp có một thao tác *Push* hoặc *Pop* được thực hiện. Mỗi lần thao tác *Push* được thực hiện phải có một cạnh vô hướng bị xoá và ngăn xếp có thêm một đỉnh. Mỗi lần thao tác *Pop* được thực hiện thì ngăn xếp bị bớt đi một đỉnh. Vì thuật toán in ra $m + 1$ đỉnh trên chu trình Euler nên sẽ phải có tổng cộng $m + 1$ thao tác *Pop*. Trước khi vào vòng lặp ngăn xếp có một

đỉnh và khi vòng lặp kết thúc ngăn xếp trở thành rỗng, suy ra số thao tác *Push* phải là m . Từ đó, vòng lặp repeat...until thực hiện $2m + 1$ lần.

Tiếp theo ta đánh giá số thao tác duyệt danh sách liên thuộc của đỉnh u . Bởi sau vòng lặp while có lệnh cập nhật $head[u] := i$ nên có thể thấy rằng lệnh gán $i := link[i]$ được thực hiện bao nhiêu lần thì danh sách liên thuộc của u bị giảm đi đúng chừng đó cung. Tổng số phần tử của các danh sách liên thuộc là $2m$ và khi thuật toán kết thúc, các danh sách liên thuộc đều rỗng. Suy ra tổng thời gian thực hiện phép duyệt danh sách liên thuộc (vòng lặp while) trong toàn bộ thuật toán là $\Theta(m)$.

Suy ra thời gian thực hiện giải thuật là $\Theta(m)$.

EULER.PAS ✓ Tìm chu trình Euler trong đa đồ thị Euler vô hướng

```
{ $MODE OBJFPC }
program EulerTour;
const
  maxN = 100000;
  maxM = 1000000;
type
  TArc = record //Cấu trúc một cung
    x, y: Integer; //Đỉnh đầu và đỉnh cuối
    edge: Integer; //Chỉ số cạnh vô hướng tương ứng
  end;
var
  n, m: Integer;
  a: array[1..2 * maxM] of TArc; //Danh sách các cung
  link: array[1..2 * maxM] of Integer; //link[i]: Chỉ số cung kế tiếp a[i]
  trong cùng danh sách liên thuộc
  head: array[1..maxN] of Integer; //head[u]: chỉ số cung đầu tiên trong
  danh sách các cung đi vào u
  deleted: array[1..maxM] of Boolean; //Đánh dấu cạnh vô hướng bị xoá
  hay chưa
  Stack: array[1..maxM + 1] of Integer; //Ngăn xếp
  Top: Integer; //Phần tử đỉnh ngăn xếp
procedure Enter; //Nhập dữ liệu và xây dựng danh sách liên thuộc
var
  i, j, u, v: Integer;
begin
  ReadLn(n, m);
  j := 2 * m;
  for i := 1 to m do
```

```

begin
    ReadLn(u, v); //Đọc một cạnh vô hướng, thêm 2 cung có hướng tương ứng
    a[i].x := u; a[i].y := v; a[i].edge := i;
    a[j].x := v; a[j].y := u; a[j].edge := i;
    Dec(j);
end;
FillChar(head[1], n * SizeOf(head[1]), 0); //Khởi tạo các danh sách liên thuộc rỗng
for i := 2 * m downto 1 do
    with a[i] do //Duyệt từng cung (x, y)
begin //Đưa cung đó vào danh sách liên thuộc các cung đi vào y
    link[i] := head[y];
    head[y] := i;
end;
FillChar(deleted[1], n * SizeOf(deleted[1]), False); //Các cạnh vô hướng đều chưa xoá
end;
procedure FindEulerTour;
var
    u, i: Integer;
begin
    Top := 1; Stack[1] := 1; //Khởi tạo ngăn xếp chứa đỉnh 1
repeat
    u := Stack[Top]; //Đọc phần tử ở đỉnh ngăn xếp
    i := head[u]; //Cung a[i] đang đứng đầu danh sách liên thuộc
    while (i > 0) and (deleted[a[i].edge]) do
        i := link[i]; //Dịch chỉ số i đọc danh sách liên thuộc để tìm cung ứng với
    cạnh vô hướng chưa xoá
    head[u] := i; //Cập nhật lại head[u], "nhảy" qua các cung ứng với cạnh vô
    hướng đã xoá
    if i > 0 then //u còn cung đi vào ứng với cạnh vô hướng chưa xoá
        begin
            Inc(Top); Stack[Top] := a[i].x; //Đi ngược cung a[i], đẩy đỉnh
            nối tới u vào ngăn xếp
            Deleted[a[i].edge] := True; //Xoá cạnh vô hướng tương ứng với a[i]
        end
    else //u không còn cung đi vào
        begin
            Write(u, ' '); //In ra u trên chu trình Euler
            Dec(Top); //Lấy u khỏi ngăn xếp
        end;
until Top = 0; //Lặp tới khi ngăn xếp rỗng
WriteLn;

