Phần VI.
Các thuật toán trên đồ thị


Leonhard Euler
1707-1783

Trên thực tế có nhiều bài toán liên quan tới một tập các đối tượng và những mối liên hệ giữa chúng, đòi hỏi toán học phải đặt ra một mô hình biểu diễn một cách chặt chẽ và tổng quát bằng ngôn ngữ ký hiệu, đó là đồ thị: một mô hình toán học gồm các đỉnh biểu diễn các đối tượng và các cạnh biểu diễn mối quan hệ giữa các đối tượng. Những ý tưởng cơ bản của đồ thị được đưa ra từ thế kỷ thứ XVIII bởi nhà toán học Thuỵ Sĩ Leonhard Euler. Năm 1736, ông đã dùng mô hình đồ thị để giải bài toán về bảy cây cầu Königsberg (*Seven Bridges of Königsberg*). Bài toán này cùng với bài toán mã đi tuần (*Knight Tour*) được coi là những bài toán đầu tiên của lý thuyết đồ thị.

Rất nhiều bài toán của lý thuyết đồ thị đã trở thành nổi tiếng và thu hút được sự quan tâm lớn của cộng đồng nghiên cứu. Ví dụ bài toán bốn màu, bài toán đẳng cấu đồ thị, bài toán người du lịch, bài toán người đưa thư Trung Hoa, bài toán đường đi ngắn nhất, luồng cực đại trên mạng v.v… Trong phạm vi chuyên đề này, chúng ta sẽ xem xét lý thuyết đồ thị dưới góc độ người lập trình, tức là khảo sát những thuật toán xử lý đồ thị và một số ứng dụng của chúng.

Chương 22. Các khái niệm cơ bản

22.1. Đồ thị

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

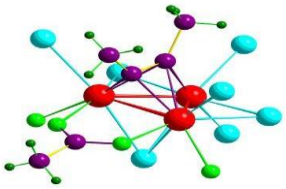
$$Graph = Objects + Connections$$
$$G = (V, E)$$

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

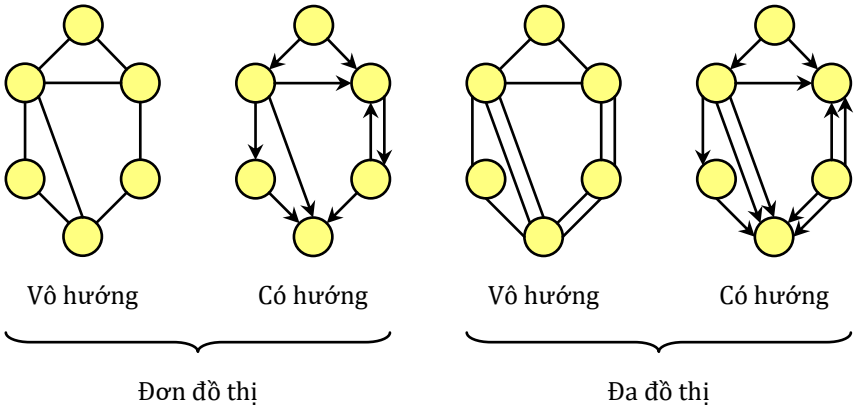
Hình 22-1. Một số hình ảnh của đồ thị

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 22-2 là ví dụ về đơn đồ thị/đa đồ thị có hướng/vô hướng.



Hình 22-2. Phân loại đồ thị

22.2. Các khái niệm

22.2.1. 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.

Định lý 22-1

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

$$G = (V, E) \Longrightarrow \sum_{\forall u \in V} \deg(u) = 2|E| \qquad (22.1)$$

Chứng minh

Vì đồ thị không có khuyên, khi lấy tổng tất cả các bậc đỉnh thì 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.

Định lý cũng có thể mở rộng ra trong trường hợp đồ thị có khuyên, đơn giản là mỗi khuyên (u, u) coi như được tính hai lần trong $\deg(u)$.

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 22-2

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:

$$G = (V, E) \Longrightarrow \sum_{\forall u \in V} \deg^+(u) = \sum_{\forall u \in V} \deg^-(u) = |E| \qquad (22.2)$$

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.

Chú ý rằng định lý đúng ngay cả khi đồ thị có khuyên, tức là có cung nối từ một đỉnh đến chính nó. Khuyên (u, u) khi đó được tính một lần trong $\deg^-(u)$ và cũng được tính một lần trong $\deg^+(u)$.

