NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 5A

LÝ THUYẾT ĐỒ THỊ VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 5

LÝ THUYẾT ĐỒ THỊ VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

LỜI NÓI ĐẦU

Những năm trước khi lập trình VieTeX tôi toàn dùng C/C++ thu thập tài liệu nhiều nhưng không có thời gian để viết lại. Nay muốn viết lại thì sức khỏe không ổn định. Tôi đã cố gắng gom lại thành các tập lập trình theo chủ đề. Nội dung mỗi thuật toán bắt đầu từ lý thuyết đến lập trình bằng C/C++.

Cuốn sách viết ra không dành riêng cho các bạn học tin học, mà các bạ học toán, thầy cô giáo, các bạn thích tìm hiểu về thuật toán. Cũng như tôi bắt đầu có biết gì về lập trình đâu, tự học và chăm chỉ là thành công thôi. Tôi dùng trình biên dịch Dev-C++: https://www.bloodshed.net/
Hiện nay Dev-C++ cải tiến rất nhiều và chạy tốt với môi trường unicode. Những ví dụ trong tài liệu các bạn chép thẳng vào soạn thảo và biên dịch không cần cấu hình trình biên dịch.

Tôi đã làm các quyển sách:

1. Thuật toán và số học.
2. Thuật toán và dữ liệu.
3. Thuật toán xắp xếp
4. Thuật toán tìm kiếm
5. Thuật toán đồ thị,
6. Thuật toán quay lui
7. Thuật toán chia để trị
8. Thuật toán động
9. Thuật toán tham
10. Thuật toán nén
11. Một số đề thi Olympic Tin học.

Cuốn sách dành cho học sinh phổ thông yêu toán, học sinh khá giỏi môn toán, các thầy cô giáo, sinh viên đại học ngành toán, ngành tin học và những người yêu thích Toán - Tin. Trong biên soạn không thể tránh khỏi sai sót và nhầm lẫn mong bạn đọc cho ý kiến.

Hà Nội, ngày 25 tháng 2 năm 2022

Nguyễn Hữu Điển

NHỮNG KÝ HIỆU

Trong cuốn sách này ta dùng những kí hiệu với các ý nghĩa xác định trong bảng dưới đây:

$\mathbb{N}$	tập hợp số tự nhiên
$\mathbb{N}^*$	tập hợp số tự nhiên khác 0
$\mathbb{Z}$	tập hợp số nguyên
$\mathbb{Q}$	tập hợp số hữu tỉ
$\mathbb{R}$	tập hợp số thực
$\mathbb{C}$	tập hợp số phức
$\equiv$	dấu đồng dư
∞	dương vô cùng (tương đương với $+\infty$)
$-\infty$	âm vô cùng
$\emptyset$	tập hợp rỗng
C_m^k	tổ hợp chập k của m phần tử
$\vdots$	phép chia hết
$\not\vdots$	không chia hết
$UCLN$	ước số chung lớn nhất
$BCNN$	bội số chung nhỏ nhất
deg	bậc của đa thức
IMO	International Mathematics Olympiad
APMO	Asian Pacific Mathematics Olympiad

NỘI DUNG

DANH SÁCH CÁC HÌNH

DANH SÁCH CÁC BẢNG

Danh sách chương trình

CHƯƠNG 5
LÝ THUYẾT ĐỒ THỊ VÀ THUẬT TOÁN

"Có hai kết quả có thể xảy ra: Nếu kết quả xác nhận giả thuyết, nghĩa là bạn đã thực hiện một phép đo. Nếu kết quả trái với giả thuyết, thì bạn đã thực hiện một khám phá."

E. Fermi

5.1. Các khái niệm cơ bản

Lý thuyết đồ thị là một nhánh của toán học hiện đại đã có sự phát triển ấn tượng trong vài thập kỷ qua. Đồ thị là một trong những cấu trúc dữ liệu trừu tượng hữu ích nhất trong khoa học máy tính. Nhiều vấn đề từ các lĩnh vực khoa học và thực tiễn khác nhau có thể được mô hình hóa bằng đồ thị và được giải quyết bằng cách sử dụng một thuật toán thích hợp trên đó. Vì lý do này, Chúng ta sẽ dành cả một chương cho họ, và đến cuối cuốn sách này, họ sẽ có mặt trong nhiều bài toán khác. Trước khi đưa ra một số ví dụ, Chúng ta sẽ xác định rõ khái niệm.

Định nghĩa 5.1. Một đồ thị có hướng hữu hạn được gọi là một cặp có thứ tự (V, E), trong đó:

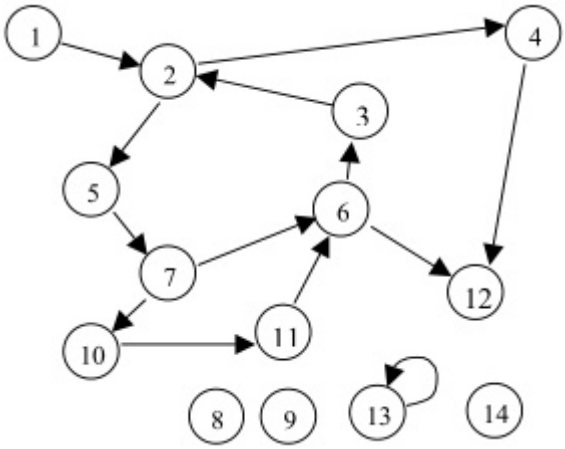
- $v_=\{v_1, v_2, \ldots, v_n\}$ là một tập hữu hạn các đỉnh;
- $E = \{e_1, e_2, \ldots, e_m\}$ là tập hữu hạn các cạnh có định hướng. Mỗi phần tử $e_k \in E(k = 1, 2, \ldots, m)$ là một cặp có thứ tự $(v_i, v_j), v_i, v_j \in V, 1 \leq i, j \leq n$.

Ngoài ra, nếu cho một hàm số $f : E \to R$, phù hợp trên mỗi cạnh e_k trọng số $f(e_k)$ thì đồ thị được gọi là có trọng số. Đôi khi, khi không có nguy cơ mơ hồ, chúng ta sẽ nói chỉ cạnh thay vì cạnh định hướng. Chúng ta sẽ lưu ý rằng một số tác giả sử dụng thuật ngữ cung cho một cạnh có định hướng và cạnh - chỉ cho một cạnh không định hướng.

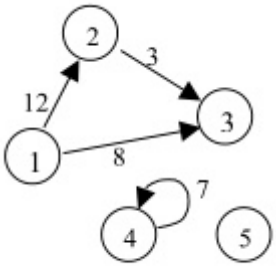
Thông thường, một đồ thị có định hướng được biểu diễn bằng đồ thị trong mặt phẳng bởi một tập hợp các điểm (đường tròn) biểu thị các đỉnh của nó và nối chúng bằng các mũi tên - các cạnh của đồ thị.

Trong Hình 5.1 một đồ thị có 14 đỉnh và 13 cạnh được hiển thị. Chúng ta đã biểu diễn mỗi đỉnh $v_i \in V$ là một đường tròn và chúng ta đặt số của nó - một số tự nhiên duy nhất.

Trong trường hợp điều này sẽ không dẫn đến hiểu lầm bổ sung, thay vì một tập V tùy ý, chúng ta sẽ giả sử rằng V là tập hợp n số tự nhiên đầu tiên. Điều này không làm giảm tính phổ biến của các thuật toán được xem xét và đồng thời dẫn đến một số thuận tiện trong việc triển khai - ví dụ, chúng ta có thể sử dụng các đỉnh làm



Hình 5.1. Đồ thị định hướng.



Hình 5.2. Đồ thị cân định hướng.

chỉ số mảng và các thuật toán khác.

Chúng ta đã biểu diễn mỗi cạnh $(i, j) \in E$ như một mũi tên chỉ từ đỉnh *i*đến đỉnh j. Lưu ý rằng có thể cho phép một cạnh nhô ra và đi vào cùng một đỉnh: ví dụ, đỉnh 13. Các đường cạnh như vậy được gọi là vòng lặp. Nếu đồ thị có trọng số, trọng lượng của mỗi cạnh sẽ được ghi bên cạnh mũi tên tương ứng, như trong Hình 5.2.

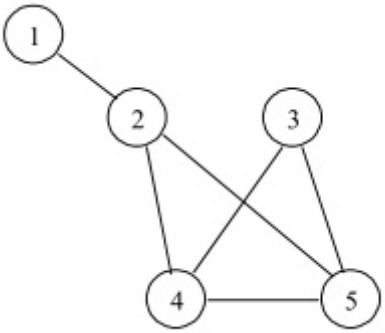
Trong một số trường hợp hiếm hoi, khi xem xét một số bài toán cụ thể, có thể sử dụng các cách lập chỉ mục đỉnh khác (ví dụ: bằng chữ cái Latinh viết thường).

Định nghĩa 5.2. Một *đồ thị vô hướng hữu hạn* được gọi là một cặp có thứ tự (V, E), trong đó:

- $V = \{v_1, v_2, \ldots, v_n\}$ là một tập hữu hạn *các đỉnh*
- $E = \{e_1, e_2, \ldots, e_m\}$ là tập hữu hạn *các cạnh vô hướng*. Mỗi phần tử $e_k \in E (k = 1, 2, \ldots, m)$ là một cặp không có thứ tự $(v_i, v_j), v_i, v_j \in V, 1 \leq i, j \leq n$.

Ngoài ra, nếu hàm $f(i, j)$ được thiết lập, khớp với giá trị nguyên của mỗi cạnh $(i, j) \in E, f(i, j) = f(j, i)$, đồ thị được gọi là *đồ thị không định hướng có trọng số*.

Trong Hình 5.3 một đồ thị không định hướng được hiển thị - các



Hình 5.3. Đồ thị không định hướng

đỉnh được biểu diễn bằng các đường tròn và mỗi cạnh (i, j) bằng một đoạn thẳng.

Đôi khi chúng ta sẽ muốn coi đồ thị không định hướng $G(V, E)$ là một $G'(V, E')$ có hướng. Trên mỗi đồ thị không định hướng có thể so sánh một đồ thị có hướng tương ứng: trên mỗi cạnh $(i, j) \in E$ các cạnh $(i, j) \in E'$ và $(j, i) \in E'$ trong G' được so sánh. Điều này rất hữu ích khi chúng ta muốn áp dụng một thuật toán hợp lệ cho các đồ thị có định hướng và một đồ thị không định hướng được đưa ra trong câu lệnh bài toán.