```

```

end;
begin
    Enter;
    FindEulerTour;
end.

```

d) Vài nhận xét

Bằng việc quan sát hoạt động của ngăn xếp, chúng ta có thể sửa mô hình cài đặt của thuật toán nhằm tận dụng chính ngăn xếp của chương trình con đệ quy chứ không cần cài đặt cấu trúc dữ liệu ngăn xếp để chứa các đỉnh:

```

procedure Visit(u: Integer);
var
    i: Integer;
begin
    i := head[u];
    while i ≠ 0 do
        begin //Xét cung a[i] đi vào u
            if not deleted[a[i].edge] then //Cạnh vô hướng tương ứng chưa bị xoá
                begin
                    deleted[a[i].edge] := True; //Xoá cạnh vô hướng tương ứng
                    Visit(a[i].x); //Đi ngược chiều cung a[i] thăm đỉnh nối tới u
                end;
        end;
        Output ← u; //Tù u không thể đi ngược chiều cung nào nữa, in ra u trên chu trình Euler
    end;
    begin
        «Nhập đồ thị và xây dựng danh sách liên thuộc»;
        Visit(1); //Khởi động thuật toán tìm chu trình Euler
    end.

```

Cách cài đặt này khá đơn giản vì thao tác trên ngăn xếp được thực hiện tự nhiên qua cơ chế gọi và thoát thủ tục đệ quy. Tuy nhiên cần chú ý rằng độ sâu của dây chuyền đệ quy có thể lên tới $m + 1$ cấp nên với một số công cụ lập trình cần đặt lại dung lượng bộ nhớ Stack¹.

Chúng ta có thể liên hệ thuật toán này với thuật toán tìm kiếm theo chiều sâu: Từ mô hình DFS, nếu thay vì đi thăm đỉnh chúng ta đi thăm cạnh (một cạnh có thể đi tiếp sang cạnh chung đầu mút với nó). Đồng thời ta đánh dấu cạnh đã qua/chưa

¹ Trong Free Pascal 32 bit, dung lượng bộ nhớ Stack dành cho biến địa phương và tham số chương trình con mặc định là 64 KiB. Có thể đặt lại bằng dẫn hướng biên dịch {\$M...}

qua thay cho cơ chế đánh dấu một đỉnh đã thăm/chưa thăm. Khi đó *thứ tự duyệt xong (finish)* của các cạnh cho ta một chu trình Euler.

Thuật toán không có gì sai nếu ta xây dựng danh sách liên thuộc kiểu forward star thay vì kiểu reverse star. Tuy nhiên ta chọn kiểu reverse star bởi cách biểu diễn này thích hợp để tìm chu trình Euler trên cả đồ thị vô hướng và có hướng.

Người ta còn có thuật toán Fleury (1883) để tìm chu trình Euler bằng tay: Bắt đầu từ một đỉnh, chúng ta đi thoái mái theo các cạnh theo nguyên tắc: xoá bỏ các cạnh đi qua và chỉ đi qua cầu khi không còn cách nào khác để chọn. Khi không thể đi tiếp được nữa thì đường đi tìm được chính là chu trình Euler.

Bằng cách “lạm dụng thuật ngữ”, ta có thể mô tả được thuật toán tìm Fleury cho cả đồ thị Euler có hướng cũng như vô hướng:

- Dưới đây nếu ta nói cạnh (u, v) thì hiểu là cạnh (u, v) trên đồ thị vô hướng, hiểu là cung (u, v) trên đồ thị có hướng.
- Ta gọi cạnh (u, v) là “một đi không trở lại” nếu như từ u đi tới v , sau đó xoá cạnh này đi thì không có cách nào từ v quay lại u .