22.2.2. Đường đi và chu trình

Đường đi (*walk*) trên đồ thị là một cách di chuyển giữa hai đỉnh qua một số hữu hạn các cạnh. Cụ thể là một dãy các đỉnh:

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

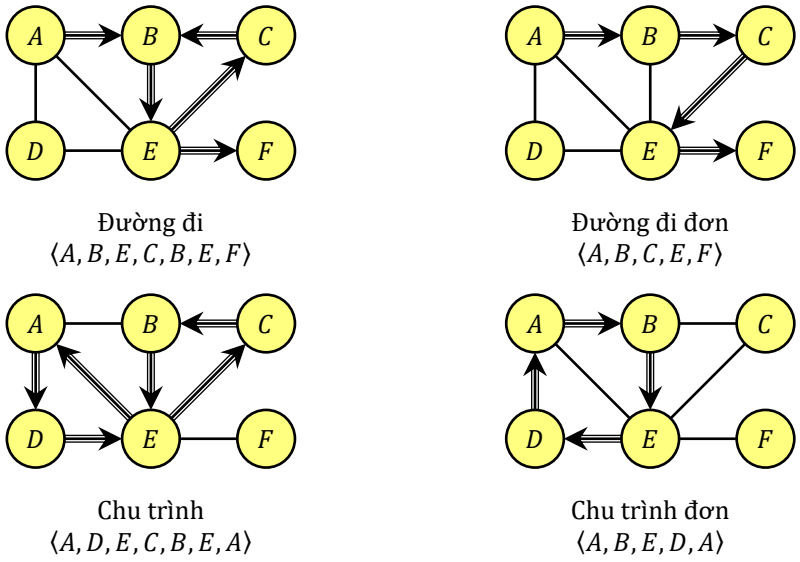
sao cho $(p_{i-1}, p_i) \in E, \forall i: 1 \le i \le k$.

Mặc dù người ta thường ký hiệu đường đi bởi dãy các đỉnh (theo đúng thứ tự trên đường đi), ta cần hiểu rằng đường đi được xác định bằng cả dãy các đỉnh và dãy các cạnh trên đường đi đó. Điều này đảm bảo tính chính xác của định nghĩa đường đi trên đa đồ thị.

Xét một đường đi $P = \langle p_0, p_1, \dots, p_k \rangle$, đường đi này xuất phát từ đỉnh p_0, qua cạnh (p_0, p_1) sang đỉnh p_1, qua tiếp cạnh (p_1, p_2) sang đỉnh p_2,..., cuối cùng qua cạnh (p_{k-1}, p_k) sang đỉnh p_k. Ta nói p_k *đến được* (*reachable*) từ p_0 hay p_0 đến được p_k, ký hiệu $p_0 \rightsquigarrow p_k$. Đỉnh p_0 được gọi là đỉnh đầu, đỉnh p_k gọi là đỉnh cuối, còn các đỉnh $p_1, p_2, \dots, p_{k-1}$ được gọi là những *đỉnh trong* (*internal vertices*) của đường đi P. Đường đi có đỉnh đầu trùng với đỉnh cuối được gọi là một *chu trình* (*closed walk*).

Một đường đi gọi là *đơn giản* hay *đường đi đơn* (*simple walk/path*) nếu tất cả các đỉnh trên đường đi là hoàn toàn phân biệt (dĩ nhiên khi đó các cạnh trên đường đi

cũng hoàn toàn phân biệt). Một chu trình được gọi là chu trình đơn (*cycle*) nếu tập đỉnh đầu (cũng là đỉnh cuối) và các đỉnh trong hoàn toàn phân biệt*.



