Chương 21. Các khái niệm cơ bản

21.1. Đồ thị

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

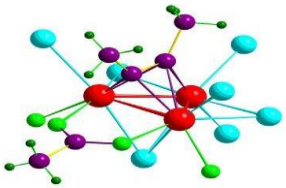
$$Graph = Objects + Connections$$
$$G = (V, E)$$

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

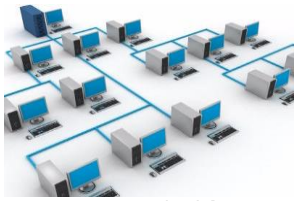
Ta cho phép đồ thị có cả cạnh nối từ một đỉnh tới chính nó. Những cạnh như vậy gọi là *khuyên* (*self-loop*)

Một số hình ảnh của đồ thị:

Sơ đồ giao thông



Cấu trúc phân tử



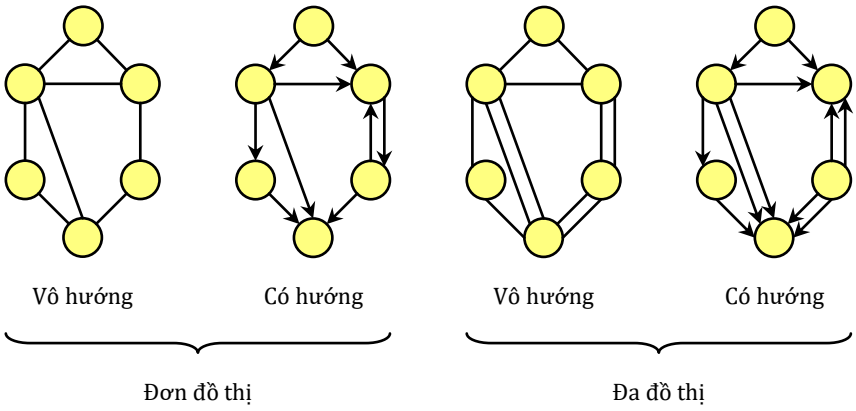
Mạng máy tính

Hình 21-1. Một số hình ảnh của đồ thị

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

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

Hình 22-4 là ví dụ về đơn đồ thị/đa đồ thị có hướng/vô hướng.



Hình 21-2. Phân loại đồ thị

21.2. Các khái niệm

21.2.1. Cạnh liên thuộc, đỉnh kề, bậc

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

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

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

Định lý 21-1

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

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

Chứng minh

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

Hệ quả

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

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

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

Định lý 21-2

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

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

Chứng minh

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

21.2.2. Đường đi và chu trình

Một dãy các đỉnh:

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

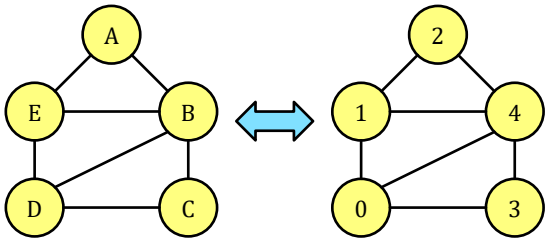
sao cho $(p_{i-1}, p_i) \in E, \forall i: 1 \le i \le k$ được gọi là một *đường đi* (*path*), đường đi này gồm $k + 1$ đỉnh $p_0, p_1, \dots, p_k$ và k cạnh $(p_0, p_1), (p_1, p_2), \dots, (p_{k-1}, p_k)$. Nếu có một đường đi như trên thì ta nói p_k *đến được* (*reachable*) từ p_0 hay p_0 đến được p_k, ký hiệu $p_0 \rightsquigarrow p_k$. 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 (dĩ 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 $k \ge 3$ và các đỉnh $p_1, p_2, \dots, p_k$ hoàn toàn phân biệt*.

21.2.3. 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 $f: V \to V'$ sao cho số cung nối u với v trên E bằng số cung nối $f(u)$ với $f(v)$ trên E'. Nói cách khác, ta có thể "đặt tên" lại các đỉnh trong G để thu được đồ thị G'.



$f: \{A, B, C, D, E\} \longrightarrow \{0,1,2,3,4\}$
$A \longmapsto 2$
$B \longmapsto 4$
$C \longmapsto 3$
$D \longmapsto 0$
$E \longmapsto 1$

Hình 21-3. Đẳng cấu

❋ Đồ thị con

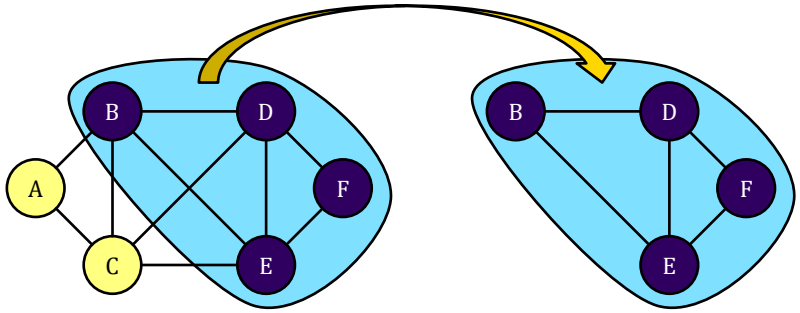
Đồ thị $G' = (V', E')$ là *đồ thị con* (*subgraph*) của đồ thị $G = (V, E)$ nếu $V' \subseteq V$ và $E' \subseteq E$.

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

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

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

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



Hình 21-4. Đồ thị con cảm ứng

❋ Phiên bản có hướng/vô hướng

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

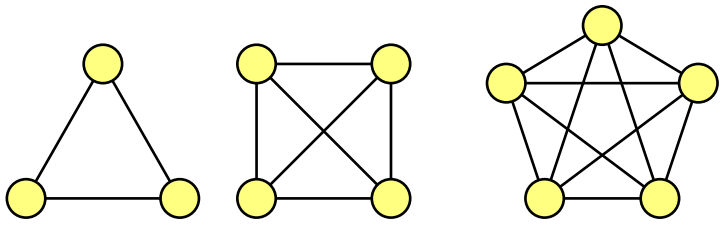
Với một đồ thị có hướng $G = (V, E)$, ta gọi *phiên bản vô hướng* (*undirected version*) của G là một đồ thị vô hướng $G' = (V, E')$ tạo thành bằng cách thay mỗi cung (u, v) bằng cạnh vô hướng (u, v). Nói cách khác, G' tạo thành từ G bằng cách bỏ đi chiều của cung.

❋ Tính liên thông

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

❋ Đồ thị đầy đủ

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

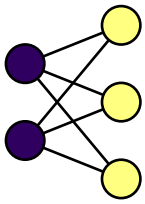


Hình 21-5. Đồ thị đầy đủ

❋ Đồ thị hai phía

Một đồ thị vô hướng gọi là *hai phía* (*bipartite*) nếu tập đỉnh của nó có thể chia làm hai tập rời nhau X, Y sao cho không tồn tại cạnh nối hai đỉnh thuộc X cũng như không tồn tại cạnh

nối hai đỉnh thuộc Y. Nếu $|X| = m$ và $|Y| = n$ và giữa mọi cặp đỉnh (x, y) trong đó $x \in X, y \in Y$ đều có cạnh nối thì đồ thị hai phía đó được gọi là đồ thị hai phía đầy đủ, ký hiệu $K_{m,n}$. Hình 22-4 là ví dụ về đồ thị hai phía đầy đủ $K_{2,3}$.



Hình 21-6. Đồ thị hai phía đầy đủ

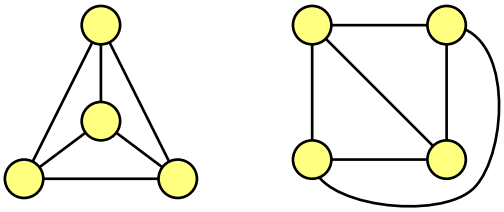
❋ Đồ thị phẳng

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

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

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

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



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

Định lý 21-3 (Định lý Kuratowski)

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

Định lý 21-4 (Công thức Euler)

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

Định lý 21-5

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

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

❋ Đồ thị đường

Từ đồ thị vô hướng G, ta xây dựng đồ thị vô hướng G' như sau: Mỗi đỉnh của G' tương ứng với một cạnh của G, giữa hai đỉnh x, y của G' có cạnh nối nếu và chỉ nếu tồn tại đỉnh liên thuộc với cả hai cạnh x, y trên G. Đồ thị G' như vậy được gọi là đồ thị đường của đồ thị G.