Thuật toán Fleury tìm chu trình Euler: Xuất phát từ một đỉnh, ta đi một cách tuỳ ý theo các cạnh tuân theo hai nguyên tắc: Xoá bỏ cạnh vừa đi qua và chỉ chọn cạnh “một đi không trở lại” nếu như không còn cạnh nào khác để chọn.

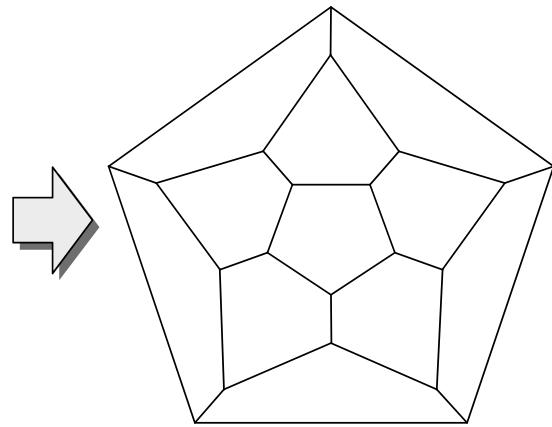
Thuật toán Fleury là một thuật toán thích hợp cho việc tìm chu trình Euler bằng tay (với những đồ thị vẽ ra được trên mặt phẳng thì việc kiểm tra cầu bằng mắt thường là tương đối dễ dàng). Tuy vậy khi cài đặt thuật toán trên máy tính thì thuật toán này tỏ ra không hiệu quả.

6.2. Đồ thị Hamilton

a) Bài toán

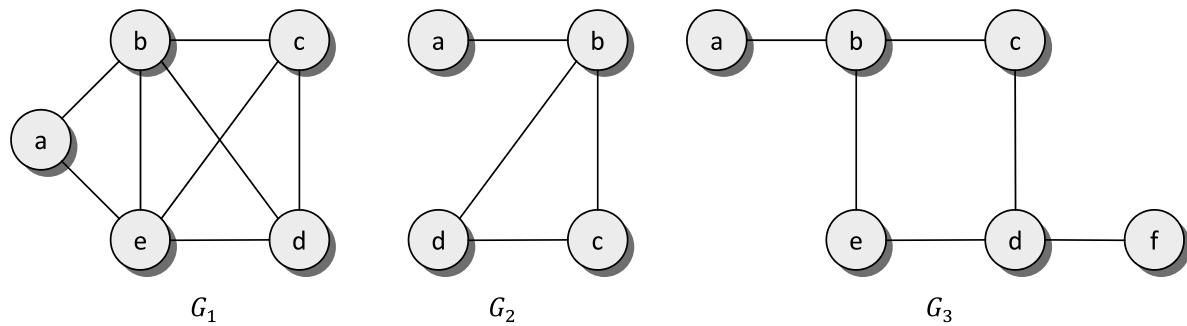
Khái niệm về đường đi và chu trình Hamilton được đưa ra bởi William Rowan Hamilton (1856) khi ông thiết kế một trò chơi trên khối đa diện 20 đỉnh, 30 cạnh, 12 mặt, mỗi mặt là một ngũ giác đều và người chơi cần chọn các cạnh để thành lập một đường đi qua 5 đỉnh cho trước (Hình 5-30).

Đồ thị $G = (V, E)$ được gọi là *đồ thị Hamilton (Hamiltonian graph)* nếu tồn tại chu trình đơn đi qua tất cả các đỉnh. Chu trình đơn đi qua tất cả các đỉnh được gọi là *chu trình Hamilton (Hamiltonian Circuit/Hamiltonian Circle)*. Để thuận tiện, người ta quy ước rằng đồ thị chỉ gồm 1 đỉnh là đồ thị Hamilton, nhưng đồ thị gồm 2 đỉnh liên thông không phải là đồ thị Hamilton.



Hình 5-30

Đồ thị $G = (V, E)$ được gọi là *đồ thị nửa Hamilton* (*traceable graph*) nếu tồn tại đường đi đơn qua tất cả các đỉnh. Đường đi đơn đi qua tất cả các đỉnh được gọi là *đường đi Hamilton* (*Hamiltonian Path*).