Hình 22-3. Đường đi và chu trình

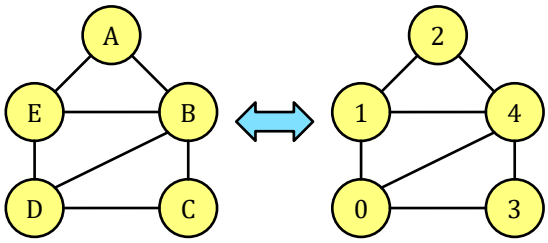
Hình 22-3 là ví dụ về một số đường đi và chu trình trên một đồ thị vô hướng (mũi tên trong hình là chỉ hướng đi, còn cạnh của đồ thị là vô hướng). Trong ví dụ này, $\langle A, B, E, C, B, E, F\rangle$ là một đường đi từ A tới F, nhưng đây không phải là đường đi đơn do nó đi qua đỉnh B và E hai lần, $\langle A, B, C, E, F\rangle$ là một đường đi đơn từ A tới F. $\langle A, D, E, C, B, E, A\rangle$ là một chu trình bắt đầu và kết thúc ở A, nó không phải chu trình đơn vì có đỉnh trong E đi qua hai lần. Chú ý rằng $\langle A, B, A\rangle$ cũng không phải chu trình đơn, dù đỉnh đầu A và đỉnh trong B là phân biệt, nhưng có cạnh (A, B) bị lặp lại. $\langle E, A, D, E, C, B, E\rangle$ cũng không phải chu trình đơn, dù các cạnh và các đỉnh trong $\{A, D, E, C, B\}$ hoàn toàn phân biệt, bởi nó có đỉnh trong E trùng với đỉnh đầu (cũng là đỉnh cuối) chu trình. Ví dụ về chu trình đơn trong đồ thị này có thể là $\langle A, B, E, D, A\rangle$ hoặc $\langle B, C, E, B\rangle$.

22.2.3. 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' (Hình 22-4).

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



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

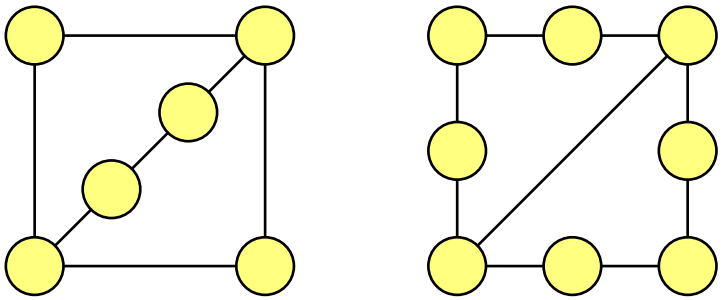
Hình 22-4. Đẳng cấu

❋ Đồng phôi

Với đồ thị $G = (V, E)$, và (u, v) là một cạnh $\in E$, phép chia cạnh (u, v) là thao tác bổ sung thêm một đỉnh w nằm trên cạnh (u, v) để tách nó ra thành hai cạnh (u, w) và (w, v). Sau phép chia cạnh, ta thu được đồ thị mới $G_s = (V_s, E_s)$ trong đó:

$$V_s = V \cup \{w\}$$
$$E_s = E - \{(u, v)\} \cup \{(u, w), (w, v)\}$$

Hai đồ thị $G = (V, E)$ và $G' = (V', E')$ được gọi là đồng phôi (*Homeomorphism*) nếu như tồn tại một dãy các phép chia cạnh trên G và G' để biến chúng thành hai đồ thị đẳng cấu (Hình 22-5).



G $\qquad$ G'

Hình 22-5. Đồng phôi

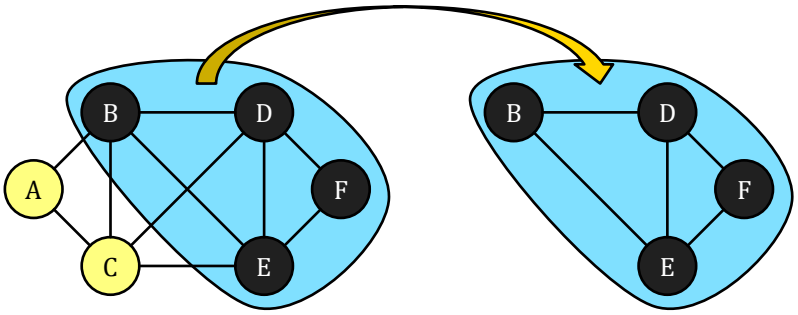
❋ Đồ thị con

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

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

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

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



Hình 22-6. 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).