Hầu như bất kỳ tập hợp đối tượng nào có kết nối xác định giữa chúng đều có thể được biểu diễn dưới dạng đồ thị. Dưới đây là một số ví dụ:

1) Xem xét bản đồ giao thông của Hà Nội và các ngã ba, tư năm. Chúng có thể được biểu diễn dưới dạng các đỉnh của đồ thị và các đường dẫn trực tiếp giữa chúng - dưới dạng các cạnh. Trọng lượng của các cạnh sẽ là độ dài của các đoạn đường. Ví dụ minh họa là Hình 5.4, trong đó đồ thị là vô hướng, và có thể coi là đồ thị có định hướng.

2) Một mạng máy tính có thể được biểu diễn bằng một đồ thị không định hướng trong đó các máy tính là các đỉnh và mỗi cạnh giữa hai đỉnh cho biết rằng các máy tính tương ứng được kết nối trực tiếp với một mạng.

3) Tập hợp các trang web và các liên kết giữa chúng có thể được biểu diễn dưới dạng một đồ thị có định hướng.

Hình 5.4. Bản đồ giao thông thành phố Hà Nội

4) Một số hợp chất hóa học có thể được biểu diễn dưới dạng các đỉnh của đồ thị không định hướng: Mỗi cạnh sẽ cho biết liệu các hợp chất hóa học tương ứng có thể tương tác hay không. Một ví dụ tương tự là phần ngọn tượng trưng cho các loài cá cảnh và phần cạnh cho biết hai loài cá có thể cùng tồn tại hay không.

5) Các giai đoạn trong quá trình sản xuất một sản phẩm có thể được biểu diễn dưới dạng các đỉnh của một đồ thị có định hướng. Các đường gân chỉ ra công đoạn nào trước ai trong quá trình làm ra sản phẩm.

Nhiều ví dụ khác về đồ thị có thể được đưa ra, cũng như các bài toán được xây dựng trên chúng. Ví dụ: chúng ta có thể quan tâm đến con đường ngắn nhất hoặc rẻ nhất giữa hai ngã tư trong ví dụ 1), hoặc thời gian tối thiểu để tạo ra toàn bộ sản phẩm trong ví dụ 5).

Để xem xét đầy đủ tài liệu hơn nữa, cần phải xác định thêm một

số khái niệm từ lý thuyết đồ thị. Người đọc không cần phải cố gắng ghi nhớ tất cả các định nghĩa ngay bây giờ. Chỉ cần anh ta quay lại đây sau khi kiểm tra phần liên quan và nhớ những gì mình cần là đủ.

Định nghĩa 5.3. Một đồ thị có định hướng (vô hướng) $G(V, E)$được đưa ra. Nếu cho phép lặp lại trong tập các cạnh của nó (tức là E là một tập đa) thì G được gọi là một *đa đồ thị*.

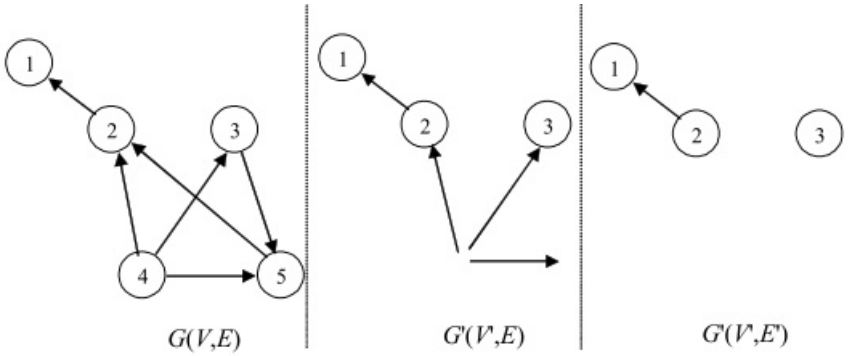
Lưu ý rằng nếu cân đa đồ thị, thì các cạnh khác nhau (i, j) được so sánh với các trọng lượng riêng biệt (nói chung là khác nhau) $f^{(1)}(i, j), f^{(2)}(i, j), ...$

Định nghĩa 5.4. Một đồ thị có định hướng $G(V, E)$ được cho. Hai đỉnh i và $j (i, j \in V)$ được gọi là *kề nhau* nếu ít nhất một trong các cạnh (i, j) và (j, i) thuộc E. Trong trường hợp này, chúng ta cũng nói rằng i và j là điểm đầu các cạnh (i, j) và (j, i). Đối với mỗi cạnh (i, j), đỉnh i được gọi là *trước* của j và j là đỉnh *kế tiếp* của i. Mỗi đỉnh i và j được gọi là *chung cạnh* với cạnh (i, j). Chúng ta nói rằng hai cạnh là ngẫu nhiên khi chúng ngẫu nhiên có cùng một đỉnh. Trong trường hợp đồ thị vô hướng, các số hạng kề đỉnh, diểm cuối cạnh, tỷ lệ đỉnh và cạnh được định nghĩa tương tự.

Định nghĩa 5.5. Một đồ thị có định hướng $G(V, E)$ được cho. Bán độ ở đầu ra của đỉnh $i, i \in V$ được gọi là số cạnh $(i, j), j \in V$. Tương tự, số lượng cạnh $(j, i), j \in V$ được gọi là bán độ ở đầu vào của i. Tổng của nửa giai đoạn đầu vào và nửa giai đoạn đầu ra được gọi là bậc của đỉnh i. Một đỉnh cô lập được gọi, có tung độ bằng 0. Trong một đồ thị không có hướng, tung độ của đỉnh i được gọi là số cạnh (i, j) ngẫu nhiên của nó.

Trong Hình 5.1 tung độ của đỉnh 2 là 4 và đỉnh 14 là biệt lập.

Định nghĩa 5.6. Cho là một đồ thị có định hướng (vô hướng) $G(V, E)$ và $V' \subseteq V$. Nếu tất cả các cạnh (i, j) bị loại trừ khỏi E, sao cho $i \in V'$ hoặc $j \in V'$ và phần còn lại được giữ nguyên, thì chúng ta nói rằng $G'(V', E')$ được quy nạp bởi đồ thị con V' (hoặc chỉ đồ thị con) của G (xem Hình5.5).



Hình 5.5. Được quy nạp bởi tập $V' = \{1, 2, 3\}$ đồ thị con G' của G.

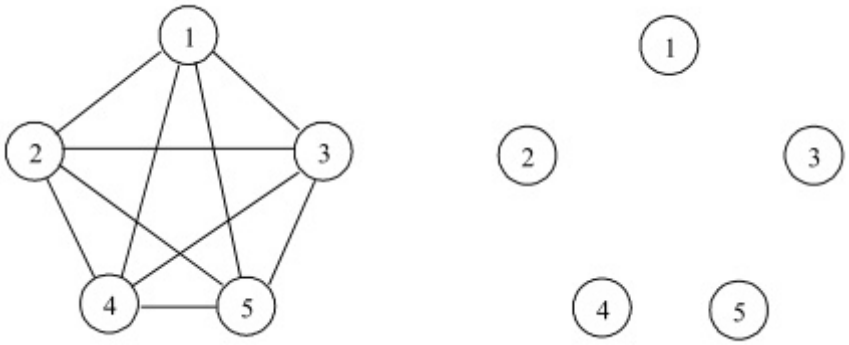
Định nghĩa 5.7. *Một đường đi* trong đồ thị có định hướng (vô hướng) $G(V, E)$ được gọi là dãy các đỉnh $v_1, v_2, ..., v_k$ sao cho mỗi $i = 1, 2, ..., k-1$ được thỏa mãn (vi, trong $i+1$) $\in E$. Các đỉnh v_1 và v_k được gọi là các *điểm cuối của con đường*. Nếu $v_1 = v_k$, đường đi được gọi là *vòng lặp*. Nếu với mỗi $i \neq j (1 \leq i, j \leq k)$ theo sau $v_i \neq v_j$, thì đường đi được gọi là *đơn giản*. Theo đó, nếu $v_1 = v_k$ và tất cả các đỉnh khác khác nhau, thì vòng lặp được gọi là *đơn giản*. Khi một đồ thị chứa ít nhất một chu trình, nó được gọi là *chu trình*, nếu không nó được gọi là *không có chu trình*.

Định nghĩa 5.8. Cho đồ thị $G(V, E)$ và $G'(V, E')$ sao cho với mỗi $i, j \in V, i \neq j$, cặp có thứ tự (i, j) thuộc đúng một trong các tập E' hoặc E. Khi đó G' được gọi là phần bù của G. Một đồ thị không có cạnh được gọi là rỗng (Hình 5.6). Nếu $(i, j) \in E$ với mọi $i, j \in V, i \neq j$ thỏa mãn đối với một đồ thị đã cho thì đồ thị được gọi là hoàn chỉnh.

Lưu ý rằng định nghĩa cuối cùng là đối xứng, tức là nếu G' là phần bù của G thì G là phần bù của G'.

Định nghĩa 5.9. Nếu một đồ thị $G'(V', E')$ với đỉnh t được gây ra bởi G và G' là hoàn chỉnh, chúng ta nói rằng G' là một nhấp chuột của G. T lớn nhất mà G có t -click được gọi là số lần nhấp cho G.

Định nghĩa 5.10. Một đồ thị có định hướng được cho là có tính liên kết yếu nếu cứ hai đỉnh i và j là kết thúc của ít nhất một đường đi (tức là có ít nhất một đường đi từ i đến j hoặc từ j đến i). Khi trong



Hình 5.6. Đồ thị vô hướng hoàn chỉnh và trống rỗng.

một đồ thị có hướng mà cứ hai đỉnh i, j có một đường đi từ i đến j và từ j đến i, thì đồ thị được gọi là liên thông mạnh. Một đồ thị vô hướng được gọi là liên thông nếu có một đường đi giữa mỗi cặp đỉnh i, j của nó.

Định nghĩa 5.11. Đường kính của đồ thị được gọi là số lớn nhất k, sao cho đường đi nguyên tố ngắn nhất (đường có số đỉnh tối thiểu) giữa hai đỉnh $i, j \in V$ bất kỳ chứa ít nhất k đỉnh.