Đồ thị đường được nghiên cứu trong các bài toán kiểm tra tính liên thông, tập độc lập cực đại, tô màu cạnh đồ thị, chu trình Euler và chu trình Hamilton v.v…

Chương 22. Biểu diễn đồ thị

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

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

22.1. Ma trận kề

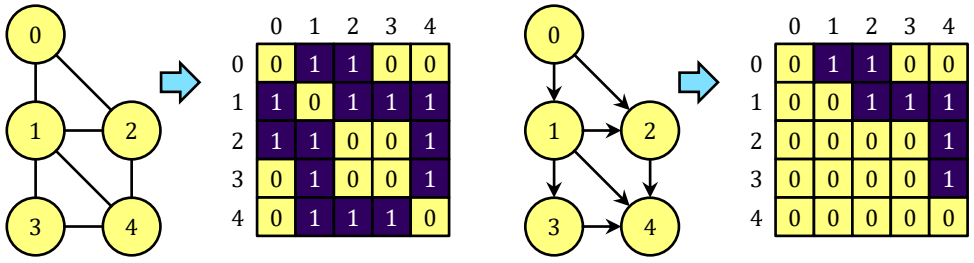
Với $G = (V, E)$ là một đơn đồ thị có hướng trong đó $|V| = n$, ta có thể đánh số các đỉnh từ 0 tới $n - 1$ và đồng nhất mỗi đỉnh với số thứ tự của nó. Bằng cách đánh số như vậy, đồ thị G có thể biểu diễn bằng ma trận vuông $A = \{a_{ij}\}_{n \times n}$ với các hàng và các cột đánh số từ 0 tới $n - 1$.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

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

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

Với $\forall i$, giá trị của các phần tử trên đường chéo chính ma trận A: $\{a_{ii}\}$ có thể đặt tuỳ 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 22-1. Ma trận kề biểu diễn đồ thị

Ma trận kề có một số tính chất:

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

Ưu điểm của ma trận kề:

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

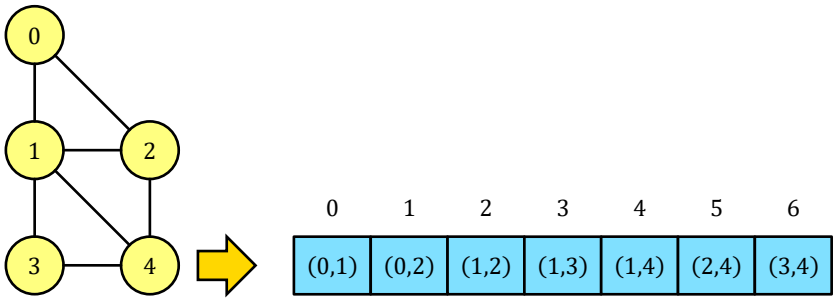
Nhược điểm của ma trận kề

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

22.2. Danh sách cạnh

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

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



Hình 22-2. Danh sách cạnh

Ưu điểm của danh sách cạnh:

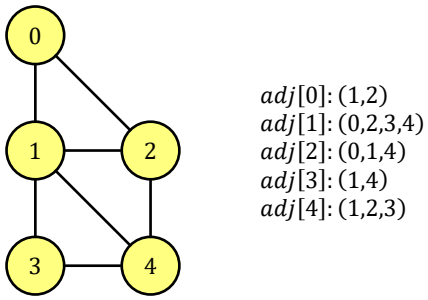
- Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ, bởi nó chỉ cần $O(m)$ ô nhớ để lưu danh sách cạnh.
- Trong một số trường hợp, ta phải xét tất cả các cạnh của đồ thị thì cài đặt trên danh sách cạnh làm cho việc duyệt các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

Nhược điểm của danh sách cạnh:

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

22.3. Danh sách kề

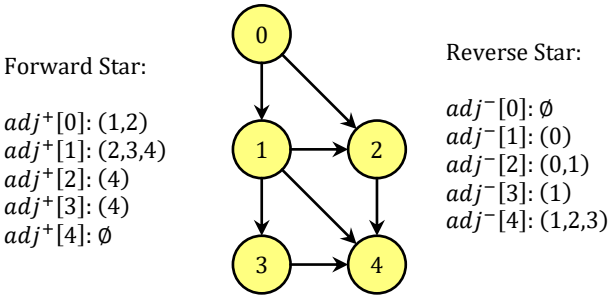
Để khắc phục nhược điểm của các phương pháp ma trận kề và danh sách cạnh, người ta đề xuất phương pháp biểu diễn đồ thị bằng *danh sách kề* (*adjacency list*). Trong cách biểu diễn này, với mỗi đỉnh u của đồ thị vô hướng, ta cho tương ứng với nó một danh sách $adj[u]$ gồm các đỉnh kề với u (Hình 22-4).



Hình 22-3. Danh sách kề

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

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



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

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

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

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

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

22.4. Danh sách liên thuộc

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

22.5. Chuyển đổi giữa các cách biểu diễn đồ thị

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

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

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

Bài tập 22-1

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

Bài tập 22-2

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

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

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

Bài tập 22-3

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

Bài tập 22-4

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

Gợi ý: Sử dụng chứng minh quy nạp.

Bài tập 22-5

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

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

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

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

Bài tập 22-6

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

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

Bài tập 22-7

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

Bài tập 22-8

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

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

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

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

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

Bài tập 22-9

Xét đồ thị vô hướng $G = (V, E)$ với ma trận liên thuộc B định nghĩa như trong Bài tập 22-8. Xét đồ thị đường $L(G)$, trong đó mỗi đỉnh của $L(G)$ ứng với một cạnh của G. Hai đỉnh i, j

của $L(G)$ kề nhau nếu tồn tại một đỉnh u của G liên thuộc với cả hai cạnh i, j. Gọi A là ma trận kề của đồ thị $L(G)$ xác định bởi

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

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

Chương 23. Các thuật toán tìm kiếm trên đồ thị

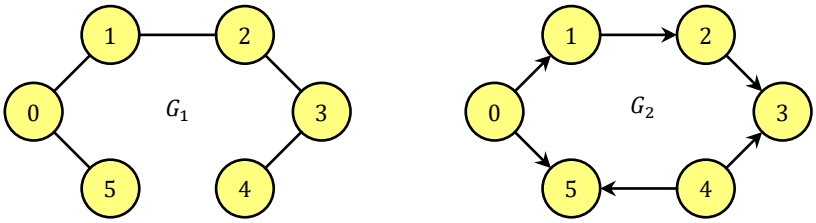
23.1. Bài toán tìm đường

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

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

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

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



Hình 23-1. Đồ thị và đường đi

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

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

Khuôn dạng Input/Output quy định như sau:

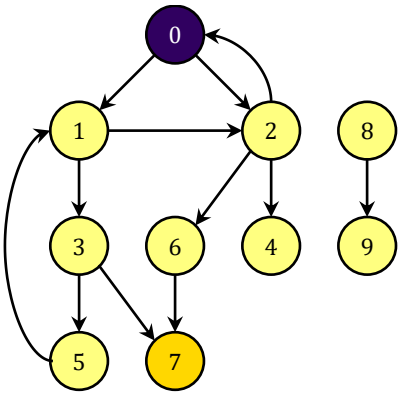
Input

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

Output

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

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



23.2. Biểu diễn đồ thị

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

23.3. Thuật toán tìm kiếm theo chiều sâu

23.3.1. Ý tưởng

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

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

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

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

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

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

23.3.2. Cài đặt

DFS.CPP ✓ Tìm đường bằng DFS

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

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

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

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

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

```
    }
}

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

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

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

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

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

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

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

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

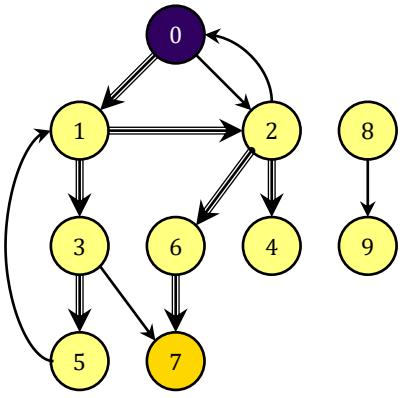
Cũng có thể thay danh sách kề dạng forward star bằng danh sách kề dạng reverse star rồi thực hiện thuật toán tìm kiếm theo chiều sâu bắt đầu từ t, sau đó truy vết từ s.

23.3.3. Một vài tính chất của DFS

❋ Cây DFS

Nếu ta sắp xếp danh sách kề của mỗi đỉnh theo thứ tự tăng dần thì thuật toán DFS luôn trả về đường đi có thứ tự từ điển nhỏ nhất trong số tất cả các đường đi từ s tới 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 bên là đồ thị với danh sách kề của mỗi đỉnh được sắp xếp tăng dần và cây DFS tương ứng với đỉnh xuất phát $s = 0$. Chú ý rằng cấu trúc của cây DFS phụ thuộc vào thứ tự trong các danh sách kề.



❋ Mô hình duyệt đồ thị theo DFS

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

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

Input → đồ thị G;
for (∀u ∈ V) d[u] = 0; //Các đỉnh đều chưa thăm: d[.] = 0
Time = 0;
for (∀u ∈ V) //Xét mọi điểm chưa thăm
    if (d[u] == 0) DFSVisit(u);
```

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

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

❋ Thứ tự thăm đến và duyệt xong

Hãy để ý hàm $DFSVisit(u)$:

- Khi bắt đầu vào hàm ta nói đỉnh u được *thăm đến* (*discover*), có nghĩa là tại thời điểm đó, quá trình tìm kiếm theo chiều sâu bắt đầu từ u sẽ xây dựng nhánh cây DFS gốc u.
- Khi chuẩn bị thoát khỏi hàm để lùi về , ta nói đỉnh u được *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, một biến đếm $Time$ được dùng để xác định thời điểm thăm đến d_u và thời điểm duyệt xong f_u của mỗi đỉnh u. Thứ tự thăm đến và duyệt xong này có ý nghĩa rất quan trọng trong nhiều thuật toán có áp dụng DFS, chẳng hạn như các thuật toán tìm thành phần liên thông mạnh, thuật toán sắp xếp tô pô…

Định lý 23-1

Với hai đỉnh phân biệt u, v:

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

Chứng minh

Bản chất của việc đỉnh v được thăm đến (hay duyệt xong) trong thời gian từ d_u đến f_u chính là hàm $DFSVisit(v)$ được gọi (hay thoát) khi mà hàm $DFSVisit(u)$ đã bắt đầu nhưng chưa kết thúc, nghĩa là hàm $DFSVisit(v)$ được dây chuyền đệ quy từ $DFSVisit(u)$ gọi tới. Điều này chỉ ra rằng v nằm trong nhánh DFS gốc u, hay nói cách khác, v là hậu duệ của u.