Hình 5-31

Trong Hình 5-31, Đồ thị G_1 có chu trình Hamilton $\langle a, b, c, d, e, a \rangle$. G_2 không có chu trình Hamilton nhưng có đường đi Hamilton $\langle a, b, c, d \rangle$. G_3 không có cả chu trình Hamilton lẫn đường đi Hamilton.

b) Các định lí liên quan

Từ định nghĩa ta suy ra được đồ thị đường của đồ thị Euler là một đồ thị Hamilton. Ngoài ra những định lí sau đây cho chúng ta vài cách nhận biết đồ thị Hamilton.

Định lí 5-20

Đồ thị vô hướng G , trong đó tồn tại k đỉnh sao cho nếu xoá đi k đỉnh này cùng với những cạnh liên thuộc của chúng thì đồ thị nhận được sẽ có nhiều hơn k thành phần liên thông thì khẳng định là G không phải đồ thị Hamilton

Định lí 5-21 (Định lí Dirak, 1952)

Xét đơn đồ thị vô hướng $G = (V, E)$ có $n \geq 3$ đỉnh. Nếu mọi đỉnh đều có bậc không nhỏ hơn $n/2$ thì G là đồ thị Hamilton.

Định lí 5-22 (Định lí Ghouila-Houiri, 1960)

Xét đơn đồ thị có hướng liên thông mạnh $G = (V, E)$ có n đỉnh. Nếu trên phiên bản vô hướng của G , mọi đỉnh đều có bậc không nhỏ hơn n thì G là đồ thị Hamilton.

Định lí 5-23 (Định lí Ore, 1960)

Xét đơn đồ thị vô hướng $G = (V, E)$ có $n \geq 3$ đỉnh. Với mọi cặp đỉnh không kề nhau có tổng bậc $\geq n$ thì G là đồ thị Hamilton.

Định lí 5-24 (Định lí Meynie, 1973)

Xét đơn đồ thị có hướng liên thông mạnh $G = (V, E)$ có n đỉnh. Nếu trên phiên bản vô hướng của G , với mọi cặp đỉnh không kề nhau có tổng bậc $\geq 2n - 1$ thì G là đồ thị Hamilton.

Định lí 5-25 (Định lí Bondy-Chvátal, 1972)

Xét đồ thị vô hướng $G = (V, E)$ có n đỉnh, với mỗi cặp đỉnh không kề nhau u, v mà $\deg(u) + \deg(v) \geq n$ ta thêm một cạnh nối u và v , cứ làm như vậy cho tới khi không thêm được cạnh nào nữa ta thu được đồ thị mới kí hiệu $cl(G)$. Khi đó G là đồ thị Hamilton nếu và chỉ nếu $cl(G)$ là đồ thị Hamilton.

Nếu đồ thị G thỏa mãn điều kiện của Định lí 5-21 hoặc Định lí 5-23 thì $cl(G)$ là đồ thị đầy đủ, khi đó $cl(G)$ chắc chắn có chu trình Hamilton. Như vậy định lí Bondy-Chvátal là mở rộng của định lí Dirak và định lí Ore.

c) Cài đặt

Mặc dù chu trình Hamilton và chu trình Euler có tính đối ngẫu, người ta vẫn chưa tìm ra phương pháp với độ phức tạp đa thức để tìm chu trình Hamilton cũng như đường đi Hamilton trong trường hợp đồ thị tổng quát. Tất cả các thuật toán tìm chu trình Hamilton hiện nay đều dựa trên mô hình duyệt, có thể kết hợp với một số mẹo cài đặt (*heuristics*).

Chúng ta sẽ lập trình tìm một chu trình Hamilton (nếu có) trên một đơn đồ thị vô hướng với khuôn dạng Input/Output như sau:

Input

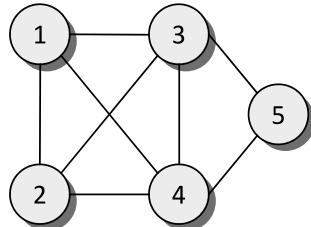
- Dòng 1 chứa số đỉnh n và số cạnh m của đơn đồ thị ($2 \leq n \leq 1000$)

- m dòng tiếp theo, mỗi dòng chứa hai số u, v tương ứng với một cạnh (u, v) của đồ thị