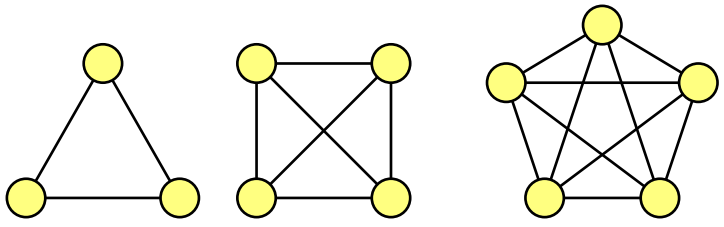
Với một đồ thị có hướng $G = (V, E)$, ta gọi *phiên bản vô hướng* (*undirected version*) của G là một đồ thị vô hướng $G' = (V, E')$ tạo thành bằng cách thay mỗi cung (u, v) bằng cạnh vô hướng (u, v). Nói cách khác, G' tạo thành từ G bằng cách bỏ đi chiều của cung.

❋ 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

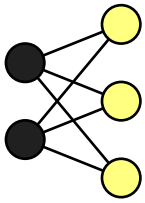
Một đồ thị vô hướng được gọi là *đầy đủ* (*complete*) nếu mọi cặp đỉnh đều là kề nhau, đồ thị đầy đủ gồm n đỉnh ký hiệu là K_n. Hình 22-7 là ví dụ về các đồ thị K_3, K_4 và K_5.



Hình 22-7. 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 22-8 là ví dụ về đồ thị hai phía đầy đủ $K_{2,3}$.



Hình 22-8. Đồ thị hai phía đầy đủ

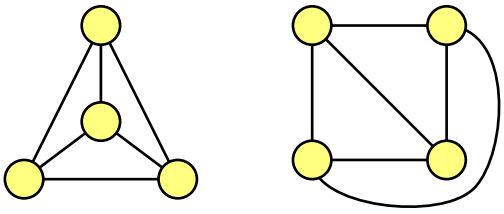
❋ Đồ thị phẳng

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 22-9



Hình 22-9. Hai cách biểu diễn phẳng của đồ thị đầy đủ K_4

Định lý 22-3 (Định lý Kuratowski)

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

Định lý 22-4 (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 22-5

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 22-5 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 23. 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.

23.1. Ma trận kề

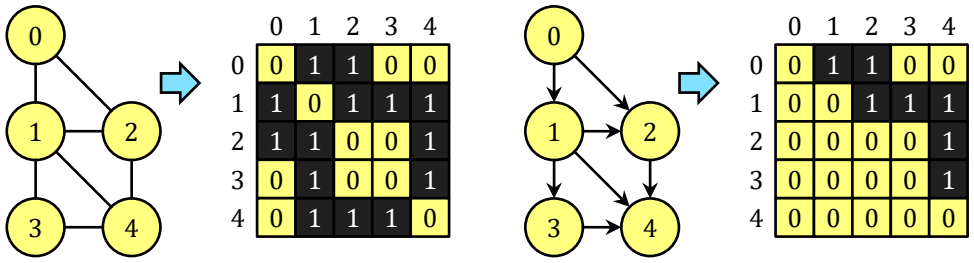
Với $G = (V, E)$ là một đơn đồ thị có hướng trong đó $|V| = n$, ta có thể đánh số các đỉnh từ 0 tới $n - 1$ và đồng nhất mỗi đỉnh với số thứ tự của nó. Bằng cách đánh số như vậy, đồ thị G có thể biểu diễn bằng ma trận vuông $A = \{a_{ij}\}_{n \times n}$ với các hàng và các cột đánh số từ 0 tới $n - 1$. 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 23-1. 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: $a_{uv} \neq 0$

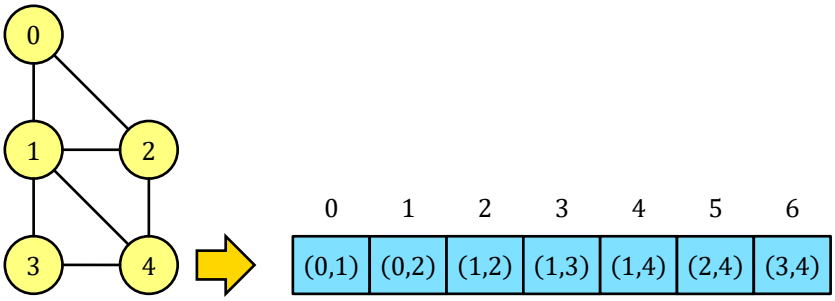
Nhược điểm của ma trận kề

- 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}.