Ví dụ, nếu chúng ta xem các trang web và liên kết như một đồ thị có định hướng, thì đường kính k của nó là số trang tối đa mà chúng ta phải đi qua, bắt đầu từ một trang ngẫu nhiên và đi theo các siêu liên kết để đến bất kỳ trang nào khác trên web. Các nghiên cứu cho thấy "đường kính của Internet" là khoảng 19 [Web-d].

Định nghĩa 5.12. *Thành phần được liên thông mạnh (yếu)* trong một đồ thị có định hướng $G(V, E)$ được gọi là bất kỳ đồ thị con $G'(V', E')$ nào mà nó được đáp ứng:

1) G' là một đồ thị liên thông mạnh (yếu).

2) Không có đồ thị con nào khác được kết nối mạnh (yếu) với $G''(V'', E'')$ của G, sao cho G' là một đồ thị con của G''.

Tương tự, một *thành phần liên thông* được xác định trong một đồ thị không định hướng.

Định nghĩa 5.13. Một đồ thị liên thông vô hướng không có vòng lặp được gọi là cây. Nếu chúng ta chọn thêm một ngọn cây cho gốc, thì cấu trúc kết quả được gọi là cây gốc.

Định nghĩa này là một sự thay thế cho định nghĩa được thảo luận trong **??** định nghĩa của cây).

Định nghĩa 5.14. Cây bao phủ trong đồ thị không định hướng liên thông $G(V, E)$ được gọi là bất kỳ đồ thị con chu trình liên thông $G'(V, E')$ nào của G.

Bài tập

▷ **5.1.** So sánh các định nghĩa **??** và **??**.

5.2. Trình bày và các thao tác đơn giản với đồ thị

Cho đến nay, Chúng ta đã sử dụng hai cách để biểu diễn một đồ thị:

- Trực tiếp theo định nghĩa - qua các tập V và E đã cho.
- Về mặt đồ họa: thông qua tập hợp các điểm (đường tròn) và các kết nối (mũi tên) giữa chúng.

Bản trình bày thứ hai (mặc dù trực quan) không thể áp dụng được và bản trình bày đầu tiên - thường không thuận tiện, đối với quá trình xử lý trên máy tính. Do đó, có nhiều cách khác để trình bày một đồ thị - chúng sẽ là chủ đề của đoạn văn này. Việc lựa chọn cách biểu diễn máy tính thích hợp nhất phụ thuộc vào bài toán cụ thể - tốt nhất là sử dụng một kiểu biểu diễn trong đó các phép toán được thực hiện thường xuyên có độ phức tạp thuật toán thấp hơn (hoặc yêu cầu ít bộ nhớ hơn).

5.2.1. Danh sách các cạnh

Gần nhất với định nghĩa của đồ thị là một biểu diễn với danh sách các cạnh của nó. Chúng ta biểu diễn một đồ thị có định hướng dưới dạng danh sách các cặp có thứ tự (i, j). Trong trường hợp đồ thị vô hướng, các cặp không có thứ tự. Trong một đồ thị có trọng số

(có định hướng), chúng ta xem xét (có thứ tự) các bộ ba (i, j, k), trong đó số thực k là trọng số $f(i, j)$ của cạnh tương ứng. Sau đó, chúng ta có thể nhận ra trên C bằng cách sử dụng mảng float A[3][m], với m là số cạnh của đồ thị.

Rõ ràng, trong cách biểu diễn này, bộ nhớ cần thiết sẽ là $\Theta(m)$, đây là mức độ phức tạp của việc kiểm tra xem hai đỉnh có kề nhau hay không, đây được coi là một trong những phép toán phổ biến nhất. Trong trường hợp Chúng ta sắp xếp các cạnh, Chúng ta sẽ có thể áp dụng tìm kiếm nhị phân (xem **??**) Và do đó giảm độ phức tạp của kiểm tra xuống) $\Theta(\log_2 m)$. Như chúng ta sẽ thấy bên dưới, có những biểu diễn trong đó độ phức tạp cuối cùng nhỏ hơn nhiều: một hàm của n, và thậm chí là một hằng số. Do đó, biểu diễn này được áp dụng chủ yếu cho các đồ thị thưa, nghĩa là, đồ thị có số cạnh tương đối nhỏ: $m \in O(n)$.

5.2.2. Ma trận lân cận, ma trận trọng số

Ma trận lân cận là một trong những cách được sử dụng phổ biến nhất để biểu diễn một đồ thị trong bộ nhớ. Trong đó, một ma trận vuông A [n] [n] được so sánh trên một đồ thị có hướng với n đỉnh. Giá trị của A[i] [j] bằng 1 khi tồn tại cạnh (i, j) và A[i] [j] = 0 - nếu không. Để thuận tiện cho việc thực hiện một số thuật toán, khi cạnh (i, j) không tồn tại mà tồn tại (j, i) thì tại vị trí A[i][j] trong ma trận được viết giá trị -1 (và lần lượt là 1 - của A[j][i]).

Nếu đồ thị là vô hướng, thì A[i] [j] = 1 khi có cạnh (i, j) hoặc (j, i) và A[i] [j] = 0 nếu không. Với một đồ thị không có trọng số, trong trường A[i] [j] không viết 1 mà là số cạnh giữa các đỉnh i và j. Khi đồ thị có trọng số, trọng số $f(i, j)$ được viết ở vị trí A[i] [j], và ma trận A[][] được gọi là *ma trận trọng số*.

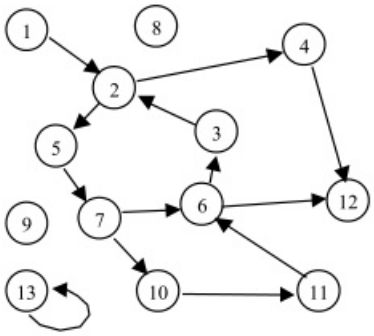
Trong một số bài toán, trọng số cho các đỉnh của đồ thị cũng được xác định. Chúng có thể được viết tại các vị trí A[i][i] cho mỗi đỉnh i của đồ thị. Các trường này từ ma trận sẽ không bị chiếm dụng trừ khi đồ thị không chứa vòng lặp (các cạnh của kiểu (i, i)).

Trong một đồ thị vô hướng, ma trận lân cận là *đối xứng* về đường chéo chính của nó. Trong trường hợp này, có thể áp dụng cách bảo quản cụ thể hơn trong bộ nhớ (ví dụ, tuyến tính hóa, để lưu phần

trùng lặp, chiếm $\frac{n(n-1)}{2}$ thêm ô nhớ [Rakhnev, Garov, Gavrilov-1995]).

Một ưu điểm quan trọng của biểu diễn ma trận tiệm cận là việc kiểm tra sự tồn tại của một cạnh giữa hai đỉnh được thực hiện bằng một phép toán duy nhất. Mặt khác, việc tìm tất cả những người thừa kế tại một đỉnh có độ phức tạp là $\Theta(n)$, bất kể đỉnh đó có bao nhiêu người thừa kế (tức là ngay cả khi đỉnh có một người thừa kế duy nhất, tất cả n lần kiểm tra vẫn sẽ phải thực hiện). Một ví dụ về đồ thị có định hướng và ma trận lân cận tương ứng của nó được cho trong Hình 5.2.2. Lưu ý rằng trong khi việc đánh số đỉnh bằng đồ thị bắt đầu từ 1, trong việc triển khai C, việc lập chỉ mục của các phần tử bắt đầu từ 0. Tính năng cụ thể này phải được tính đến khi xem xét tất cả các triển khai của các thuật toán trong chương này.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	-1	0	-1	1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	-1	0	0	0	0	0	0	0
3	0	-1	0	0	0	0	0	0	0	0	0	1	0
4	0	-1	0	0	0	0	1	0	0	0	0	0	0
5	0	0	1	0	0	0	-1	0	0	0	-1	1	0
6	0	0	0	0	-1	1	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	-1	0	0	0	1	0	0
10	0	0	0	0	0	1	0	0	0	-1	0	0	0
11	0	0	0	-1	0	-1	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1



Hình 5.7. Ma trận lân cận và đồ thị tương ứng của nó.

5.2.3. Danh sách những cạnh kề

Trong biểu diễn này, một danh sách những người thừa kế của nó được lưu giữ cho mỗi đỉnh i bằng đồ thị. Đối với cột trong Hình 5.2.2. bản trình bày sẽ như thế này:

$1 \to \{2\}; 2 \to \{4,5\}; 3 \to \{2\}; 4 \to \{12\}; 5 \to \{7\}; 6 \to \{3,12\}; 7 \to \{6.10\}; 8 \to \{\}; 9 \to \{\}; 10 \to \{11\}; 11 \to \{6\}; 12 \to \{\}; 13 \to \{13\}.$

Danh sách liên kết động thường được sử dụng để triển khai máy tính (xem **??**). Trong đó, độ phức tạp của việc tìm tất cả những người

thừa kế tại một đỉnh nhất định là tuyến tính về số lượng người thừa kế: một lợi thế so với ma trận lân cận, trong đó việc tìm những người thừa kế có độ phức tạp $\Theta(n)$. Tuy nhiên, nhược điểm là việc kiểm tra xem có một đường cạnh giữa hai đỉnh hay không cũng là một sự phức tạp phụ thuộc tuyến tính vào số lượng người thừa kế.

Cũng có thể thực hiện tĩnh biểu diễn này: mảng một chiều với kích thước bằng số lượng người thừa kế cho mỗi đỉnh của đồ thị. Trong trường hợp này, việc kiểm tra xem có một cạnh nào giữa hai đỉnh được thực hiện hiệu quả hơn nhiều hay không - với độ phức tạp logarit. Điều này đạt được bằng cách sắp xếp những người thừa kế và sử dụng tìm kiếm nhị phân.

Rõ ràng là cũng như chúng ta có thể duy trì một danh sách những người thừa kế, vì vậy chúng ta có thể xây dựng một danh sách những người tiền nhiệm, trong trường hợp đó, độ phức tạp của các hoạt động cơ bản sẽ giống nhau. Trong thực tế, một bản trình bày với một danh sách những người đi trước được sử dụng rất hiếm khi.