Hệ quả

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

Chứng minh

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

Định lý 23-2

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

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

Chứng minh

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

Định lý 23-3 (định lý đường đi trắng)

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

Chứng minh

"$\Rightarrow$"

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

"$\Leftarrow$"

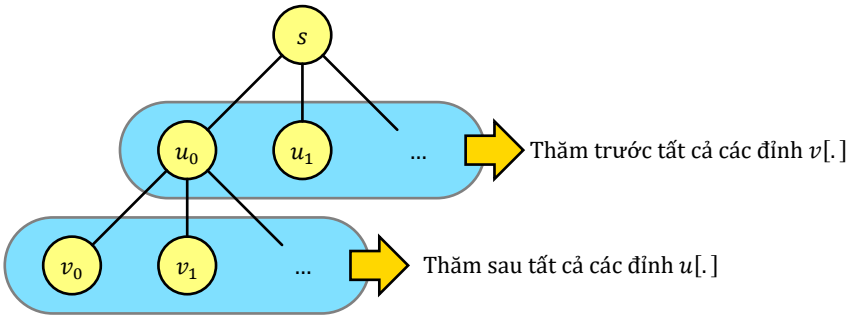
Nếu tại thời điểm d_u, tồn tại một đường đi từ u tới v mà tất cả các đỉnh khác u trên đường đi đều chưa được thăm, ta sẽ chứng minh rằng mọi đỉnh trên đường đi này đều là hậu duệ của u. Thật vậy, giả sử phản chứng rằng y là đỉnh đầu tiên trên đường đi này mà không phải hậu duệ của u, tức là tồn tại đỉnh x liền trước y trên đường đi là hậu duệ của u. Theo Định lý 23-2, y phải được thăm trước khi duyệt xong x: $d_y < f_x$; x lại là hậu duệ của u nên theo Định lý 23-1, ta có $f_x \leq f_u$, vậy $d_y < f_u$. Mặt khác theo giả thiết rằng tại thời điểm d_u thì y chưa được thăm, tức là $d_u < d_y$, kết hợp lại ta có $d_u < d_y < f_u$, vậy thì y là hậu duệ của u theo Định lý 23-1, trái với giả thiết phản chứng.

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

23.4. Thuật toán tìm kiếm theo chiều rộng

23.4.1. Ý tưởng

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



Hình 23-2. Thứ tự thăm đỉnh của BFS

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

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

23.4.2. Cài đặt

BFS.CPP ✓ Tìm đường bằng BFS

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

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

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

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

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

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

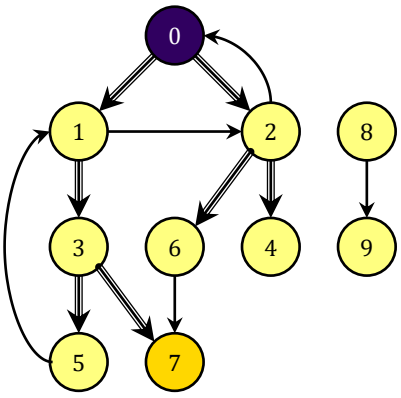
Tương tự như thuật toán tìm kiếm theo chiều sâu, ta có thể dùng mảng $trace[0 \dots n)$ kiêm luôn chức năng đánh dấu.

23.4.3. Một vài tính chất của BFS

❋ Cây BFS

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

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



❋ Mô hình duyệt đồ thị theo BFS

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

```
void BFSVisit(s ∈ V)
{
    Queue = (s); //queue chỉ gồm một đỉnh s
    d[s] = ++Time; //d[u] = thời điểm u được thăm đến, cũng là đánh dấu ≠ 0
    do
    {
        u = Queue.front(); Queue.pop();
        f[u] = ++Time;
        Output ← u;
        for (int v: adj[u]) //duyệt mọi đỉnh v nối từ u
            if (d[v] == 0) //nếu v chưa thăm
            {
                Queue.push(v);
                d[v] = ++Time; //Ghi nhận thời điểm thăm đến u, cũng là đánh dấu d[v] ≠ 0
            }
    }
    while (Queue ≠ ∅);
}

Input → đồ thị G;
for (∀u ∈ V) d[u] = 0; //Các đỉnh đều chưa thăm: d[.] = 0
Time = 0;
for (∀u ∈ V) //Xét mọi điểm chưa thăm
    if (d[u] == 0) BFSVisit(u);
```

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

❋ Thứ tự thăm đến và duyệt xong

Tương tự như thuật toán DFS, đối với thuật toán BFS người ta cũng quan tâm tới thứ tự thăm đến và duyệt xong: Khi một đỉnh được đẩy vào hàng đợi, ta nói đỉnh đó được thăm đến (được thăm) và khi một đỉnh được lấy ra khỏi hàng đợi, ta nói đỉnh đó được duyệt xong. Trong mô hình cài đặt trên, mỗi đỉnh u sẽ tương ứng với thời điểm thăm đến d_u và thời điểm duyệt xong f_u

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

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

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

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

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

Bài tập 23-1

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

Bài tập 23-2

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

Bài tập 23-3

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

Bài tập 23-4

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

Bài tập 23-5

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

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

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

Chương 24. Tính liên thông của đồ thị

24.1. Định nghĩa

24.1.1. Tính liên thông trên đồ thị vô hướng

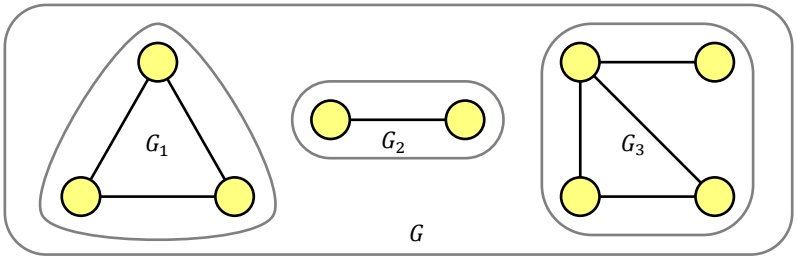
Đồ thị vô hướng $G = (V, E)$ được gọi là *liên thông* (*connected*) nếu giữa mọi cặp đỉnh của G luôn tồn tại đường đi. Đồ thị chỉ gồm một đỉnh duy nhất cũng được coi là đồ thị liên thông.

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

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

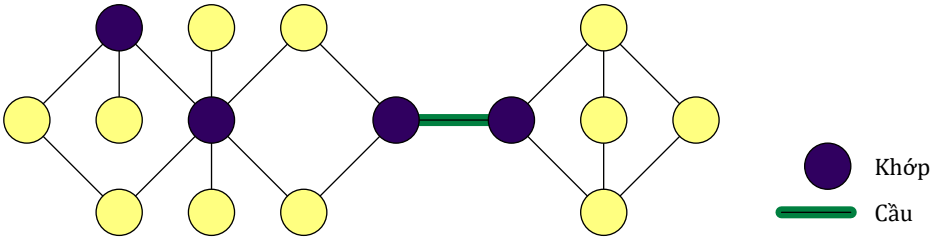
(Ta cũng đồng nhất khái niệm thành phần liên thông U với thành phần liên thông $G_U = (U, E_U)$).

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



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

Đôi khi, việc 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 24-2. Khớp và cầu

24.1.2. Tính liên thông trên đồ thị có hướng

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

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

Liên thông mạnh Liên thông yếu

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

24.2. Bài toán xác định các thành phần liên thông

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

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

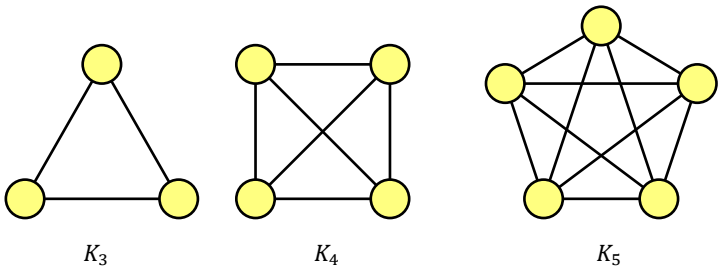
```
void Scan(u ∈ V)
{
    «Dùng BFS hoặc DFS liệt kê và đánh dấu thăm những đỉnh có thể đến được từ u»;
}

for (∀u ∈ V)
    «Đánh dấu v chưa thăm»;
for (∀u∈V)
    if «u chưa thăm» then
    {
        Output ← «Các đỉnh của một thành phần liên thông:»;
        Scan(u);
    }
```

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

24.3. Bao đóng của đồ thị vô hướng

24.3.1. Định nghĩa

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



Hình 24-4. Đồ thị đầy đủ

24.3.2. Bao đóng đồ thị

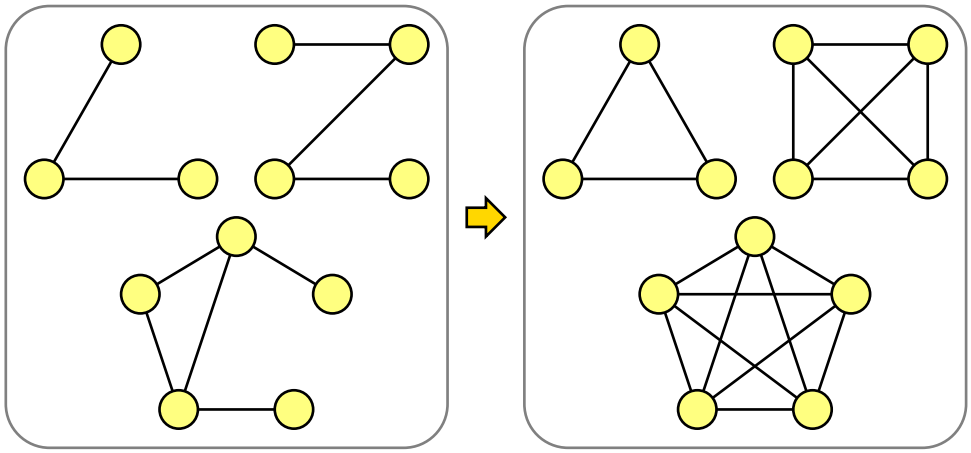
Với đồ thị $G = (V, E)$, người ta xây dựng đồ thị $\overline{G} = (V, \overline{E})$ cũng gồm những đỉnh của G còn các cạnh xây dựng như sau:

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

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

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

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



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

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

24.3.3. Thuật toán Warshall

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

Giả sử đơn đồ thị vô hướng $G = (V, E)$ có n đỉnh đánh số từ 0 tới $n - 1$, thuật toán Warshall xét tất cả các đỉnh $k \in V$, với mỗi đỉnh k được xét, thuật toán lại xét tiếp tất cả

các cặp đỉnh (i, j): nếu đồ thị có cạnh (i, k) và cạnh (k, j) thì ta tự nối thêm cạnh (i, j) nếu nó chưa có. Tư tưởng này dựa trên một quan sát đơn giản như sau: Nếu từ i có đường đi tới k và từ k lại có đường đi tới j thì chắc chắn từ i sẽ có đường đi tới j.

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

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

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

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

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

- Thuật toán Warshall hoàn toàn có thể áp dụng để tính bao đóng $\overline{G} = \left(V, \overline{E}\right)$ của đồ thị có hướng $G = (V, E)$ với định nghĩa bao đóng tương tự như trên đồ thị vô hướng: $(u, v) \in \overline{E} \Leftrightarrow u \rightsquigarrow v$ trên G
- Việc đảo lộn thứ tự ba vòng lặp for có thể cho một thuật toán sai. Thực ra có thể đảo thứ tự hai vòng lặp "for (i…" và "for (j…" nhưng vòng lặp "for (k…" chắc chắn phải nằm ở ngoài cùng.
- Tuy thuật toán Warshall rất dễ cài đặt nhưng đòi hỏi thời gian thực hiện giải thuật khá lớn: $\Theta(n^3)$. Chính vì vậy thuật toán Warshall chỉ nên sử dụng khi thực sự cần tới bao đóng của đồ thị, còn nếu chỉ cần liệt kê các thành phần liên thông thì các thuật toán tìm kiếm trên đồ thị tỏ ra hiệu quả hơn nhiều.

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

Việc cài đặt thuật toán sẽ qua những bước sau:

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

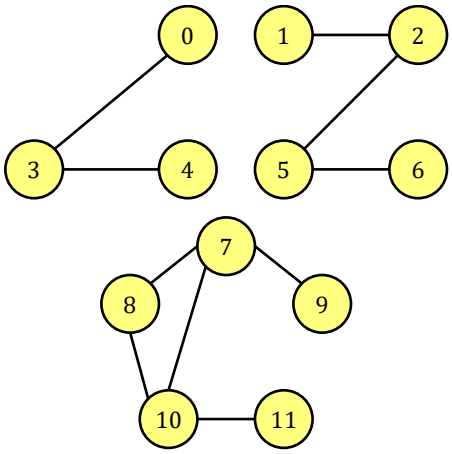
Input

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

Output

Liệt kê các thành phần liên thông của đồ thị

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



WARSHALL.CPP ✓ Thuật toán Warshall

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

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

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

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

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

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

Đối với đồ thị có hướng, người ta quan tâm đến bài toán kiểm tra tính liên thông mạnh, hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông mạnh của đồ thị có hướng. Các thuật toán tìm kiếm thành phần liên thông mạnh hiệu quả hiện nay đều dựa trên thuật toán tìm kiếm theo chiều sâu Depth-First Search.

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

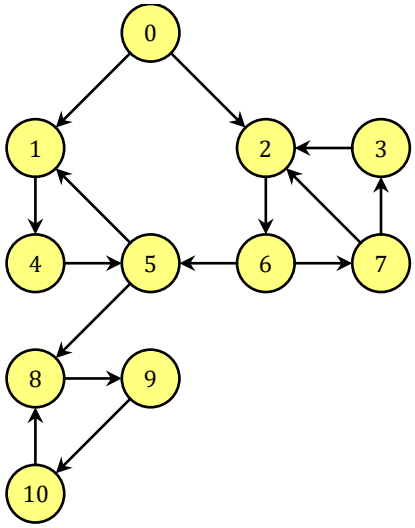
Input

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

Output

Các thành phần liên thông mạnh.

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



24.4.1. Phân tích

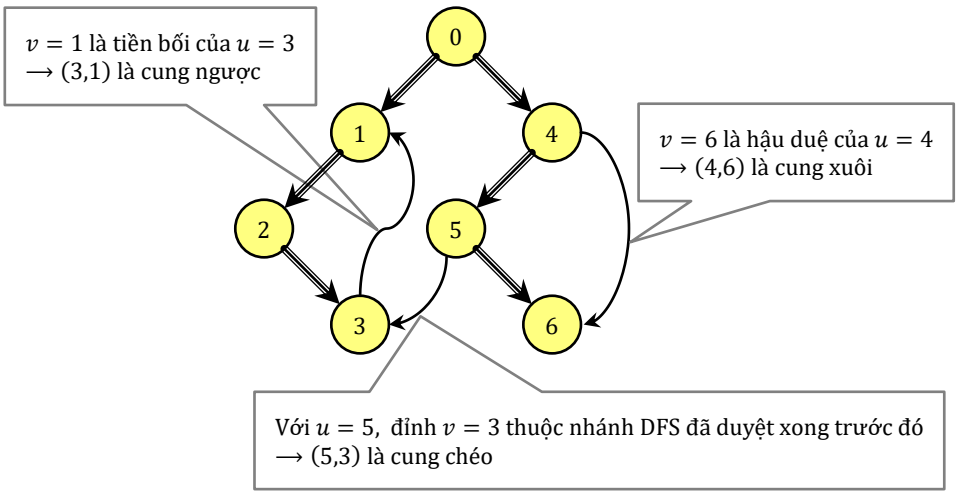
Xét thuật toán tìm kiếm theo chiều sâu:

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

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

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

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

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



Hình 24-6. Ba loại cung ngoài cây DFS

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

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

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

24.4.2. Cây DFS và các thành phần liên thông mạnh

Bổ đề 24-1

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

Chứng minh

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

Bổ đề 24-2

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

Chứng minh

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

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

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

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

Bổ đề 24-3

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

Chứng minh:

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

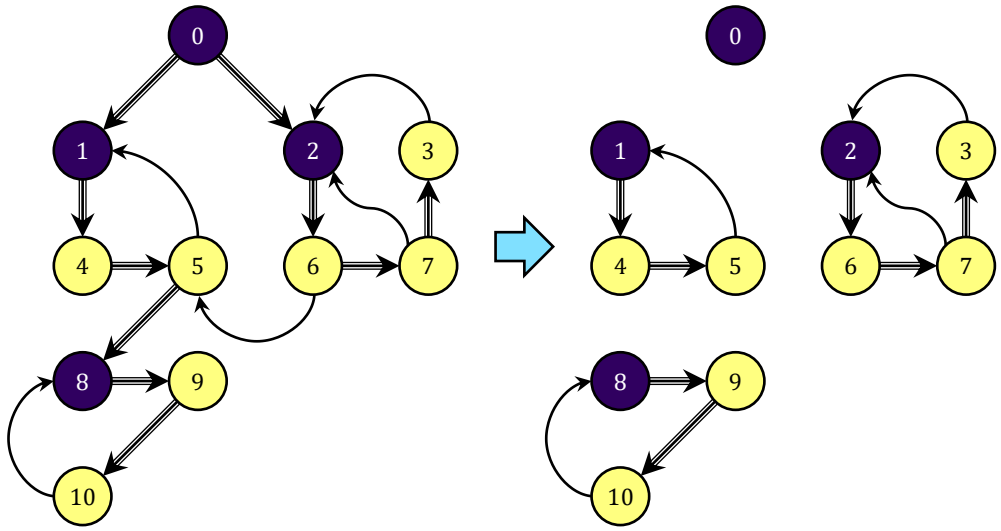
Bổ đề 24-2 khẳng định thành phần liên thông mạnh chứa r nằm trong nhánh DFS gốc r, theo chứng minh trên ta lại có: Mọi đỉnh trong nhánh DFS gốc r nằm trong thành phần liên

thông mạnh chứa r. Kết hợp lại được: Nhánh DFS gốc r chính là thành phần liên thông mạnh chứa r.

24.4.3. Thuật toán Tarjan

❋ Ý tưởng

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



Hình 24-7. Thuật toán Tarjan "bẻ" cây DFS

Mô hình cài đặt của thuật toán Tarjan:

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

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

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

Định lý 24-4

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

Chứng minh

Ta nhắc lại các tính chất của 4 loại cung:

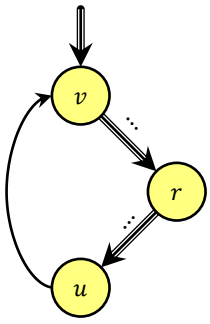
- Cung DFS và cung xuôi nối từ đỉnh thăm trước đến đỉnh thăm sau, hơn nữa chúng đều là cung nối từ tiền bối tới hậu duệ
- Cung ngược và cung chéo nối từ đỉnh thăm sau tới đỉnh thăm trước, cung ngược nối từ hậu duệ tới tiền bối còn cung chéo nối hai đỉnh không có quan hệ tiền bối–hậu duệ.