23.2. 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 23-2. Danh sách cạnh

Ưu điểm của danh sách cạnh:

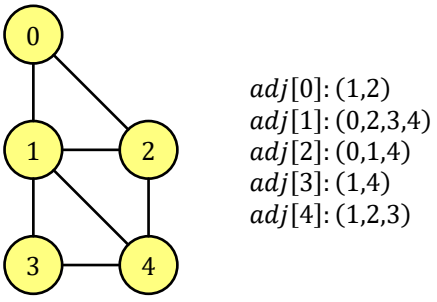
- Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ, bởi nó chỉ cần $O(m)$ ô nhớ để lưu danh sách cạnh.
- Trong một số trường hợp, ta phải xét tất cả các cạnh của đồ thị thì cài đặt trên danh sách cạnh làm cho việc duyệt các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

23.3. Danh sách kề

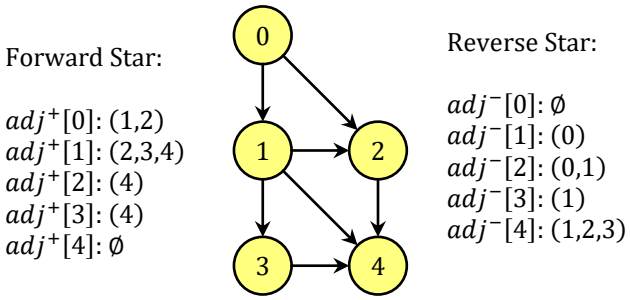
Để khắc phục nhược điểm của các phương pháp ma trận kề và danh sách cạnh, người ta đề xuất phương pháp biểu diễn đồ thị bằng *danh sách kề* (*adjacency list*). Trong cách biểu diễn này, với mỗi đỉnh u của đồ thị vô hướng, ta cho tương ứng với nó một danh sách $adj[u]$ gồm các đỉnh kề với u (Hình 23-4).



Hình 23-3. 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 23-4. 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.

23.4. 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.

23.5. 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 23-1

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 23-2

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 23-3

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 23-4

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 23-5

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 23-6

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 23-7

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 23-8

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 23-9

Xét đồ thị vô hướng $G = (V, E)$ với ma trận liên thuộc B định nghĩa như trong Bài tập 23-8. 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 24. Các thuật toán tìm kiếm trên đồ thị

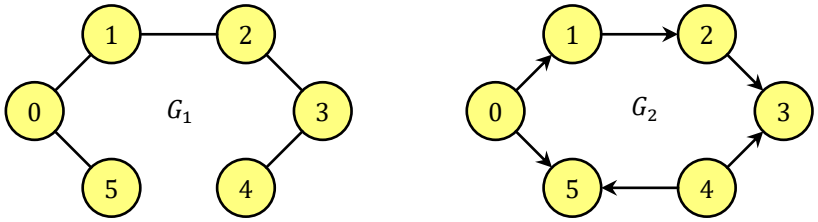
24.1. Bài toán tìm đường

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 24-1. Đồ thị và đường đi

Trên cả hai đồ thị ở Hình 24-1, ⟨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:

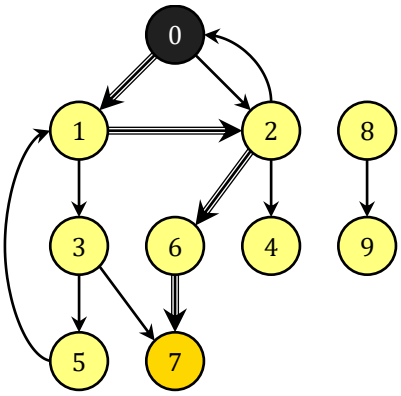
Input

- 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



24.2. 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).

24.3. Thuật toán tìm kiếm theo chiều sâu

24.3.1. 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) //Đánh dấu các đỉnh đều chưa thăm
    avail[u] = true;
DFSVisit(s);
if (!avail[t]) //Từ s có đường tới t
    «Truy theo vết từ t để tìm đường đi từ s tới t»;
```

24.3.2. Cài đặt

DFS.cpp 🗎 Tìm đường bằng DFS