5.2.4. Ma trận tỷ lệ giữa các đỉnh và cạnh

Trong biểu diễn này, một ma trận có kích thước $n \times m$ được sử dụng. Nếu chúng ta xem xét một đồ thị không định hướng $G(V, E)$, với mỗi cạnh $e_k = (i, j)$, cột k sẽ chứa hai đơn vị - tại vị trí i và j, và trong tất cả các phần tử khác trong cột sẽ được viết 0. Đối với một đồ thị có định hướng của vị trí thứ i được viết là 1 và thứ $j-$thứ - -1, vì tất cả các phần tử khác trong cột là số không. Nếu chúng ta có một vòng lặp (i, i), thì tại vị trí thứ i của cột e_k được viết 2.

Biểu diễn này có thể áp dụng cho một số bài toán cụ thể (ví dụ: tìm số cây bao phủ trong đồ thị, v.v.), nhưng thường rất hiếm khi được sử dụng vì hai lý do. Trước hết, bộ nhớ cần thiết là $\Theta(m.n)$, trong khi ma trận rất thưa thớt. Kiểm tra xem có một cạnh nào giữa hai đỉnh cũng không hiệu quả và có độ phức tạp $\Theta(m)$.

5.2.5. Các thành phần kết nối

Cách trình bày cuối cùng mà chúng ta sẽ xem xét là thông qua các thành phần kết nối. Nó khác với những cái trước ở chỗ đồ thị,

một khi được biểu diễn trong bộ nhớ, sẽ bị "mất", và không thể khôi phục rõ ràng các tập các đỉnh và các cạnh của nó, tức là chúng ta xem xét sự biểu diễn các thuộc tính nhất định của đồ thị. Một điều kiện bổ sung là đồ thị không được định hướng và không có trọng số.

đồ thị được chia thành các thành phần kết nối, với mỗi tập hợp được ánh xạ thành một tập hợp. Ưu điểm là dễ dàng kiểm tra xem hai đỉnh có được nối với nhau bằng một đường dẫn hay không - điều này được thực hiện nếu và chỉ khi chúng thuộc cùng một tập hợp.

Mảng một chiều A[n] có thể được sử dụng để triển khai máy tính, mảng này chấp nhận các giá trị từ 1 đến n. Và [i] có giá trị k t.s.t.k đỉnh thứ i của đồ thị thuộc thành phần thứ k của kết nối. Biểu diễn này cho phép duy trì một đồ thị khổng lồ (chỉ bộ nhớ cần thiết $\Theta(n)$), mà nó có thể kiểm tra xem hai đỉnh có được nối với nhau bằng một đường dẫn hay không với một phép so sánh duy nhất: liệu A[i]có bằng A[j] , hay không.

Ma trận khả năng tiếp cận (mà chúng ta sẽ thảo luận lại trong 5.4.2) Là một cách khác để biểu diễn các thuộc tính đồ thị thông qua các thành phần kết nối. Tuy nhiên, nó kém hiệu quả hơn nhiều so với những cách được mô tả ở trên. Khi trình bày một đồ thị có ma trận khả năng truy cập, một mảng hai chiều A[n][n] được sử dụng, trong đó giá trị của A[i][j] là 1 nếu có một đường đi giữa các đỉnh i và j, và 0 - nếu không.

5.2.6. Xây dựng và các hoạt động đơn giản với đồ thị

Các thao tác chính liên quan đến việc xây dựng và sửa đổi đồ thị như sau:

- tạo một đồ thị rỗng.
- thêm / bớt đỉnh.
- thêm / bớt một cạnh.

Sau đây là các hoạt động được đề cập trong các điểm trước:

- kiểm tra sự tồn tại của một đỉnh
- kiểm tra sự tồn tại của một cạnh
- tìm những người thừa kế của một đỉnh

Chúng ta sẽ không chỉ rõ cách thực hiện các thao tác này cho từng loại bản trình bày. Ví dụ, Chúng ta sẽ chỉ ra một cách triển khai cho một đồ thị được biểu diễn bằng một ma trận lân cận (ma trận trọng số).

```
/*Số đỉnh tối  đa trong đồ thị*/
#define MAXN 200

/*  Số đỉnh trong đồ thị*/
unsigned n;

/*Ma trận trọng số của đồ thị  */
int A[MAXN][MAXN];

/*  Sự thay đổi  trọng lượng của đỉnh i*/
A[i][i] = k;

/*  Thêm một cạnh có trọng số k nối các đỉnh i và j*/
A[i][j] = k;
A[j][i] = k;  /*nếu đồ thị  không có định hướng*/

/*  Loại bỏ cạnh nối các  đỉnh i  và j*/
A[i][j] = 0;

/*  Kiểm tra một cạnh giữa các đỉnh i và j*/
if (A[i][j] != 0)  {  /*  có cạnh */;  } else {  /*  không có cạnh */;  }

/*Tìm tất  cả  những cạnh thừa kế ở đầu*/
for (k = 0;  k < n;  k++) if (k != i)  {
if (A[i][k]  != 0)  {  /*đỉnh k là  đỉnh kế  tiếp của i  */  ;  }
}
```

Trước khi kết thúc, chúng ta sẽ so sánh mức độ phức tạp của các thao tác trong các màn trình diễn khác nhau. Trong mỗi bài toán, chúng ta sẽ cố gắng trình bày trong đó các thao tác được thực hiện thường xuyên nhất có độ phức tạp thấp nhất. Có thể trình bày một đồ thị theo nhiều cách tại một thời điểm nếu chúng ta có đủ bộ nhớ và khi điều này dẫn đến việc triển khai thuật toán đã chọn hiệu quả hơn.

Bảng 5.1 có nghĩa là không thể khôi phục duy nhất đồ thị về đồ

Bài thuyết trình Thao tác	Ma trận cạnh kề	Danh sách cạnh thừa kế	Danh sách các cạnh	Ma trận liên thuộc	Thành phần liên thông
Thêm cạnh	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 m)$	$\Theta(1)$	$\Theta(n.\log_2 n)$
Xóa cạnh	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 m)$	$\Theta(m)$	$\times$
Kiểm tra sự tồn tại của cạnh	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 m)$	$\Theta(m)$	$\times$
Tìm cạnh thừa kế đỉnh cao	$\Theta(n)$	$\Theta(d_i)$	$\Theta(n+\log_2 m)$	$\Theta(m)$	$\times$
Kiểm tra kết nối của hai đỉnh với một con đường	$\Theta(n^2)$	$\Theta(n+m)$	$\Theta(n.\log_2 m)$	$\Theta(m^2)$	$\Theta(1)$
Bộ nhớ cần thiết	$\Theta(n^2)$	$\Theta(m)$	$\Theta(m)$	$\Theta(n.m)$	$\Theta(n)$

Bảng 5.1. So sánh mức độ phức tạp của các hoạt động trong các buổi biểu diễn khác nhau.

thị đã chọn trình bày và không thể thực hiện thao tác tương ứng.

Bài tập

▷ **5.2.** Xem xét làm thế nào để đạt được độ phức tạp trong Bảng 5.1 không được đề cập trong đoạn này và theo giả định nào: Ví dụ, để đạt được độ phức tạp $\Theta(\log_2 m)$ để kiểm tra sự tồn tại của một cạnh trong danh sách các cạnh, chúng ta giả sử rằng chúng ta sử dụng tìm kiếm nhị phân trong một danh sách tĩnh các cạnh được sắp xếp theo số đỉnh.

5.3. Trình duyệt trên đồ thị

Bằng cách duyệt qua một đồ thị không định hướng (có định hướng), chúng ta sẽ hiểu được một lần truy cập liên tiếp (xem) mỗi

đỉnh của đồ thị đúng một lần. *Chiến lược trình duyệt xác định thứ tự* mà các đỉnh sẽ được xem xét.

Có hai chiến lược cơ bản để duyệt qua một đồ thị (tùy thuộc vào hai phần tiếp theo), được gọi là duyệt theo chiều rộng và duyệt theo chiều sâu. Tầm quan trọng đặc biệt của hai thuật toán này được xác định bởi:

- Phạm vi rộng của các bài toán mà họ thấy ứng dụng.
- Khả năng giải quyết các vấn đề thuật toán phức tạp bằng cách áp dụng các sửa đổi nhỏ của các thuật toán cơ bản.
- Độ phức tạp thuật toán tốt, đạt được trong hầu hết các trường hợp.

5.3.1. Duyệt theo chiều rộng

Duyệt theo chiều rộng của đỉnh i sẽ được gọi là chiến lược sau để duyệt qua đồ thị: chúng ta bắt đầu từ đỉnh i, xem xét tất cả các lân cận trực tiếp của nó và chỉ sau đó chuyển sang duyệt xa hơn (duyệt theo chiều rộng của từng lân cận của nó). Bằng cách này, đạt được sự đi ngang tuần tự của các "mức", bắt đầu từ đỉnh bắt đầu, trong khi đi qua tất cả các đỉnh của đồ thị, có thể truy cập được bởi i. Duyệt theo chiều rộng của đồ thị được gọi là đi qua tất cả các đỉnh của nó theo cách được mô tả - tức là liên tiếp chọn một đỉnh bắt đầu (ngẫu nhiên) cho đến khi tất cả các đỉnh của đồ thị được duyệt qua.

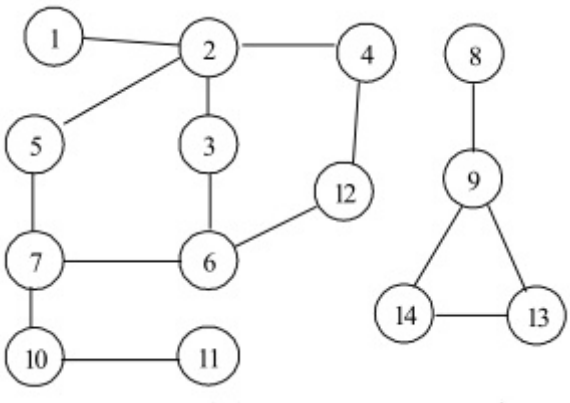
Từ bây giờ, chúng ta sẽ sử dụng chữ viết tắt $BFS(i)$ (Breadth-First-Search) để biểu thị chức năng duyệt theo chiều rộng của đỉnh i.

Để làm ví dụ, chúng ta sẽ áp dụng chiến lược được mô tả trên đồ thị trong Hình 5.8.