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

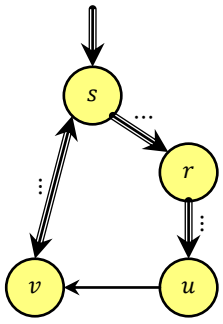
Ngược lại, nếu từ đỉnh u của nhánh DFS gốc r có cung (u, v) tới đỉnh v thăm trước r, ta chứng minh rằng r không thể là chốt.

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

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



TH (u, v) là cung ngược



TH (u, v) là cung chéo

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

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

Suy ra khi hàm DFSVisit(u) được gọi, hai hàm 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 lý 24-4, việc sẽ kiểm tra đỉnh r có là chốt hay không có thể thay bằng việc *kiểm tra xem có tồn tại cung nối từ một đỉnh thuộc nhánh DFS gốc r tới một đỉnh thăm trước r hay không?*.

Dưới đây là một cách cài đặt rất thông minh, nội dung của nó là đánh số thứ tự các đỉnh theo thứ tự thăm đến. Định nghĩa $num[u]$ là số thứ tự của đỉnh u theo cách đánh số đó. Ta tính thêm $low[u]$ là giá trị $num[.]$ 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 hàm $DFSVisit(u)$, trước hết ta đánh số thứ tự thăm cho đỉnh u: $num[u]$ 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:
$$low[u]_{\text{mới}} = \min(low[u]_{\text{cũ}}, num[v])$$
- Nếu v chưa thăm thì ta gọi đệ quy $DFSVisit(v)$ để xây dựng nhánh DFS gốc v (con của u), cũng là để tính giá trị $low[v]$ (bằng $num[.]$ nhỏ nhất của các đỉnh có thể đến được từ nhánh DFS gốc v), sau đó cực tiểu hoá $low[u]$ theo công thức:
$$low[u]_{\text{mới}} = \min(low[u]_{\text{cũ}}, low[v])$$

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

Để công việc dễ dàng hơn nữa, ta sử dụng một stack để chứa các đỉnh thuộc một nhánh DFS: Khi thăm đến một đỉnh u, ta đẩy ngay đỉnh u đó vào stack, thì khi duyệt xong đỉnh u, mọi đỉnh thuộc nhánh DFS gốc u sẽ được đẩy vào stack ngay sau u. Nếu u là chốt, ta chỉ việc lấy các đỉnh ra khỏi 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 chi tiết của thuật toán Tarjan

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

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

Bởi thuật toán Tarjan chỉ là sửa đổi của thuật toán DFS, các phép vào/ra stack được thực hiện không quá n lần. Vậy nên thời gian thực hiện giải thuật vẫn là $\Theta(|V| + |E|)$ trong trường hợp đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, là $\Theta(|V|^2)$ nếu dùng ma trận kề và là $\Theta(|V||E|)$ nếu dùng danh sách cạnh.

❋ Cài đặt

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

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

TARJAN.CPP ✓ Thuật toán Tarjan

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

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

```
}

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

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

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

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

24.4.4. Thuật toán Kosaraju-Sharir

❋ Ý tưởng của thuật toán

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

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

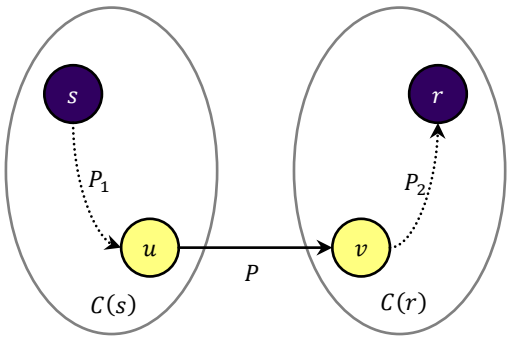
Bổ đề 24-5

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

Chứng minh

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

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



Hình 24-8

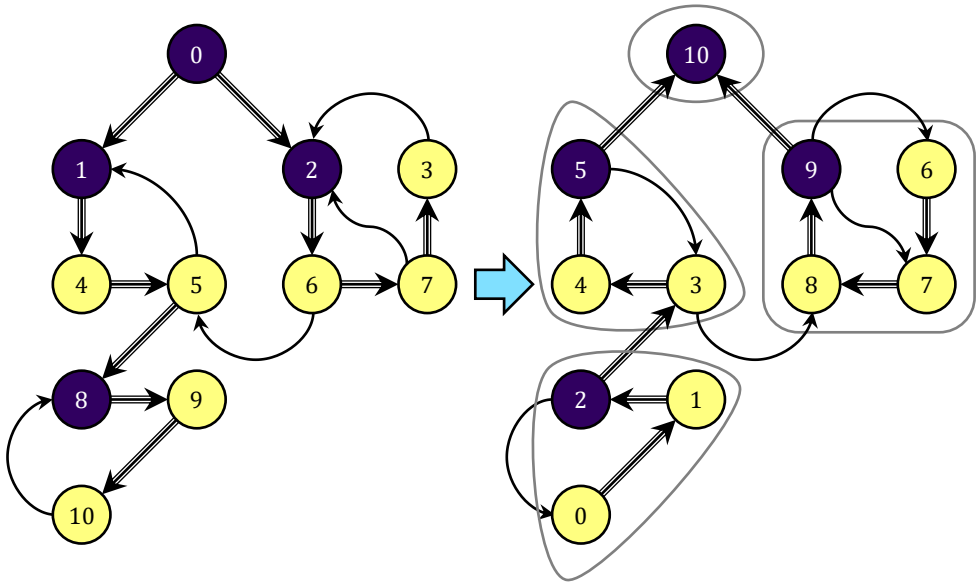
Theo tính chất của chốt là đỉnh thăm trước mọi đỉnh khác thuộc cùng thành phần liên thông mạnh, s sẽ được thăm trước mọi đỉnh $\in P_1$ và r sẽ được thăm trước mọi đỉnh $\in P_2$. Có hai khả năng xảy ra:

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

Bổ đề được chứng minh.

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

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



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

❋ Cài đặt

Trong việc lập trình thuật toán Kosaraju–Sharir, việc đánh số lại các đỉnh được thực hiện bằng danh sách: Tại bước duyệt đồ thị lần 1, mỗi khi duyệt xong một đỉnh thì đỉnh đó được đưa vào cuối danh sách. Sau khi đảo chiều các cung của đồ thị, chúng ta chỉ cần duyệt từ cuối danh sách sẽ được các đỉnh đúng thứ tự ngược với thứ tự duyệt xong. Danh sách nói trên có thể biểu diễn bằng một Stack.

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

KOSARAJUSHARIR.PAS ✓ Thuật toán Kosaraju–Sharir