```cpp
#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
        }
}

```

```
33 | void PrintPath() //In đường đi
34 | {
35 |     if (avail[t]) //t chưa thăm: không có đường từ s tới t
36 |         cout << "There's no path from " << s << " to " << t << '\n';
37 |     else
38 |     {
39 |         cout << "The path from " << s << " to " << t << ":\n";
40 |         for (int u = t; u != s; u = trace[u]) //Truy vết ngược từ t về s
41 |             cout << u << " <- ";
42 |         cout << s << "\n";
43 |     }
44 | }
45 |
46 | int main()
47 | {
48 |     Enter();
49 |     fill(avail, avail + n, true); //Khởi tạo các đỉnh đều chưa thăm
50 |     cout << "Reachable vertices from " << s << ":\n";
51 |     DFSVisit(s); //Thuật toán tìm kiếm theo chiều sâu bắt đầu từ s
52 |     cout << "\n";
53 |     PrintPath();
54 | }
```

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:

Cách 1: Lưu đường đi vào mảng và in mảng ra theo thứ tự ngược lại.

Cách 2: Viết một hàm đệ quy `DoTrace(t)` để truy vết:

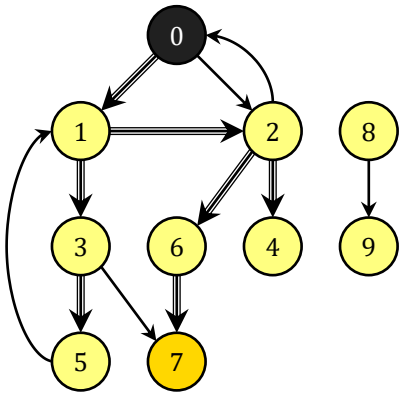
```
1 | void DoTrace(int v) //In ra đường đi từ s tới v
2 | {
3 |     if (v != s)
4 |     {
5 |         DoTrace(trace[v]); //In ra đường đi từ s tới trace[v] trước
6 |         cout << " -> ";
7 |     }
8 |     cout << v; //Sau đó in ra v
9 | }
```

Cách 3: 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.

24.3.3. 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.



Hình 24-2. Cây DFS

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 24-2 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

Trong bài toán này, thực ra ta mới chỉ khảo sát một ứng dụng của thuật toán DFS để liệt kê các đỉnh đến được từ một đỉnh s cho trước. Mô hình cài đặt đầy đủ của thuật toán DFS cho phép duyệt qua tất cả các đỉnh cũng như tất cả các cạnh của đồ thị:

```
1  | void DFSVisit(u ∈ V)
2  | {
3  |     d[u] = Time++; //d[u] = thời điểm u được thăm, cũng là đánh dấu ≠ -1
4  |     Output ← u; //Thăm tới đỉnh nào in ra luôn đỉnh đó
5  |     for (∀v ∈ adj[u]) //duyệt mọi đỉnh v nối từ u
6  |         if (d[v] == -1) //nếu v chưa thăm
7  |             DFSVisit(v); //Gọi đệ quy tìm kiếm theo chiều sâu từ v
8  |     f[u] = Time++; //f[u] = thời điểm u được duyệt xong
9  | }
10 |
11 | Input → đồ thị G;
12 | for (∀u ∈ V) //Khởi tạo các đỉnh đều chưa thăm: d[.] = -1
13 |     d[u] = -1;
14 | Time = 0;
15 | for (∀u ∈ V) //Xét mọi đỉnh chưa thăm
16 |     if (d[u] == -1)
17 |         DFSVisit(u);
```

❋ Thời gian thực hiện giải thuật

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

Trong mô hình duyệt đồ thị bằng DFS, 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 đồ thị bằng DFS, 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…

Bổ đề 24-1

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 Bổ đề 24-1, ta có ĐPCM.

Bổ đề 24-2

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 24-3 (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 Bổ đề 24-1, 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 Bổ đề 24-2, y phải được thăm trước khi duyệt

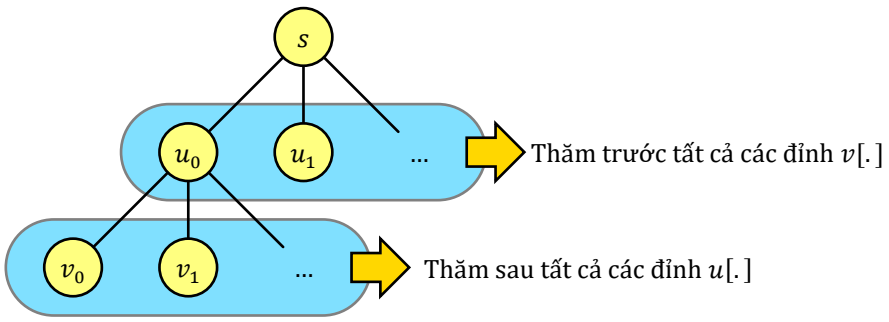
xong x: $d_y < f_x$; x lại là hậu duệ của u nên theo Bổ đề 24-1, ta có $f_x \le f_u$, vậy $d_y < f_u$. Mặt khác theo giả thiết rằng tại thời điểm d_u thì y chưa được thăm, tức là $d_u < d_y$, kết hợp lại ta có $d_u < d_y < f_u$, vậy thì y là hậu duệ của u theo Bổ đề 24-1, trái với giả thiết phản chứng.

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.

24.4. Thuật toán tìm kiếm theo chiều rộng

24.4.1. Ý tưởng

Thực ra ta đã biết về thuật toán này trong khi tìm hiểu ứng dụng của cấu trúc dữ liệu hàng đợi (thuật toán loang). Tư tưởng của thuật toán tìm kiếm theo chiều rộng (Breadth-First Search – BFS) là "lập lịch" duyệt các đỉnh. Việc thăm một đỉnh sẽ lên lịch duyệt các đỉnh nối từ nó sao cho thứ tự duyệt là ưu tiên chiều rộng (đỉnh nào gần đỉnh xuất phát s hơn sẽ được duyệt trước). Đầu tiên ta thăm đỉnh s. Việc thăm đỉnh s sẽ phát sinh thứ tự thăm những đỉnh $u_0, u_1, \ldots$ 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 24-3).



Hình 24-3. 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»;
```