Nếu chúng ta chọn 1 làm đỉnh bắt đầu, kết quả duyệt sẽ như sau:
$BFS(1)$:
cấp độ 1: 1
cấp độ 2: 2
cấp độ 3: 3, 4, 5
cấp độ 4: 7, 6, 12
cấp độ 5: 10

cấp độ 6:11
Nếu thay vì 1, chúng ta chọn 3 cho đỉnh ban đầu, khi thực thi $BFS(3)$, chúng ta nhận được:
$BFS(3)$:
cấp độ 1: 3
cấp độ 2: 2, 6
cấp độ 3: 1, 4, 5, 7, 12
cấp độ 4:10
cấp độ 5:11

Một số ứng dụng có thể có của việc duyệt theo chiều rộng là đáng chú ý ngay lập tức. Ví dụ, tìm số lượng đỉnh tối thiểu có liên quan giữa hai đỉnh hoặc tìm các thành phần kết nối của đồ thị. Chúng ta sẽ xem xét những điều này và các ví dụ khác xa hơn một chút.



Hình 5.8. Đồ thị không định hướng.

duyệt theo chiều rộng được sử dụng gián tiếp trong các tác vụ khác khi yêu cầu khoảng cách tối thiểu. Ví dụ, phương pháp sóng (xem Bài toán 5.110), áp dụng để tìm đường đi nhỏ nhất giữa hai ô của ma trận, v.v., cũng có thể được coi là đi ngang qua một loại đồ thị đặc biệt theo chiều rộng.

Thực hiện duyệt theo chiều rộng từ một đỉnh nhất định

Chúng ta sẽ biểu diễn đồ thị bằng ma trận lân cận và chúng ta sẽ sử dụng cách biểu diễn này trong hầu hết các bài toán từ đồ thị. Nó

đủ hiệu quả cho hầu hết các tác vụ và đồng thời không làm giảm khả năng đọc của mã với phân bổ bộ nhớ động, triển khai nhiều cấu trúc dữ liệu phức tạp và hơn thế nữa.

Tất nhiên, cả trong quá trình duyệt và các bài toán được thảo luận bên dưới, chúng ta sẽ cố gắng trình bày cho người đọc các giải pháp luôn tốt nhất (theo quan điểm thuật toán) được biết đến cho đến nay.

Trong quá trình duyệt theo chiều rộng, chúng ta phải tìm những người thừa kế của một đỉnh i cho trước. Từ Bảng 5.1, có thể thấy rằng nếu chúng ta sử dụng ma trận lân cận, độ phức tạp sẽ là $\Theta(n)$. Điều này là do đối với mỗi đỉnh n của đồ thị, nó được kiểm tra xem nó có phải là người thừa kế của i hay không (để so sánh, trong trường hợp thực hiện với một danh sách những người thừa kế, chúng ta có thể lấy trực tiếp những người thừa kế của i bằng cách duyệt qua danh sách). Do đó, tổng độ phức tạp của duyệt là $\Theta(m+n)$ - khi trình bày với danh sách lân cận và $\Theta(n^2)$ - khi trình bày với ma trận lân cận.

Chúng ta sẽ duy trì một hàng đợi trong đó ban đầu chỉ có đỉnh xuất phát. Sau đó, trong khi có ít nhất một đỉnh trong hàng đợi, chúng ta thực hiện như sau: chúng ta lấy ra đỉnh ở đầu hàng đợi, nhìn vào nó và thêm vào hàng đợi tất cả những người thừa kế chưa được kiểm tra cho đến nay của nó. Chúng ta sẽ đánh dấu các ngọn là đã truy cập vào thời điểm chúng ta thêm chúng vào hàng đợi:

Thuật toán duyệt theo bề rộng

```
BFS(i)
{ Chúng ta tạo một hàng đợi trống;
  Chúng ta thêm vào hàng đợi đỉnh i;
  for (k = 1,2,...,n) used[k] = 0;
  used[i] = 1;
  while (Hàng đợi không trống) {
    p = Chúng ta xóa phần tử ở đầu hàng đợi;
    Phân tích đỉnh p;
   for (với mọi đỉnh kề j của p)
     if (0 == used[j]) { /*nếu đỉnh j không bị bỏ qua*/
     Thm vào hàng đợi đỉnh j;
     used[j] = 1; /* chúng ta đánh dấu j là bỏ qua*/
   }
```

```
    }
}
```

Nhận thức đầy đủ sau đây. Dữ liệu đầu vào được sử dụng trong quá trình triển khai bên dưới được đặt làm hằng số ở đầu chương trình và dành cho cột được hiển thị trong Hình 5.8 Số đỉnh của đồ thị là n và A[MAXN][MAXN] là ma trận lân cận của nó. Đỉnh bắt đầu cho việc duyệt được đặt với hằng số v.

Trong ví dụ minh họa, đồ thị là không có hướng, nhưng chương trình dưới đây sẽ áp dụng chính xác thuật toán trong trường hợp đồ thị có hướng.

Chương trình 5.1. Duyệt theo chiều rộng(501bfs.c)

```
#include <stdio.h>
/* Số đỉnh tối  đa trong đồ thị  */
#define MAXN 200
/* Số đỉnh của đồ thị  */
const unsigned n = 14;
/* Duyệt theo chiều rộng từ đỉnh  v */
const unsigned v = 5;
/* Ma trận kề của đồ thị  */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};
char used[MAXN];
/* Chức năng duyệt chiều rộng từ đỉnh đã cho */
void BFS(unsigned i)
```

```
{ unsigned k, j, p, queue[MAXN], currentVert, levelVertex,
    queueEnd;
  for (k = 0; k < n; k++) queue[k] = 0;
  for (k = 0; k < n; k++) used[k] = 0;
  queue[0] = i; used[i] = 1;
  currentVert = 0; levelVertex = 1; queueEnd = 1;
  while (currentVert < queueEnd) { /* cho đến khi hàng đợi trống rỗ
    ng */
    for (p = currentVert; p < levelVertex; p++) {
    /* p - lấy phần tử tiếp theo từ hàng đợi */
    printf("%u ", queue[p]+1);
    currentVert++;
    /* cho mỗi thừa kế bắt buộc j trong số queue[p] */
    for (j = 0; j < n; j++)
      if (A[queue[p]][j] && !used[j]) {
        queue[queueEnd++] = j;
        used[j] = 1;
      }
    }
    printf("\n");
    levelVertex = queueEnd;
  }
}

int main() {
  printf("Duyệt theo chiều rộng từ đỉnh %u: \n", v);
  BFS(v-1);
  return 0;
}
```

Kết quả thực hiện chương trình:

```
duyệt theo chiều rộng từ đỉnh 5:
5
2 7
1 3 4 6 10
12 11
```

Bài tập

▷ **5.3.** Chương trình trên thực hiện một thuật toán để duyệt đồ thị theo chiều rộng từ một đỉnh cho trước. Trong những trường hợp nào, được thực hiện cho một đỉnh tùy ý, nó sẽ đi qua tất cả các đỉnh của đồ thị?

▷ **5.4.** Chương trình nên được sửa đổi như thế nào để, trong trường hợp các đỉnh vẫn cần thiết, duyệt theo chiều rộng mới được thực hiện trên chúng?

5.3.2. Duyệt theo chiều sâu

Tìm kiếm chiều sâu (DFS). Tìm kiếm theo độ sâu - nó là một phần không thể thiếu của một số thuật toán phức tạp hơn là một ý tưởng cơ bản ở một trong những các phương pháp cơ bản để giải quyết các bài toán toàn diện - tìm kiếm có trả lại.

Trong khi duyệt theo chiều rộng kiểm tra các đỉnh của đồ thị một cách tuần tự theo các cấp, duyệt theo chiều sâu từ một đỉnh có xu hướng "đi xuống" sâu nhất có thể trong quá trình duyệt. Thuật toán để duyệt đồ thị theo chiều sâu được mô tả một cách đệ quy dễ dàng nhất: chúng ta bắt đầu từ đỉnh ban đầu đã chọn $i \in V$, đánh dấu nó là đã thăm và tiếp tục thu thập dữ liệu một cách đệ quy theo chiều sâu cho từng phần tử kế tiếp không được kiểm tra của nó, tức là chức năng duyệt chuyên sâu $(DFS(i))$ trông giống như sau:

1) Chúng ta xem xét i.

2) Chúng ta đánh dấu i là bỏ qua.

3) Đối với mỗi sự cố với i cạnh $(i, j) \in E$ Nếu j là đỉnh cần thiết của đồ thị, chúng ta thực hiện $DFS(j)$ một cách đệ quy.

Chúng ta sẽ áp dụng DFS cho ví dụ trong Hình 5.8. Trong trường hợp có nhiều hơn một cạnh là ngẫu nhiên với đỉnh được đề cập (bước 2), Chúng ta sẽ tiến hành tuần tự từ các đỉnh có số nhỏ hơn đến các đỉnh có số lượng lớn hơn. Do đó kết quả của $DFS(1)$ sẽ là: $1, 2, 3, 6, 7, 5, 10, 11, 12, 4$ và của $DFS(3)$: $3, 2, 1, 4, 12, 6, 7, 5, 10, 11$.

Chương trình duyệt sẽ được thực hiện trực tiếp như mô tả ở trên (thông qua hàm đệ quy $DFS(i)$). Trong mảng Boolean used[] húng ta khởi tạo nó bằng các số không, và khi truy cập vào đỉnh i, chúng ta thực hiện phép gán used[i] = 1.

Việc nhận ra như sau:

Chương trình 5.2. Duyệt theo chiều sau(502dfs.c)

```
#include <stdio.h>
/ * Số đỉnh tối  đa trong cột  * /
#define MAXN 200
/ * Số đỉnh trong cột  * /
const trái  dấu n = 14;
/ * Duyệt theo chiều sâu với đỉnh đầu v * /
const trái  dấu v = 5;
/ * Ma trận cạnh kề của đồ thị  * /
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};

char used[MAXN];
/ * Duyệt theo chiều sâu từ đỉnh * /
void DFS(unsigned i)
{ unsigned k;
  used[i]  = 1;
  printf("%u ", i+1);
  for (k = 0;  k < n;  k++)
    if (A[i][k]  && !used[k]) DFS(k);
}

int main() {
  unsigned k;
  for (k = 1;  k < n;  k++) used[k] = 0;
```

```
    printf("Duyệt theo chiều sâu từ đỉnh %u: \n", v);
    DFS(v-1);
    printf("\n");
    return 0;
}
```

Kết quả thực hiện chương trình:

```
duyệt độ sâu từ đỉnh 5:
5 2 1 3 6 7 10 11 12 4
```

Bảng 5.2 so sánh mức độ phức tạp của việc duyệt qua đồ thị trong một số kiểu biểu diễn (độ phức tạp giống nhau khi duyệt theo chiều rộng và chiều sâu). Chúng ta để nó cho người đọc để tìm hiểu xem chúng diễn ra như thế nào.