```
#include <iostream>
```

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

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

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

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

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

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

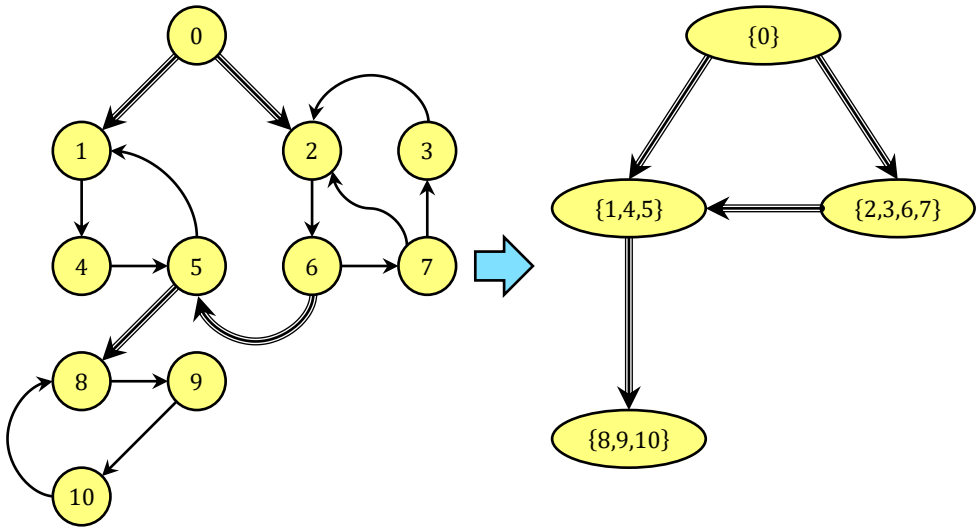
Thời gian thực hiện giải thuật có thể tính bằng hai lượt DFS, vậy nên thời gian thực hiện giải thuật sẽ là $\Theta(|V| + |E|)$ trong trường hợp đồ thị được biểu diễn bằng danh sách kề

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

24.5. Sắp xếp tô pô

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

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



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

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

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

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

Bài tập 24-1

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

Bài tập 24-2

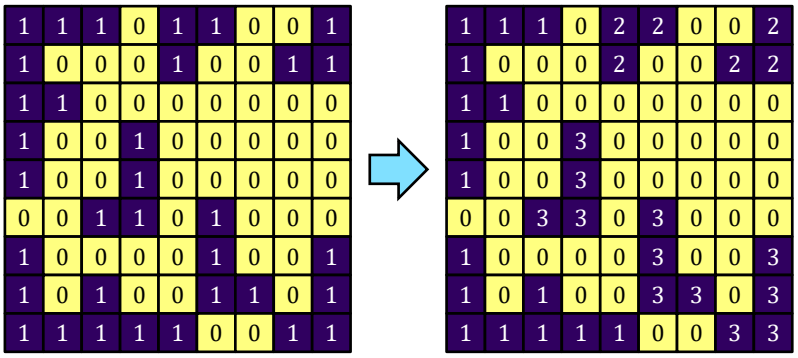
Cho đồ thị có hướng không có chu trình $G = (V, E)$ và hai đỉnh s, t. Hãy tìm thuật toán đếm số đường đi từ s tới t (chỉ cần đếm số lượng, không cần liệt kê các đường).

Bài tập 24-3

Trên mặt phẳng với hệ toạ độ Decattes 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 kỳ trong một nhóm bất kỳ có thể đi được sang nhau sau một số hữu hạn các bước di chuyển giữa hai đường tròn thông nhau.

Bài tập 24-4

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



Bài tập 24-5

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

Bài tập 24-6

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

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

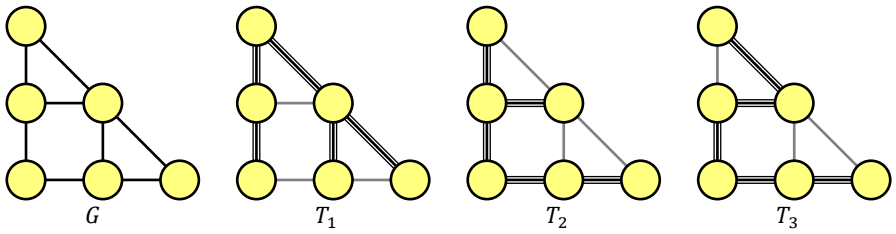
b) Tìm thuật toán và viết chương trình kiểm tra tính nửa liên thông của đồ thị.

Chương 25. Vài ứng dụng của DFS và BFS

25.1. 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. Một đồ thị vô hướng liên thông có thể có nhiều cây khung.



Hình 25-1. Đồ thị G và các cây khung T_1, T_2, T_3 của nó.

Định lý 25-1

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 hễ cứ 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) $(3 \le p \le k)$, 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 kỳ 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 mình 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 kỳ 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 đi lại được giữa hai đỉnh bất kỳ. 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 dọc theo các cạnh trên chu trình đó thì từ u có hai đường đi đơn tới v. Vô lý.

Giữa hai đỉnh (u, v) bất kỳ 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 kỳ 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 lý). Vậy T là cây.

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

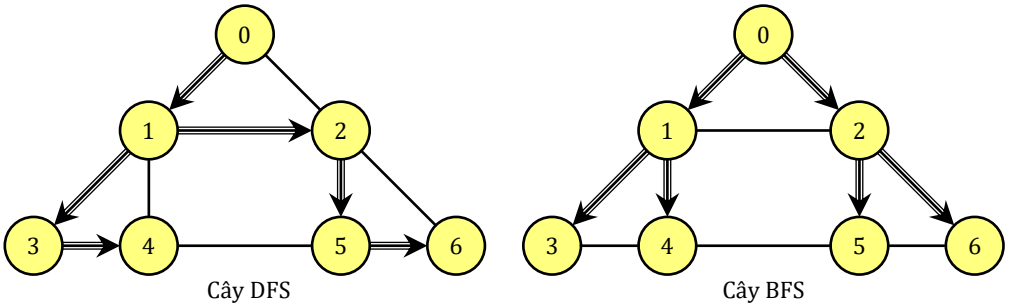
25.1.1. Xây dựng cây khung bằng thuật toán hợp nhất

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 (NEEDREF) 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.

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

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. Hình 22-4 là ví dụ về cây khung DFS và cây khung BFS của cùng một đồ thị (danh sách kề của mỗi đỉnh được sắp xếp theo thứ tự tăng dần).



Hình 25-2. Cây khung DFS và cây khung BFS của cùng một đồ thị

25.2. 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, ký 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 kỳ đều phải thuộc ít nhất 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 lý.

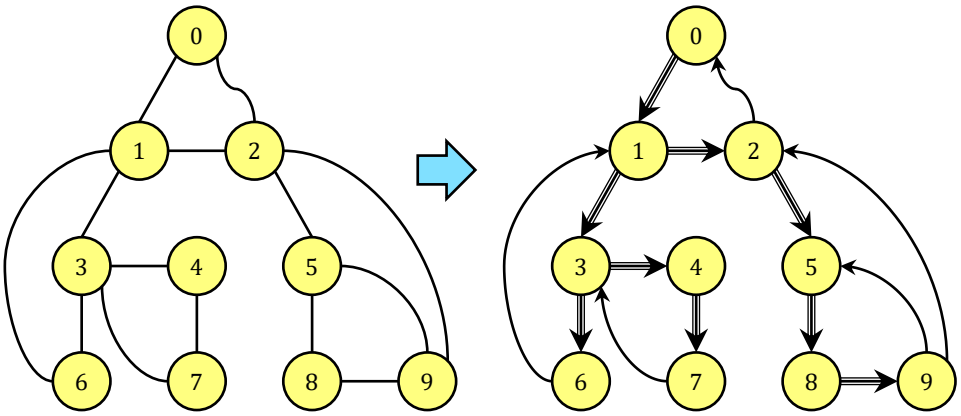
Đối với đồ thị $G = (V, E)$ có k thành phần liên thông, ta có thể xét các thành phần liên thông và xét rừng các cây khung của các thành phần đó. Khi đó có thể mở rộng khái niệm tập các chu trình cơ sở cho đồ thị vô hướng tổng quát: Mỗi khi thêm một cạnh $\in E$ nhưng không thuộc các cây khung vào rừng, 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 $|E| - |V| + k$.

25.3. 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 kỳ hay không.

Có thể tổng quát hoá 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 25-3. Phép định chiều DFS

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

```
void DFSVisit(u ∈ V)
{
    «Đánh dấu u đã thăm»;
    for (∀v: (u, v)∈ E)
    {
        «Định chiều cạnh (u, v) thành cung (u, v) ⇔ xóa cung (v, u) khỏi đồ thị»;
        if («v chưa thăm»)
            DFSVisit(v);
    }
}