Output

Một chu trình Hamilton nếu có

Sample Input	Sample Output
5 8	1 2 3 5 4 1
1 2	
1 3	
1 4	
2 3	
2 4	
3 4	
3 5	
4 5	



Tìm chu trình Hamilton trên đồ thị vô hướng

```

{$MODE OBJFPC}
program HamiltonCycle;
const
    maxN = 1000;
var
    a: array[1..maxN, 1..maxN] of Boolean; //Ma trận kề
    avail: array[2..maxN] of Boolean;
    x: array[1..maxN] of Integer;
    Found: Boolean;
    n: Integer;
procedure Enter; //Nhập dữ liệu và khởi tạo
var
    m, i, u, v: Integer;
begin
    FillChar(a, SizeOf(a), False);
    ReadLn(n, m);
    for i := 1 to m do
        begin
            Read(u, v);
            a[u, v] := True;
            a[v, u] := True;
        end;
    FillChar(avail, SizeOf(avail), True); //Mọi đỉnh 2...n đều chưa đi qua
    Found := False; //Found = False: Chưa tìm ra nghiệm

```

```

x[1] := 1;
end;
procedure Attempt(i: Integer); //Thuật toán quay lui
var
  v: Integer;
begin
  for v := 2 to n do
    if avail[v] and a[x[i - 1], v] then //Xét các đỉnh v chưa đi qua kể
    với x[i - 1]
      begin
        x[i] := v; //Thử đi sang v
        if i = n then //Nếu đã qua đủ n đỉnh, đến đỉnh thứ n
          begin
            if a[v, 1] then Found := True; //Đỉnh thứ n quay về được
            i thì tìm ra nghiệm
            Exit; //Thoát luôn
          end
        else //Qua chưa đủ n đỉnh
          begin
            avail[v] := False; //Dánh dấu đỉnh đã qua
            Attempt(i + 1); //Di tiếp
            if Found then Exit; //Nếu đã tìm ra nghiệm thì thoát ngay
            avail[v] := True;
          end;
      end;
  end;
procedure PrintResult; //In kết quả
var
  i: Integer;
begin
  if not Found then
    WriteLn('There is no Hamilton cycle')
  else
    begin
      for i := 1 to n do
        Write(x[i], ' ');
      WriteLn();
    end;
  end;
begin
  Enter;

```

```

Attempt(2) ;
PrintResult;
end.

```

6.3. Hai bài toán nổi tiếng ★

a) Bài toán người đưa thư Trung Hoa

Bài toán người đưa thư Trung Hoa (Chinese Postman) được phát biểu đầu tiên dưới dạng tìm hành trình tối ưu cho người đưa thư: Anh ta phải đi qua tất cả các quãng đường để chuyển phát thư tín và mong muốn tìm hành trình ngắn nhất để đi hết các quãng đường trong khu vực mà anh ta phụ trách. Chúng ta có thể phát biểu trên mô hình đồ thị như sau:

Bài toán: Cho đồ thị $G = (V, E)$, mỗi cạnh $e \in E$ có độ dài (trọng số) $c(e)$. Hãy tìm một chu trình đi qua tất cả các cạnh, mỗi cạnh ít nhất một lần sao cho tổng độ dài các cạnh đi qua là nhỏ nhất.

Đi nhiên nếu G là đồ thị Euler thì lời giải chính là chu trình Euler, nhưng nếu G không phải đồ thị Euler thì sao?. Người ta đã có thuật toán với độ phức tạp đa thức để giải bài toán người đưa thư Trung Hoa nếu G là đồ thị vô hướng hoặc có hướng. Một trong những thuật toán đó là kết hợp thuật toán tìm chu trình Euler với một thuật toán tìm bộ ghép cực đại trên đồ thị. Tuy nhiên nếu G là đồ thị hỗn hợp (có cả cung có hướng và cạnh vô hướng) thì bài toán người đưa thư Trung Hoa là bài toán NP-đầy đủ, trong trường hợp này, việc chỉ ra một thuật toán đa thức cũng như việc chứng minh không tồn tại thuật toán đa thức để giải quyết hiện vẫn đang là thách thức của ngành khoa học máy tính.