	Ma trận lân cận	Danh sách những cạnh kề	Danh sách các cạnh
Duyệt đồ thị theo *DFS* hoặc *BFS*	$\Theta(n^2)$	$\Theta(n+m)$	$\Theta(n.m)$

Bảng 5.2. Độ phức tạp của việc duyệt qua một đồ thị trong các biểu diễn khác nhau.

Trước khi kết thúc đoạn này, chúng ta sẽ xem xét ngắn gọn cách chúng ta có thể tìm thấy một cây bao phủ tùy ý trong đồ thị bằng cách sử dụng tìm kiếm theo chiều sâu (hoặc tìm kiếm theo chiều rộng, tương ứng) (xem Định nghĩa 5.14).

Gọi $G(V,E)$ là một đồ thị vô hướng liên thông. Chúng ta sẽ xây dựng cây bao phủ $T(V,D)$ theo thuật toán sau:

1) Ban đầu T là một đồ thị trong đó các cạnh không tham gia, tức là ta khởi tạo $D=\emptyset$.

2) Thực hiện duyệt theo độ sâu trong G. Đối với mỗi lần chuyển đổi đệ quy trong quá trình duyệt từ đỉnh i đến lân cận j yêu cầu của nó, chúng ta thêm cạnh (i,j) vào D.

Dễ dàng chứng minh rằng các điều kiện của định nghĩa cây bao phủ sẽ được đáp ứng: đồ thị con kết quả được kết nối (vì G được kết

nối, duyệt chiều sâu / chiều rộng sẽ "đạt đến" tất cả các đỉnh của đồ thị) và không theo chu trình (mỗi đỉnh được duyệt) được xem xét nhiều nhất một lần).

Bài tập

▷ **5.5.** Chứng minh các kết quả trong Bảng 5.2.

5.4. Đường dẫn, chu trình và dòng chảy tối ưu trong đồ thị

Một trong những bài toán đồ thị phổ biến nhất là tìm đường đi tối ưu. Nếu chúng ta xem lại ví dụ 1) từ đầu chương và giải thích các đỉnh của đồ thị là các khu định cư và các đường nối chúng dưới dạng các đường phố (đường trực tiếp), một bài toán thực tế là tìm một con đường có độ dài tối thiểu giữa hai khu định cư. Ngược lại, nếu việc ghé thăm mỗi đường phố (hoặc ngôi làng) mang lại lợi nhuận nhất định, thì mục tiêu của chúng ta có thể là tìm ra con đường tối đa (tức là con đường mà chúng ta sẽ thu được nhiều nhất) giữa hai điểm. Nói chung, các điều kiện cho tính tối ưu của tuyến đường mà chúng ta sẽ tìm kiếm có thể được xác định theo nhiều cách khác nhau. Ví dụ, có thể cần phải áp dụng các hạn chế bổ sung (ví dụ: các điều khoản); được yêu cầu đến thăm từng ngôi làng chính xác một lần, hoặc băng qua từng con phố chính xác một lần, v.v. Tuy nhiên, cần xác định rõ hai điều:

- *cách ước tính một tuyến đường* (ví dụ - dưới dạng tổng trọng lượng của các cạnh liên quan đến nó)
- *tiêu chí tối ưu của lộ trình* (ví dụ: chúng ta muốn giảm thiểu số tổng trên)

Như một sự tiếp tục tự nhiên của việc tìm kiếm các đường đi tối ưu trong một đồ thị, chúng ta sẽ xem xét việc tìm kiếm các chu trình (tối ưu) trong một đồ thị (nhớ lại rằng một chu trình là một đường mà đỉnh đầu và đỉnh cuối cùng trùng nhau). Có một số kiểu chu trình đáng chú ý mà chúng ta sẽ đặc biệt chú ý.

Chúng ta sẽ kết thúc đoạn này với bài toán tìm dòng chảy cực đại trong đồ thị - một bài toán cơ bản được tìm thấy ứng dụng trong một số lượng lớn các bài toán thực tế.

5.4.1. Các ứng dụng trực tiếp của thuật toán duyệt

- đường đi ngắn nhất giữa hai đỉnh theo số đỉnh

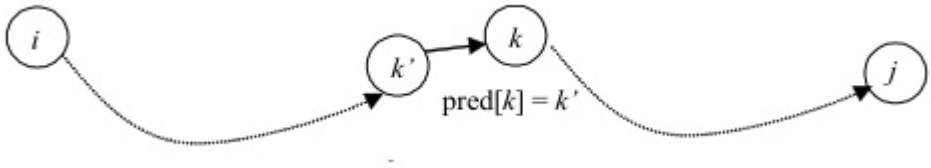
Cho đồ thị $G(V, E)$ và hai đỉnh i và j của nó. Chúng ta tìm một đường đi trong G có đầu là đỉnh i và cuối là đỉnh j, có độ dài tối thiểu trong số các đỉnh tham gia vào nó. Sau khi thực hiện duyệt theo chiều rộng, chúng ta có thể dễ dàng biên dịch một thuật toán để tìm một đường đi tối thiểu như vậy: Chúng ta chạy $BFS(i)$ và nếu ở bất kỳ bước nào chúng ta đạt đến j, nó sẽ ngay lập tức theo sau rằng có một đường đi giữa hai đỉnh, đồng thời chúng ta cũng có một con đường tối thiểu cụ thể.

Để làm ví dụ, chúng ta sẽ xem xét đồ thị trong Hình 5.8. Hãy tìm đường đi từ đỉnh 1 đến đỉnh 10. Sau đó, kết quả của việc thực hiện $BFS(1)$ sẽ là:
cấp độ 1: 1
cấp độ 2: 2
cấp độ 3: 3, 4, 5
cấp độ 4: 7, 6, 12
cấp độ 5: 10

Khi chúng ta đã đạt đến đỉnh cuối cùng j (trong trường hợp này là $j = 10$), rõ ràng độ dài của đường đi nhỏ nhất là mức mà chúng ta đã đi qua j, trừ đi một. Tuy nhiên, làm thế nào để tìm và in các đỉnh trung gian tham gia vào đường? Vấn đề dễ dàng được giải quyết nếu ở mỗi cấp độ chúng ta có tiền thân của đỉnh được duyệt, tức là đỉnh từ lần duyệt lặp lại trước đó, từ đó chúng ta thêm đỉnh hiện tại làm hàng xóm ngay lập tức. Với mục đích này, chúng ta sẽ nhập mảng pred[]: trong đó, chúng ta sẽ giữ các giá trị trước của mỗi đỉnh bỏ qua. Đối với đỉnh ban đầu i, pred[i] chúng ta sẽ khởi tạo với giá trị -1. Do đó, pred[k] sẽ chứa đỉnh mà từ đó chúng ta đã chuyển đến k

Việc phục hồi các đỉnh trung gian tham gia vào đường đi nhỏ nhất từ i đến j được thực hiện theo sơ đồ sau:

```
while (j != i) {
print(j);
j = pred[j];
```



Hình 5.9. Tiết kiệm một đường đi.

```
}
print(j);
```

Đoạn mã giả trên sẽ in ra đường dẫn tối thiểu theo thứ tự ngược lại (tức là từ j đến i). Để có được đường dẫn một cách chính xác, chúng ta có thể viết các đỉnh trong một mảng, sau đó chúng ta có thể in theo thứ tự ngược lại. Do đó, chúng ta hoàn toàn sử dụng một ngăn xếp - trước tiên hãy thêm tất cả các phần tử theo thứ tự, sau đó tắt và in. Cách sau là một cách cổ điển để "đảo ngược kết quả" và được thực hiện một cách thanh lịch nhất với đệ quy (xem cách triển khai bên dưới - hàm printPath). Độ phức tạp của thuật toán là $\Theta(n+m)$. Triển khai đầy đủ như sau:

Chương trình 5.3. Đường đi ngắn nhất(503bfsminw.c)

```
#include <stdio.h>
#define MAXN 200 /* Số đỉnh tối đa trong đồ thị*/
/*  Số đỉnh trong đồ thị  */
const unsigned n = 14;
const unsigned sv = 1; /*  Đỉnh bắt đầu */
const unsigned ev = 10; /*Đỉnh kết thúc */
/*  Ma trân kề của đồ thị  */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
```

```
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};

int pred[MAXN];
char used[MAXN];

/*Duyệt chiều rộng của đỉnh trong khi giữ nguyên giá trị trước*/
void BFS(unsigned i)
{ unsigned queue[MAXN];
  unsigned currentVert, levelVertex, queueEnd, k, p, j;
  for (k = 0; k < n; k++) queue[k] = 0;
  queue[0] = i; used[i] = 1;
  currentVert = 0; levelVertex = 1; queueEnd = 1;
  while (currentVert < queueEnd) { /* Khi hàng đợi khác rỗng */
    for (p = currentVert; p < levelVertex; p++) {
    /* p - lấy phần tử tiếp theo từ hàng đợi */
    currentVert++;
    /* với mỗi đỉnh kề bắt buộc j trong số queue[p] */
    for (j = 0; j < n; j++)
      if (A[queue[p]][j] && !used[j]) {
        queue[queueEnd++] = j;
        used[j] = 1;
        pred[j] = queue[p];
      }
    }
    levelVertex = queueEnd;
  }
}

/* In các đỉnh của đường đi nhỏ nhất và trả về độ dài của nó*/
unsigned printPath(unsigned j)
{ unsigned count = 1;
  if (pred[j] > -1) count += printPath(pred[j]);
  printf("%u ", j + 1); /* In một đỉnh khác của đường dẫn được tìm
     thấy */
  return count;
```

```
}

void solve(unsigned start, unsigned end)
{ unsigned k;
  for (k = 0; k < n; k++) { used[k] = 0; pred[k] = -1; }
  BFS(start);
  if (pred[end] > -1) {
    printf("Đường được tìm thấy: \n");
    printf("\nĐộ dài của đường là %u.\n", printPath(end));
  }
  else
    printf("Không có đường đi giữa hai đỉnh! \n");
}

int main() {
  solve(sv-1, ev-1);
  return 0;
}
```

Kết quả thực hiện chương trình:

```
Đường dẫn được tìm thấy là:
1 2 5 7 10
Con đường là 5.
```

Bài tập

▷ **5.6.** Sửa đổi chức năng duyệt BFS() để nó ngừng hoạt động khi đạt đến mức cao nhất cuối cùng.