«Đánh dấu mọi đỉnh đều chưa thăm»;
for ∀v ∈ V do
    if («v chưa thăm») DFSVisit(v);
```

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

Bổ đề 25-2

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 kỳ, không giảm tính tổng quát, giả sử rằng u được thăm đến trước v. Theo Định lý 23-3 (định lý 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 hàm DFSVisit(u) hoặc trong hàm DFSVisit(v).

- Nếu cạnh (u, v) được định chiều trước khi đỉnh v được thăm đến, nghĩa là việc định chiều được thực hiện trong hàm 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 thăm đến, nghĩa là khi hàm DFSVisit(v) được gọi thì cạnh (u, v) chưa định chiều. Vòng lặp bên trong hàm 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 lý 25-3

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.

"⇒"

Nếu G là định chiều được thì sau phép định chiều đó ta sẽ thu 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.

"⇐"

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 ít nhất 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 kỳ 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.

25.4. Liệt kê các khớp và cầu của đồ thị

25.4.1. Thuật toán

Việc kiểm tra tính chất của khớp/cầu có thể thực hiện trực tiếp bằng định nghĩa (Đếm số thành phần liền thông, thử xóa đỉnh/cạnh rồi đếm lại số thành phần liên thông). Tuy vậy nếu áp dụng cách thức này để liệt kê tất cả các khớp cũng như cầu thì thuật toán có độ phức tạp khá lớn. Trong phần này ta sẽ trình bày thuật toán $O(|V| + |E|)$ liệt kê các khớp/cầu của đồ thị vô hướng dựa trên mô hình DFS.

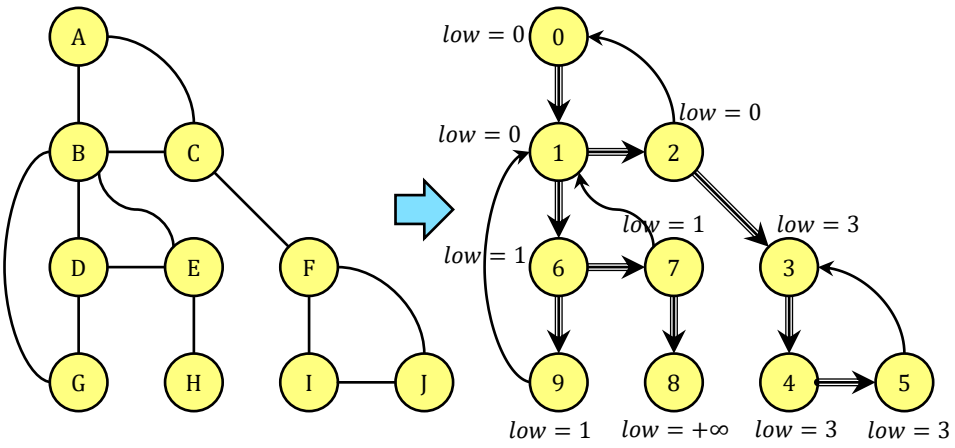
Xét phép duyệt đồ thị bằng DFS, ta đánh số các đỉnh theo thứ tự thăm đến và gọi $num[u]$ là số thứ tự của đỉnh u theo cách đánh số đó. Định nghĩa thêm $low[u]$ là giá trị $num[.]$ nhỏ nhất của những đỉnh đến được từ nhánh DFS gốc u bằng một cung ngược (tức là nếu nhánh DFS gốc u có nhiều cung ngược hướng 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 $num[.]$ và $low[.]$ tương tự như thuật toán Tarjan (mục 24.4.3): Trong hàm DFSVisit(u), trước hết $num[u]$ được gán bằng số thứ tự thăm của đỉnh u và $low[u]$ được khởi tạo bằng $+\infty$. Sau đó với từng đỉnh v kề u, thuật toán 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 hàm DFSVisit(v) thoát, nhánh DFS gốc v đã xây dựng xong và là con của u. Vì những cung ngược đi từ nhánh DFS gốc v cũng là cung ngược đi từ nhánh DFS gốc u, vì vậy $low[u]$ được cực tiểu hóa theo $low[v]$:

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

- Nếu v đã thăm thì (u, v) là một cung ngược đi từ nhánh DFS gốc u, trong trường hợp này $low[u]$ được cực tiểu hóa theo $num[v]$:

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



Hình 25-4. 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 $low[v] \geq num[v]$. Như ví dụ ở Hình 22-4, 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 tồn tại một đỉnh v con của u mà $low[v] \geq num[u]$. Như ví dụ ở Hình 22-4, 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 2 nhánh con trở lên. Như ví dụ ở Hình 22-4, gốc A không là khớp vì nó chỉ có một nhánh con trên cây DFS.

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 trên 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.

25.4.2. Cài đặt

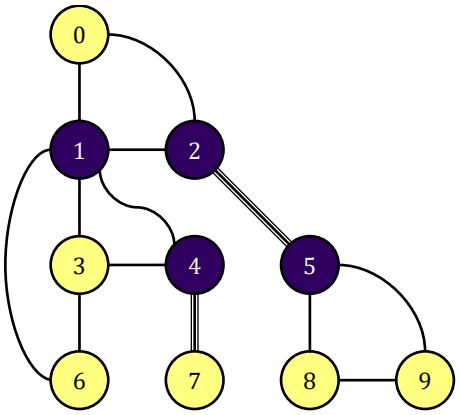
Input

- Dòng 1: Chứa hai số nguyên dương $n, m \leq 10^6$ lần lượt là số đỉnh và số cạnh 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
10 13 0 1 0 2 1 2 1 3 1 4 1 6 2 5 3 4 3 6 4 7 5 8 5 9 8 9	Bridges: 6: (2,5) 9: (4,7) Articulations: 1 2 4 5



Đồ thị được biểu diễn bằng một danh sách cạnh e, mỗi cạnh là một bản ghi (struct) có hai trường x, y là chỉ số hai đỉnh đầu mút của cạnh. Mỗi đỉnh u được gắn với một danh sách liên thuộc $adj[u]$ chứa chỉ số* các cạnh liên thuộc với u. Với một cạnh $e[i]$ liên thuộc với u, để chỉ ra đỉnh v kề với u qua cạnh e có thể dùng công thức $v = e[i].x + e[i].y - u$.

Việc định chiều đồ thị chẳng qua là thao tác "ép" quá trình DFS chỉ được duyệt qua mỗi cạnh một lần theo một chiều. Cơ chế này được thực hiện trong chương trình như sau: khi duyệt qua mỗi cạnh (u, v), ta sẽ đánh dấu vô hiệu hóa luôn cạnh đó để nếu quá trình DFS về sau duyệt vào cạnh này (theo chiều ngược lại (v, u)) sẽ bỏ qua luôn.

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

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

* Có thể tối ưu hóa cách cài đặt bằng cách dùng $adj[u]$ chứa các con trỏ tới các cạnh liên thuộc với u

```
const int maxM = 1e6;
int n, m;

struct TEdge //Cấu trúc cạnh
{
    int x, y; //Hai đỉnh đầu mút
    bool invalid; //Cạnh đã đã vô hiệu hóa chưa
} e[maxM];

vector<int> adj[maxN]; //Các danh sách liên thuộc
int num[maxN], low[maxN];
bool isCut[maxN], isBridge[maxM]; //Đánh dấu đỉnh/cạnh có phải khớp/cầu không

void ReadInput() //Đọc dữ liệu
{
    cin >> n >> m;
    for (int i = 0; i < m; ++i)
    {   //Đọc một cạnh và thêm chỉ số cạnh vào hai danh sách liên thuộc của hai đầu mút
        cin >> e[i].x >> e[i].y;
        e[i].invalid = false;
        adj[e[i].x].push_back(i);
        adj[e[i].y].push_back(i);
    }
}

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

void DFSVisit(int u) //Thuật toán DFS bắt đầu từ u
{
    bool isRoot = num[u] == -2; //isRoot: u có phải gốc một cây DFS không
    static int Time = 0; //Biến đếm số thứ tự thăm
    num[u] = Time++; //Đánh số u theo thứ tự thăm
    low[u] = maxN; //low[u] khởi tạo bằng +∞
    int nBranches = 0; //Biến đếm số nhánh con của u trên cây DFS
    for (int i: adj[u])
    {   //Xét cạnh e[i] liên thuộc u
        if (e[i].invalid) continue; //e[i] đã bị vô hiệu hóa, bỏ qua
        e[i].invalid = true; //vô hiệu hóa luôn cạnh e[i] vừa duyệt qua
        int v = e[i].x + e[i].y - u; //Xét đỉnh v kề u qua e[i]
        if (num[v] == -1) //v chưa thăm
        {
            DFSVisit(v); //Thăm v
            Minimize(low[u], low[v]); //Cực tiểu hóa low[u] theo low[v]
            isBridge[i] = low[v] >= num[v]; //low[v] ≥ num[v] → e[i] là cầu
            isCut[u] |= low[v] >= num[u]; //low[v] ≥ num[u] → u là khớp
            ++nBranches; //Đếm số nhánh con của u trên cây DFS
        }
        else //v đã thăm
            Minimize(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
    }
    if (isRoot && nBranches < 2) //Nếu u là gốc và có ít hơn 2 nhánh con
        isCut[u] = false; //→ u đã bị đánh dấu nhầm là khớp nên phải đặt lại
}

void Solve()
{
    fill(num, num + n, -1); //Các đỉnh đều chưa thăm, được đánh số bởi giá trị âm
    fill(isCut, isCut + n, false);
    fill(isBridge, isBridge + m, false);
    for (int u = 0; u < n; ++u)
        if (num[u] == -1)
        {
```

```
            num[u] = -2; //Cho biết u là gốc cây DFS trước khi thăm
            DFSVisit(u); //Duyệt DFS từ gốc u
        }
    cout << "Bridges:\n"; //In các cầu
    for (int i = 0; i < m; ++i)
        if (isBridge[i])
            cout << i << ": (" << e[i].x << ',' << e[i].y << ")\n";
    cout << "Articulations:\n"; //In các khớp
    for (int u = 0; u < n; ++u)
        if (isCut[u]) cout << u << '\n';
}

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

Việc cài đặt thuật toán liệt kê khớp/cầu dễ gặp sai sót, nhất là trong trường hợp đồ thị có khuyên, đỉnh cô lập hay cạnh song song. Vì vậy cần có thiết kế và kiểm thử cẩn thận trước và sau khi viết chương trình. Một lưu ý nữa là nếu chỉ liệt kê khớp, ta không cần định chiều đồ thị, chỉ đơn giản là coi mỗi cạnh vô hướng tương ứng với hai cung có hướng ngược chiều nhau.

25.5. Các thành phần song liên thông

25.5.1. Các khái niệm và thuật toán

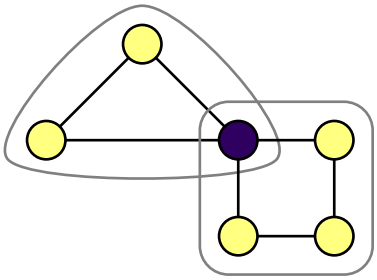
Đồ thị vô hướng liên thông được gọi là đồ thị song liên thông nếu nó không có khớp, tức là việc bỏ đi một đỉnh bất kỳ của đồ thị không ảnh hưởng tới tính liên thông của các đỉnh còn lại. Ta quy ước rằng đồ thị chỉ gồm một đỉnh và không có cạnh nào cũng là một đồ thị song liên thông.

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 đỉnh khớp nhưng không có cạnh nào chung*.

---

* Một số tài liệu định nghĩa một thành phần song liên thông là một tập tối đại các **cạnh** sao cho các cạnh này và các đầu mút của chúng tạo ra một đồ thị song liên thông, định nghĩa này tương đồng với định nghĩa trong bài ngoại trừ việc không chấp nhận thành phần song liên thông chỉ gồm một đỉnh cô lập.



Hình 25-5. Đồ thị và hai thành phần liên thông có chung khớp

Coi mỗi cạnh vô hướng tương ứng với hai cung có hướng ngược chiều nhau. Xét mô hình đánh số đỉnh theo thứ tự thăm đến và ghi nhận cung ngược lên cao nhất...