Thật đáng tiếc, sơ đồ giao thông của hầu hết các thành phố trên thế giới đều ở dạng đồ thị hỗn hợp (có cả đường hai chiều và đường một chiều) và như vậy chưa thể có một thuật toán đa thức tối ưu dành cho các nhân viên bưu chính.

b) Bài toán người du lịch

Bài toán người du lịch (Travelling Salesman) đặt ra là có n thành phố và chi phí di chuyển giữa hai thành phố bất kì trong n thành phố đó. Một người muốn đi du lịch qua tất cả các thành phố, mỗi thành phố ít nhất một lần và quay về thành phố xuất phát, sao cho tổng chi phí di chuyển là nhỏ nhất có thể. Chúng ta có thể phát biểu bài toán này trên mô hình đồ thị như sau:

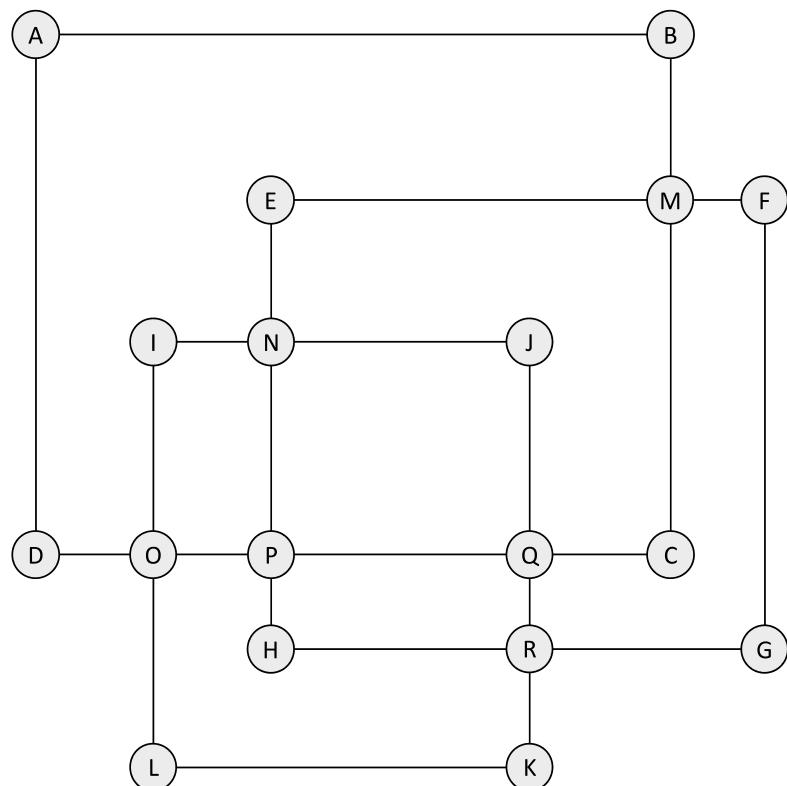
Bài toán: Cho đồ thị $G = (V, E)$, mỗi cạnh $e \in E$ có độ dài (trọng số) $c(e)$. Hãy tìm một chu trình đi qua tất cả các đỉnh, mỗi đỉnh ít nhất một lần sao cho tổng độ dài các cạnh đi qua là nhỏ nhất.

Thực ra yêu cầu đi qua mỗi đỉnh ít nhất một lần hay đi qua mỗi đỉnh đúng một lần đều khó như nhau cả. Bài toán người du lịch là NP-đầy đủ, hiện tại chưa có thuật toán đa thức để giải quyết, chỉ có một số thuật toán xấp xỉ hoặc phương pháp duyệt nhánh cạn mà thôi.

Bài tập

- 5.22.** Trên mặt phẳng cho n hình chữ nhật có các cạnh song song với các trục tọa độ. Hãy chỉ ra một chu trình:

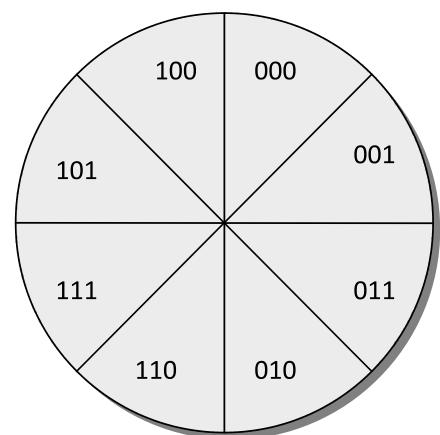
- Chỉ đi trên cạnh của các hình chữ nhật
- Trên cạnh của mỗi hình chữ nhật, ngoại trừ những giao điểm với cạnh của hình chữ nhật khác có thể qua nhiều lần, những điểm còn lại chỉ được qua đúng một lần.