24.4.2. 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
    }
}

```

```
22 | void BFS() //Thuật toán tìm kiếm theo chiều rộng
23 | {
24 |     queue<int> Queue({s}); //Hàng đợi ban đầu chỉ gồm một đỉnh s
25 |     fill (avail, avail + n, true); //Khởi tạo các đỉnh đều chưa thăm
26 |     avail[s] = false; //Riêng đỉnh s đã thăm
27 |     do
28 |     {
29 |         int u = Queue.front(); Queue.pop(); //Lấy u từ hàng đợi
30 |         cout << u << ", ";
31 |         for (int v: adj[u]) //Xét mọi đỉnh v nối từ u
32 |             if (avail[v]) //Nếu v chưa thăm
33 |             {
34 |                 trace[v] = u; //Lưu vết đường đi s → … → u → v
35 |                 Queue.push(v); //Đẩy v vào hàng đợi
36 |                 avail[v] = false;
37 |             }
38 |     }
39 |     while (!Queue.empty());
40 | }
41 |
42 | void PrintPath() //In đường đi
43 | {
44 |     if (avail[t]) //t chưa thăm: không có đường từ s tới t
45 |         cout << "There's no path from " << s << " to " << t << '\n';
46 |     else
47 |     {
48 |         cout << "The path from " << s << " to " << t << ":\n";
49 |         for (int u = t; u != s; u = trace[u]) //Truy vết ngược từ t về s
50 |             cout << u << " <- ";
51 |         cout << s << "\n";
52 |     }
53 | }
54 |
55 | int main()
56 | {
57 |     Enter();
58 |     cout << "Reachable vertices from " << s << ":\n";
59 |     BFS(); //Thuật toán tìm kiếm theo chiều rộng
60 |     cout << "\n";
61 |     PrintPath();
62 | }
```

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.

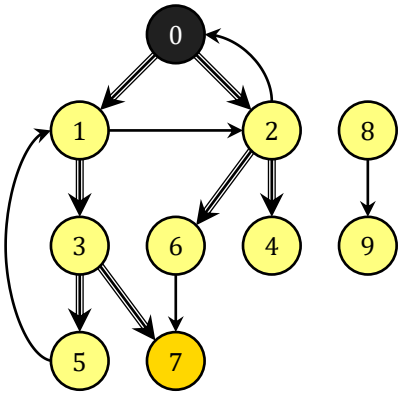
24.4.3. Một vài tính chất của BFS

❋ 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 BFS 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 24-4. Cây BFS

Hình 24-4 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:

```
1  | void BFSVisit(s ∈ V)
2  | {
3  |     Queue = (s); //hàng đợi chỉ gồm một đỉnh s
4  |     d[s] = Time++; //d[s] = 0, cũng là đánh dấu ≠ -1
5  |     do
6  |     {
7  |         u = Queue.front(); Queue.pop(); //Lấy u khỏi hàng đợi
8  |         f[u] = Time++; //Ghi nhận thời điểm duyệt xong u
9  |         Output ← u; //Duyệt xong đỉnh nào in ra luôn đỉnh đó
10 |         for (∀v ∈ adj[u]) //duyệt mọi đỉnh v nối từ u
11 |             if (d[v] == -1) //nếu v chưa thăm
12 |             {
13 |                 Queue.push(v);
14 |                 d[v] = Time++; //Ghi nhận thời điểm thăm đến u, cũng là đánh dấu d[v] ≠ -1
15 |             }
16 |     }
17 |     while (Queue ≠ ∅);
18 | }
19 |
20 | Input → đồ thị G;
21 | for (∀u ∈ V) //Các đỉnh đều chưa thăm: d[.] = -1
22 |     d[u] = -1;
23 | Time = 0;
24 | for (∀u ∈ V) //Xét mọi điểm chưa thăm
25 |     if (d[u] == -1)
26 |         BFSVisit(u);
```

❋ Thời gian thực hiện giải thuật

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

Trong mô hình duyệt đồ thị bằng 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. Cụ thể là, mỗi đỉnh u sẽ tương ứng với thời điểm thăm đến d_u và thời điểm duyệt xong f_u.

Bây giờ ta thử sửa đổi mô hình cài đặt: thay cơ chế đánh dấu thăm đến/chưa thăm đến bằng duyệt xong/chưa duyệt xong:

```
1  | void BFSVisit(s ∈ V)
2  | {
3  |     Queue = (s); //hàng đợi chỉ gồm một đỉnh s
4  |     d[s] = Time++; //Ghi nhận thời điểm thăm tới s
5  |     do
6  |     {
7  |         u = Queue.front(); Queue.pop(); //Lấy u khỏi hàng đợi
8  |         if (f[u] != -1) //u đã duyệt xong, bỏ qua
9  |         f[u] = Time++; //Ghi nhận thời điểm duyệt xong u, cũng là đánh dấu ≠ -1
10 |         Output ← u; //Duyệt xong đỉnh nào in ra luôn đỉnh đó
11 |         for (∀v ∈ adj[u]) //duyệt mọi đỉnh v nối từ u
12 |             if (f[v] == -1) //nếu v chưa duyệt xong
13 |             {
14 |                 Queue.push(v);
15 |                 d[v] = Time++; //Ghi nhận thời điểm thăm đến u
16 |             }
17 |     }
18 |     while (Queue ≠ ∅);
19 | }
20 |
21 | Input → đồ thị G;
22 | for (∀u ∈ V) //Các đỉnh đều chưa duyệt xong: f[.] = -1
23 |     f[u] = -1;
24 | Time = 0;
25 | for (∀u ∈ V) //Xét mọi điểm chưa duyệt xong
26 |     if (f[u] == -1)
27 |         BFSVisit(u);
```

Thứ tự các đỉnh được in ra không thay đổi, tuy nhiên có sự thay đổi trong quy trình xử lý:

- Ở cách cài đặt dùng cơ chế đánh dấu thăm/chưa thăm, mỗi đỉnh được đẩy vào hàng đợi đúng một lần và bị lấy ra đúng một lần
- Ở cách cài đặt dùng cơ chế duyệt xong/chưa duyệt xong, ngoại trừ đỉnh gốc của các cây BFS bị đẩy vào hàng đợi một lần và lấy ra đúng một lần, mỗi đỉnh v không phải gốc của cây BFS có thể bị đẩy vào hàng đợi nhiều lần (tối đa $\deg^-(v)$) lần, hàng đợi có thể có thời điểm phải chứa nhiều hơn n đỉnh (tuy nhiên kích thước hàng đợi không bao giờ vượt quá m).

Điều đặc biệt là trong mô hình cài đặt dùng cơ chế đánh dấu duyệt xong/chưa duyệt xong, nếu ta:

- Thay cấu trúc hàng đợi bởi cấu trúc ngăn xếp
- Lật ngược thứ tự đỉnh trong các danh sách kề

khi đó thuật toán sẽ in ra các đỉnh theo thứ tự duyệt đỉnh DFS. Đây chính là phương pháp khử đệ quy của DFS để cài đặt thuật toán trên các ngôn ngữ không cho phép đệ quy.

Bài tập 24-1

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

Bài tập 24-2

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 24-3

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

Bài tập 24-4

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 24-5

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 $O(mn)$.