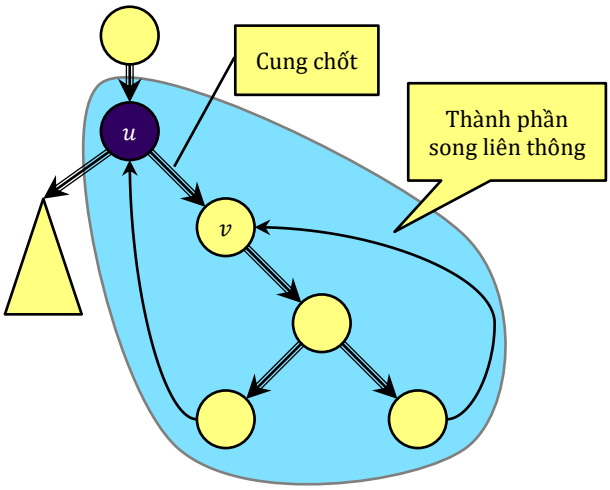
```
void DFSVisit(u ∈ V)
{
    «Đánh dấu u đã thăm»;
    num[u] = «Số thứ tự thăm đỉnh u»;
    low[u] = +∞;
    for (∀v ∈ adj[u]) //Xét mọi đỉnh v kề u
        if («v chưa thăm»)
        {
            DFSVisit(v); //Đi thăm v
            low[u] = min(low[u], low[v]); //Cực tiểu hoá low[u] theo low[v]
        }
        else //v đã thăm
            low[u] = min(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
}

«Đánh dấu mọi đỉnh chưa thăm»;
for (∀u ∈ V)
    if («u chưa thăm») DFSVisit(v);
```

Trong sơ đồ cài đặt này, $num[u]$ là số hiệu của đỉnh u theo thứ tự thăm đến và $low[u]$ là giá trị $num[.]$ nhỏ nhất của một đỉnh kề với nhánh DFS gốc u (đỉnh cao nhất trên cây DFS kề với một đỉnh thuộc nhánh DFS gốc u).

Xét hàm 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 hóa $low[u]$ theo $low[v]$. Tại thời điểm này, nếu $low[v] \geq num[u]$ thì hoặc u là khớp hoặc u là gốc của một cây DFS. Để tiện trình bày, trong trường hợp này ta gọi cung (u, v) là *cung chốt* 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 thăm 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 thấy rằng nếu (u, v) là một cung chốt 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 chốt nào (tức là nhánh DFS gốc v không chứa cung chốt 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 (Hình 22-4).



Hình 25-6. Cung chốt và thành phần song liên thông

Thuật toán liệt kê các thành phần song liên thông dựa trên chính nhận xét này: Mỗi khi quá trình DFS thăm tới một cung, cung này được đẩy vào một stack để mỗi khi quá trình DFS duyệt xong một cung chốt (u, v), ta chỉ việc lấy các cung ra khỏi stack cho tới khi lấy được cung (u, v), các đỉnh đầu mút của các cung lấy ra từ stack chính là một thành phần song liên thông. Cơ chế thực hiện khá giống với thuật toán Tarjan (mục 24.4.3), chỉ là thay khái niệm "chốt" bởi "cung chốt" và stack dùng để chứa các cung DFS thay vì chứa các đỉnh.

Vấn đề duy nhất còn phải xử lý là quy ước một đỉnh cô lập của đồ thị cũng là một thành phần song liên thông. Thuật toán trên chỉ liệt kê được thành phần song liên thông có ít nhất 1 cạnh nên sẽ bỏ sót các đỉnh cô lập. 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.

```
void DFSVisit(u ∈ V)
{
    «Đánh dấu u đã thăm»;
    num[u] = «Số thứ tự thăm đỉnh u»;
    low[u] = +∞;
    for (∀v ∈ adj[u]) //Xét mọi đỉnh v kề u
        if («v chưa thăm»)
        {
            «Đẩy cung (u, v) vào stack»;
            DFSVisit(v); //Đi thăm v
            low[u] = min(low[u], low[v]); //Cực tiểu hoá low[u] theo low[v]
            if (low[v] ≥ num[u]) //(u, v) là cung chốt
                «lấy các cung khỏi stack cho tới khi lấy được cung (u, v)
                 Các đỉnh đầu mút tạo thành một thành phần song liên thông»;
        }
        else //v đã thăm
            low[u] = min(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
}

«Đánh dấu mọi đỉnh chưa thăm»;
for (∀u ∈ V)
    if («u chưa thăm»)
    {
        DFSVisit(u);
        if («u là đỉnh cô lập»)
            «Liệt kê thành phần song liên thông chỉ gồm 1 đỉnh u»;
    }
```

25.5.2. Cài đặt

Có một chi tiết nhỏ nhằm tăng hiệu suất của chương trình ở cách sử dụng stack. Để ý rằng stack chỉ chứa các cung DFS và khi lấy các cung ra khỏi Stack ứng với một thành phần song liên thông thì thực chất là ta lấy được cả một nhánh DFS. Để in ra các đỉnh của nhánh này thì ngoài đỉnh gốc nhánh, với mỗi cung DFS (u, v), ta chỉ đưa ra đầu mút v. Vì vậy thay vì lưu trữ một cung (u, v) trong stack, ta chỉ cần lưu trữ đầu mút v của cung là đủ.

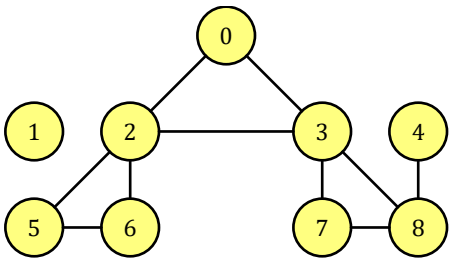
Input

- Dòng 1: Chứa hai số nguyên $n, m \leq 10^6$ lần lượt là số đỉnh và số cạnh 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 0 2 0 3 2 3 2 5 2 6 3 7 3 8 4 8 5 6 7 8	Biconnected component: 4 8 Biconnected component: 8 7 3 Biconnected component: 6 5 2 Biconnected component: 3 2 0 Biconnected component: 1



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

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
const int maxN = 1e6;
const int maxM = 1e6;
int n, m;

vector<int> adj[maxN]; //Các danh sách kề
int num[maxN], low[maxN];
stack<int> Stack;
int Time; //Biến đếm số thứ tự thăm

void ReadInput() //Đọc dữ liệu
{
    cin >> n >> m;
    for (int i = 0; i < m; ++i)
    {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
}

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

```
void DFSVisit(int u) //Thuật toán DFS bắt đầu từ u
{
    num[u] = Time++; //Đánh số u theo thứ tự thăm
    low[u] = maxN; //low[u] khởi tạo bằng +∞
    for (int v: adj[u])
        if (num[v] == -1) //v chưa thăm
        {
            Stack.push(v);
            DFSVisit(v); //Thăm v
            Minimize(low[u], low[v]); //Cực tiểu hóa low[u] theo low[v]
            if (low[v] >= num[u]) //(u, v) là cung chốt
            {
                cout << "Biconnected component:\n";
                int vertex;
                do //Lấy các đỉnh khỏi stack cho tới khi v được lấy ra
                {
                    vertex = Stack.top();
                    Stack.pop();
                    cout << vertex << ' ';
                }
                while (vertex != v);
                cout << u << '\n'; //in thêm đỉnh u là gốc nhánh DFS ứng với tp song liên thông
            }
        }
        else //v đã thăm
            Minimize(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
}

void Solve()
{
    fill(num, num + n, -1); //Các đỉnh đều chưa thăm, được đánh số bởi giá trị âm
    Time = 0; //Khởi tạo biến đếm số thứ tự thăm
    for (int u = 0; u < n; ++u)
        if (num[u] == -1) //u chưa thăm
        {
            int OldTime = Time;
            DFSVisit(u); //Duyệt DFS từ u
            if (Time - OldTime == 1) //u là đỉnh cô lập
                cout << "Biconnected component:\n" << u << "\n";
        }
}

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

Bài tập 25-1

Cho một đồ thị n đỉnh, ban đầu chưa có cạnh nào, người ta lần lượt thêm vào m cạnh vô hướng. Yêu cầu cho biết số cặp đỉnh đi sang được nhau sau mỗi bước thêm cạnh. Yêu cầu thuật toán với độ phức tạp $O\big((m+n)\alpha(n)\big)$.

Bài tập 25-2

Cho một đồ thị hình cây gồm n đỉnh, người ta thực hiện m phép duyệt, mỗi phép duyệt cho bởi cặp đỉnh (s, t) và duyệt qua tất cả các cạnh trên đường đi đơn từ s tới t. Tìm thuật toán $O(n+m)$ xác định các cạnh của cây không được duyệt qua.

Gợi ý: Với mỗi phép duyệt bổ sung thêm cạnh (s, t) vào đồ thị, với đồ thị mới tạo thành, những cầu chắc chắn là những cạnh trên cây ban đầu mà không được duyệt qua

Bài tập 25-3

Cho một bảng hình chữ nhật được chia làm lưới ô vuông đơn vị, gọi k là diện tích bảng. Mỗi ô được tô một màu. Một miền là một tập hợp các ô của bảng sao cho từ một ô của miền có thể đi sang mọi ô khác bằng các phép di chuyển qua các ô kề cạnh.

Tìm thuật toán $O(k)$ xác định miền lớn nhất sao cho các ô của miền cùng màu.

Tìm thuật toán $O(k \log k)$ xác định miền lớn nhất chỉ gồm hai màu.

Bài tập 25-4

Cho một đồ thị vô hướng G gồm n đỉnh và m cạnh, tìm thuật toán $O(n + m)$ cho biết từng mỗi đỉnh u, nếu xóa đỉnh đó cùng những cạnh liên thuộc thì đồ thị còn lại (gồm $n - 1$ đỉnh) có bao nhiêu thành phần liên thông

Bài tập 25-5

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)

Gợi ý: Tìm đọc thêm về định lý Kirchhoff và ma trận Laplace

Bài tập 25-6

Tìm hiểu các giải pháp xác định khớp/cầu/số thành phần song liên thông với đồ thị động gồm thao tác thêm đỉnh, thêm cạnh và truy vấn khớp, cầu cũng như số thành phần song liên thông.