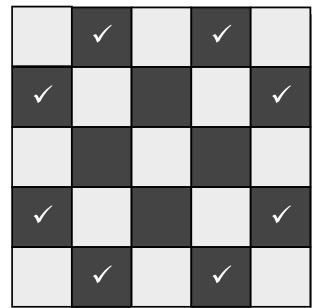
ABMFGRHPNEMCQRKLOINJQPODA

- 5.23.** Trong đám cưới của Persée và Andromède có $2n$ hiệp sĩ. Mỗi hiệp sĩ có không quá $n - 1$ kẻ thù. Hãy giúp Cassiopé, mẹ của Andromède xếp $2n$ hiệp sĩ ngồi quanh một bàn tròn sao cho không có hiệp sĩ nào phải ngồi cạnh kẻ thù của mình. Mỗi hiệp sĩ sẽ cho biết những kẻ thù của mình khi họ đến sân rồng.

- 5.24.** Gray code: Một hình tròn được chia thành 2^n hình quạt đồng tâm. Hãy xếp tất cả các xâu nhị phân độ dài n vào các hình quạt, mỗi xâu vào một hình quạt sao cho bất cứ hai xâu nào ở hai hình quạt cạnh nhau đều chỉ khác nhau đúng 1 bit. Ví dụ với $n = 3$:



- 5.25.** Bài toán mă đi tuần: Trên bàn cờ tổng quát kích thước $m \times n$ ô vuông ($5 \leq m, n \leq 1000$). Một quân mă đang ở ô (x_1, y_1) có thể di chuyển sang ô (x_2, y_2) nếu $|x_1 - x_2| \cdot |y_1 - y_2| = 2$ (Xem hình vẽ).



Hãy tìm hành trình của quân mă từ ô xuất phát từ một ô tùy chọn, đi qua tất cả các ô của bàn cờ, mỗi ô đúng 1 lần.

Ví dụ với $n = 8$

Hướng dẫn: Nếu coi các ô của bàn cờ là các đỉnh của đồ thị và các cạnh là nối giữa hai đỉnh tương ứng với hai ô mă giao chân thì dễ thấy rằng hành trình của quân mă cần tìm sẽ là một đường đi Hamilton. Tuy vậy thuật toán duyệt thuần túy là bất khả thi với dữ liệu lớn, bạn có thể thử cài đặt và ngồi xem máy tính vẫn toát mồ hôi ☺.

Để giải quyết bài toán mă đi tuần, có một mẹo nhỏ được Warnsdorff đưa ra cách đây gần 2 thế kỉ (1823). Mẹo này không chỉ áp dụng được vào bài toán mă đi tuần mà còn có thể kết hợp vào thuật toán duyệt để tìm đường đi Hamilton trên đồ thị bất kì nếu biết chắc đường đi đó tồn tại (duyệt tham phối hợp).

Với mỗi ô (x, y) ta gọi bậc của ô đó, $\deg(x, y)$, là số ô kề với ô (x, y) chưa được thăm (kề ở đây theo nghĩa đỉnh kề chứ không phải là ô kề cạnh). Đặt ngẫu nhiên quân mă vào ô (x, y) nào đó và cứ di chuyển quân mă sang ô kề có bậc nhỏ nhất. Nếu đi được hết bàn cờ thì xong, nếu không ta đặt ngẫu nhiên quân mă vào một ô xuất phát khác và làm lại.

Thuật toán này đã được thử nghiệm và nhận thấy rằng việc tìm ra một bộ $m, n: 5 \leq m, n \leq 1000$ để chương trình chạy > 10 giây cũng là một chuyện...bất khả thi.

15	26	39	58	17	28	37	50
40	59	16	27	38	51	18	29
25	14	47	52	57	30	49	36
46	41	60	31	48	53	56	19
13	24	45	62	1	20	35	54
42	61	10	23	32	55	2	5
9	12	63	44	7	4	21	34
64	43	8	11	22	33	6	3