- Kiểm tra xem đồ thị có tuần hoàn không

Sử dụng thuật toán thu thập thông tin chuyên sâu, chúng ta sẽ xây dựng một thuật toán để kiểm tra xem một đồ thị vô hướng có phải là theo chu trình hay không. Thuật toán để kiểm tra tính chu trình bao gồm những điều sau: Thực hiện thu thập thông tin sâu. Nếu tại bất kỳ bước nào của quá trình thu thập thông tin, nó chỉ ra rằng đỉnh i được đề cập có một hàng xóm mà chúng ta đã đi qua và khác với đỉnh i trước đó, thì đồ thị đó có chứa một vòng lặp.

Chúng ta sẽ thực hiện thuật toán bằng cách sửa đổi hàm DFS() (vì tham số của nó, chúng ta sẽ thêm `int` parent - hàm mẹ ở trên cùng

mà chúng ta hiện đang ở). Nếu chúng ta đạt từ đỉnh i hiện tại đến đỉnh mà chúng ta đã ở đó, không phải là giá trị gốc, thì vòng lặp đã đóng. Chúng ta sẽ đánh dấu các đỉnh đã thăm như bình thường trong used[] đã sử dụng, và trong hàm chính, chúng ta sẽ gọi DFS(i, -1) cho mỗi đỉnh i, điều này vẫn chưa được xem xét cho đến nay. Giá trị -1 cho gốc có nghĩa là đỉnh này là đỉnh đầu tiên thu thập thông tin thành phần kết nối tương ứng và do đó không có giá trị chính.

Chương trình 5.4. Đường đi ngắn nhất(504formcycl.c)

```
#include <stdio.h>
/* Số đỉnh tối  đa trong đồ thị  */
#define MAXN 200
/* Số đỉnh trong đồ thị  */
const unsigned n = 14;
/* Ma trận kề của đồ thị  */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};
char used[MAXN], cycl;

/* Sửa đổi DFS */
void DFS(unsigned i, int parent)
{ unsigned k;
  used[i] = 1;
  for (k = 0; k < n; k++) {
    if (cycl) return;
    if (A[i][k]) {
```

```
        if (used[k] && k != parent) {
          printf("Đồ thị có tính chu trình! \n");
          cycl = 1;
          return;
        }
        else if (k != parent)
          DFS(k, i);
      }
    }
}

int main() {
  unsigned k, i;
  for (k = 0; k < n; k++) used[k] = 0;
  cycl = 0;
  for (i = 0; i < n; i++) {
    if (!used[i]) DFS(i, -1);
    if (cycl) break;
}
if (0 == cycl)
    printf("đồ thị là một cây (không chứa vòng lặp)!\n");
  return 0;
}
```

Kết quả thực hiện chương trình:

```
Đồ thị là tuần hoàn!
```

Độ phức tạp của thuật toán là $\Theta(n+m)$, và khi biểu diễn đồ thị bằng ma trận lân cận, như trong cách triển khai được đề xuất, nó là $\Theta(n^2)$.

Bài tập

▷ **5.7.** Chương trình trên có hoạt động trong trường hợp đồ thị có định hướng không?

▷ **5.8.** Bạn có thể thực hiện những sửa đổi nào trong trường hợp đồ thị có định hướng? Mã được đơn giản hóa hay phức tạp?

- Tìm tất cả các đường đi đơn giản giữa hai đỉnh

Trong phần này, chúng ta sẽ xem xét thuật toán thu thập thông tin chuyên sâu. Trong *DFS* từ phần 5.3.2, khi đỉnh mà chúng ta đang xem xét hiện có nhiều hơn một người kế nhiệm, chúng ta đã chọn tiếp tục với đỉnh có số thấp nhất. Việc thu thập thông tin sẽ đúng nếu chúng ta chọn tiếp tục với đỉnh có số lượng lớn nhất, cũng như theo bất kỳ cách sắp xếp nào khác của các đỉnh. Chúng ta sẽ thấy trong giây lát quan sát này sẽ giúp chúng ta xây dựng một thuật toán tìm tất cả các đường đi có thể có giữa hai đỉnh như thế nào.

Khi tìm kiếm tất cả các đường đi từ i đến j, chúng ta phải áp dụng tính hết hoàn toàn (có thể thực hiện bằng cách quay lại tìm kiếm - xem Chương **??**). Do đó, độ phức tạp của thuật toán theo sau sẽ là cấp số nhân. Một lần nữa, chúng ta sẽ lưu ý rằng chúng ta đang tìm kiếm tất cả các con đường đơn giản, tức là. không chứa các đỉnh lặp lại. Ngược lại, nếu đường dẫn chứa đỉnh lặp lại, điều đó có nghĩa là nó có chứa một vòng lặp - khi đó bạn sẽ có thể tìm tùy ý nhiều đường dẫn xuất phát từ đỉnh đã cho (mỗi đường tiếp theo sẽ có được bằng cách "quay" một hoặc nhiều lần trong chu trình).

Thuật toán tìm tất cả các đường dẫn đơn giản

Nếu chúng ta thực hiện thu thập thông tin quen thuộc theo chiều rộng hoặc chiều sâu, bắt đầu từ i, chúng ta sẽ tìm thấy một đường đi đơn giản có thể từ i đến j, vì chúng ta nhất thiết phải đi qua tất cả các đỉnh của cùng một thành phần (bao gồm cả đỉnh j). Tất nhiên, nếu hai đỉnh không thuộc cùng một thành phần của kết nối, thì không có đường đi nào giữa chúng và như vậy sẽ không được tìm thấy. Việc sửa đổi thuật toán thu thập thông tin theo chiều sâu bao gồm thực hiện đệ quy tuần tự $DFS(k)$ cho mỗi đỉnh k liền kề với đỉnh i hiện đang được xem xét và không chỉ cho lân cận có số lượng tối thiểu. Bằng cách này, chu trình chính trong chương trình sẽ không thay đổi:

```
for (k = 0; k < n; k++)
  if (A[i][k] && !used[k]) allDFS(k, j);
```

Sự khác biệt sẽ là trong cách triển khai mới, chúng ta sẽ thực hiện phép gán used[i] = 0 sau khi trở về từ đệ quy, trong khi trong *DFS* thông thường thì điều này không đúng như vậy. Trong ví dụ của Hình 5.8 điều này có nghĩa như sau: Bây giờ chúng ta hãy nhìn vào

đỉnh 2 - nó có bốn người hàng xóm: 1, 4, 3, 5. Chúng ta đến từ đỉnh 1 và do đó chúng ta sẽ không tiếp tục nó. Thực hiện **for** -chu trình, chọn đỉnh 3 và bắt đầu $DFS(3)$. Do đó, chúng ta sẽ tiếp tục thu thập dữ liệu một cách đệ quy ngoài đỉnh 3, nhưng sau khi quay trở lại từ đệ quy ở cuối $DFS(3)$, phép gán used[3] = 0 sẽ được thực hiện. Hơn nữa, khi quá trình thu thập thông tin tiếp tục với các hàng xóm tiếp theo - đỉnh 4 và 5, trong những lần thu thập thông tin này, nó sẽ có thể vượt qua đỉnh 3. Đây không phải là trường hợp với *DFS* thông thường - với nó, chúng ta đã từng đánh dấu đỉnh 3 với used[3] = 1, nó không còn khả dụng ở bất kỳ bước thu thập thông tin tiếp theo nào.

Chúng ta cũng sẽ giới thiệu một đường dẫn mảng path[], trong đó chúng ta sẽ giữ các đỉnh theo thứ tự của đường đi ngang của chúng. Do đó, nếu tại một số bước chúng ta đạt đến mục tiêu đỉnh cuối cùng j, chúng ta có thể in ra đường dẫn hiện tại. Sau đây là cách triển khai hàm allDFS (i, j). Nó được đáp ứng bởi hai tham số: đỉnh mà chúng ta bắt đầu từ *i*, và đỉnh mà chúng ta phải đạt tới *j*. Tất cả các đầu vào được đặt ở đầu chương trình dưới dạng hằng số.

Chương trình 5.5. Tìm tất cả đường dẫn(505btdfs.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 200
/*Số đỉnh trong đồ thị*/
const unsigned n = 14;
const unsigned sv = 1; /* Đỉnh xuất phát */
const unsigned ev = 10; /* Đỉnh kết thúc */
/* Ma trân kề của đồ thị */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};

char used[MAXN];
unsigned path[MAXN], count;

void printPath(void)
{ unsigned k;
for (k = 0; k <= count; k++)
printf("%u ", path[k] + 1);
printf("\n");
}

/* Tìm tất cả các đường đi đơn giản giữa các đỉnh i và j*/
void allDFS(unsigned i, unsigned j)
{ unsigned k;
  if (i == j) {
     path[count] = j;
     printPath();
     return;
}

/* đánh dấu đỉnh đã ghé thăm*/
used[i] = 1;
path[count++] = i;
for (k = 0; k < n; k++) /* đệ quy cho tất cả những đỉnh kề của i*/
  if (A[i][k] && !used[k]) allDFS(k, j);
  /* return: bỏ đánh dấu đỉnh đã ghé thăm*/
  used[i] = 0; count--;
}

int main() {
  unsigned k;
  for (k = 0; k < n; k++) used[k] = 0;
  count = 0;
  printf("Những con đường đơn giản ở giữa %u và %u: \n", sv, ev);
  allDFS(sv-1, ev-1);
```

```
    return 0;
}
```

Kết quả từ việc thực hiện chương trình:

```
Các đường dẫn đơn giản từ 1 đến 10:
1 2 3 6 7 10
1 2 4 12 6 7 10
1 2 5 7 10
```

Bài tập

▷ **5.9.** Để triển khai một thuật toán để tìm tất cả các đường dẫn đơn giản bằng cách thu thập thông tin theo chiều rộng (*BFS*). Mã được đơn giản hóa hay phức tạp? Độ phức tạp của thuật toán có thay đổi không?

5.4.2. Đường tối ưu trong đồ thị

Định nghĩa 5.15. Cho được một đồ thị có hướng trọng số $G(V, E)$ với các cạnh của các cạnh cho trước là các số thực. Chiều dài của một đường trong G là tổng trọng số của các cạnh có liên quan.

Trong phần này, chúng ta sẽ tập trung vào các thuật toán để tìm độ dài đường dẫn tối đa và nhỏ nhất trong một đồ thị. Trong 5.4.1 chúng ta đã xem xét một thuật toán để tìm tất cả các đường đi giữa hai đỉnh. Một giải pháp như vậy với sự cạn kiệt hoàn toàn là cách duy nhất để giải quyết một số vấn đề về tìm đường đi tối ưu, nhưng đặc biệt trong trường hợp chúng ta đang tìm đường đi tối đa hoặc tối thiểu dưới dạng tổng / tích của trọng lượng của các cạnh cấu thành của nó, có các thuật toán hiệu quả hơn nhiều.

Trong một số tác vụ, có thể xác định độ dài đường dẫn không phải là một tổng, mà là một số hàm khác của trọng số của các cạnh (và thậm chí cả các đỉnh) liên quan đến đường dẫn. Để các thuật toán này cũng vẫn có giá trị trong những trường hợp này, phải có một số tiêu chí cụ thể về tính tối ưu (chúng phụ thuộc vào thuật toán cụ thể đang được xem xét).

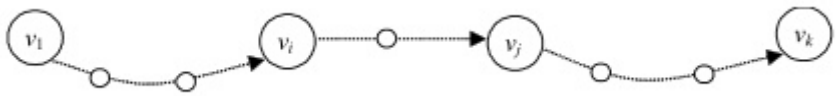
Trước khi chuyển sang phần thực, chúng ta sẽ chú ý đến một số chi tiết chưa rõ ràng. Vì có thể không có hạn chế nào đối với đường

dẫn là đơn giản, chúng ta phải cẩn thận trong trường hợp đồ thị có chứa một vòng lặp. Ví dụ: nếu chúng ta tìm kiếm một đường đi nhỏ nhất và có một chu trình âm (chu trình có độ dài âm), chúng ta sẽ có thể "xoay" trên chu trình này bất kỳ số lần nào, trong đó chiều dài của đường dẫn (bằng tổng của các cạnh trong đó) sẽ giảm nhiều tùy ý đến trừ vô cùng. Tương tự, nếu chúng ta tìm một đường đi cực đại và có một chu trình dương, thì với mỗi đường đi chứa nó, chúng ta sẽ có thể "xoay" chu trình, do đó thu được một chiều dài lớn tùy ý.

Các thuật toán mà chúng ta sẽ bắt đầu sẽ là tìm đường đi nhỏ nhất trong đồ thị. Nếu chúng ta tìm kiếm đường dẫn tối đa, chúng ta có thể dễ dàng sửa đổi chúng. Một giải pháp khả thi là "đảo ngược" trọng số của các cạnh của đồ thị: chúng ta sẽ thay đổi trọng số $f(i,j)$ của mỗi cạnh (i,j) thành $-f(i,j)$. Nếu sau đó chúng ta thực hiện một thuật toán để tìm đường đi nhỏ nhất trong đồ thị mới thu được, thì đường đi tìm được sẽ là đường lớn nhất cho đường ban đầu. Tuy nhiên, cách tiếp cận này không phải lúc nào cũng có thể áp dụng và không phổ biến để tìm các loại đường dẫn cực đại khác nhau trong các loại đồ thị khác nhau.

Hầu hết các thuật toán để tìm một đường cực trị đều dựa trên dấu hiệu tối ưu sau:

Định lý 5.1 (dấu hiệu của tính tối ưu)**.** *Giả sử một đồ thị, một cặp đỉnh của nó (v_1, v_k) và một đường đi nhỏ nhất $p = (v_1, v_2, ..., v_k)$ giữa v_1 và v_k. Khi đó, với $1 \leq i < j \leq k$, đường đi $p' = (v_i, v_i + 1, ..., v_j)$ là đường đi nhỏ nhất từ đỉnh v_i đến v_j.*



Hình 5.10. Tối ưu hóa đường.

Tính hợp lệ của định lý cuối cùng có thể dễ dàng chứng minh bằng cách giả thiết ngược lại (Chúng ta để độc giả xem như một bài tập).

Để tìm những con đường tối thiểu, một mảng thường được giới thiệu nhất, các giá trị của nó có nghĩa là giới hạn trên của những con đường tối thiểu bắt buộc. Ở mỗi bước, một trong các giới hạn trên được giảm xuống (cách thực hiện điều này phụ thuộc vào thuật toán cụ thể) cho đến khi đạt đến mức tối thiểu cần thiết.

Chúng ta sẽ chứng minh sơ đồ tổng quát được coi là với một số thuật toán nổi tiếng nhất để tìm đường dẫn tối thiểu trong đồ thị.

Bài tập

▷ **5.10.** Hãy chứng minh dấu hiệu của sự tối ưu.

- Bất đẳng thức của tam giác

Các thuật toán của Ford-Bellman, Floyd và Dijkstra dựa trên bất đẳng thức của một tam giác: tổng độ dài của hai cạnh bất kỳ lớn hơn độ dài của cạnh thứ ba. Gọi $d(i,j)$ là khoảng cách giữa các đỉnh i và j. Khi đó, trong trường hợp đồ thị có hướng, các bất đẳng thức của tam giác đối với bộ ba đỉnh i,j,k tùy ý có thể được viết như sau:

$$\begin{aligned}
d(i,k) + d(k,j) &\geq d(i,j) \\
d(i,j) + d(j,k) &\geq d(i,k) \\
d(k,i) + d(i,j) &\geq d(k,j) \\
d(j,k) + d(k,i) &\geq d(j,i) \\
d(k,j) + d(j,i) &\geq d(k,i) \\
d(j,i) + d(i,k) &\geq d(j,k)
\end{aligned}$$

Trong trường hợp đồ thị vô hướng, chúng ta có $d(x,y) = d(y,x)$ cho mỗi cặp đỉnh x, y và ba bất đẳng thức cuối cùng trở thành dư thừa. Mỗi thuật toán được liệt kê dựa trên việc kiểm tra tuần tự (một số) bất đẳng thức của tam giác và trong trường hợp vi phạm, thiết lập đẳng thức.

Bài tập

▷ **5.11.** Chứng minh sự cần thiết phải quan sát bất đẳng thức của tam giác như là một hệ quả của dấu hiệu của tối ưu.

- Thuật toán Ford-Bellman

Cho là một đồ thị có hướng $G(V, E)$ và tìm khoảng cách ngắn nhất từ một đỉnh s đến tất cả các đỉnh khác. Chúng ta sẽ giả sử rằng chúng ta đang làm việc với ma trận trọng số A[i][j] của đồ thị G. Trong trường hợp cạnh (i, j) không tồn tại, giá trị của A[i][j] sẽ là write $+\infty$ (và vì chúng ta đang làm việc với cấu trúc dữ liệu hữu hạn, đây sẽ là giá trị cho phép lớn nhất của kiểu các phần tử của A).

Thuật toán Ford-Bellman:

1) Chúng ta nhập một mảng D[], và sau khi hoàn thành thuật toán, D[i] sẽ chứa độ dài của đường đi nhỏ nhất từ s đến mọi đỉnh i khác của đồ thị. Chúng ta khởi tạo D[i] = A[s][i], cho mỗi đỉnh $i \in V$.

2) Chúng ta cố gắng tối ưu hóa giá trị của D[i] cho mỗi $i \in V$ theo cách sau: Với mỗi $j \in V$, nếu D[i]>D[j] + A[j][i], chúng ta gán D[i] = D[j] + A[j][i].

3) Sau khi lặp lại bước 2) $n - 2$ lần, mảng D[] sẽ chứa các đường đi nhỏ nhất cần thiết.

Chương trình 5.6. Thuật toán Ford-Bellman (506belman.c)

```
for (k = 1; k <= n - 2; k++) /* lặp lại (n-2) lần */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (D[i] > D[j] + A[j][i])
        D[i] = D[j] + A[j][i];
```

Rõ ràng, độ phức tạp của thuật toán là $\Theta(n^3)$, vì bước 2) có độ phức tạp $\Theta(n^2)$ và được thực hiện $n - 2$ lần. Điều duy nhất cần cẩn thận trong trường hợp này theo quan điểm của hiện thực là tính tổng D[j] + A[j][i] để không bị tràn (khi bất kỳ D[j] hoặc A[j][i] được khởi tạo với giá trị lớn nhất cho kiểu).

Thuật toán Ford-Bellman hợp lệ cho các đồ thị có trọng số cạnh tùy ý: cả dương và âm. Tất nhiên, trong sự hiện diện của một chu trình âm, chúng ta không thể nói về một con đường tối thiểu (xem ở trên). Một tính năng hữu ích của thuật toán Ford-Bellman là nó cho phép phát hiện các vòng lặp âm: Nếu sau khi hoàn thành, đối với bất kỳ cặp đỉnh (i, j) nào được thỏa mãn D[i]> D[j] + A[j][i], thì đồ

thị chứa một vòng lặp âm (Tại sao?).

Bài tập

▷ **5.12.** Hãy chứng minh thuật toán Ford-Bellman.

▷ **5.13.** Tại sao chu trình ngoài cùng của thuật toán trên được lặp lại $n-2$ lần mà không phải ví dụ như n hay $n-1$?

▷ **5.14.** Chứng minh rằng nếu sau khi hoàn thành thuật toán Ford-Bellman cho bất kỳ cặp đỉnh nào (i, j), D[i]> D[j] + A[j][i] được thỏa mãn, thì đồ thị chứa một số âm xe đạp.

▷ **5.15.** Phát biểu ngược lại có đúng không, cụ thể là: Nếu đồ thị chứa chu trình âm thì sau khi hoàn thành thuật toán Ford-Bellman sẽ có một cặp đỉnh (i, j) mà D[i]> D[j] + A[j][i]?

- Thuật toán của Floyd

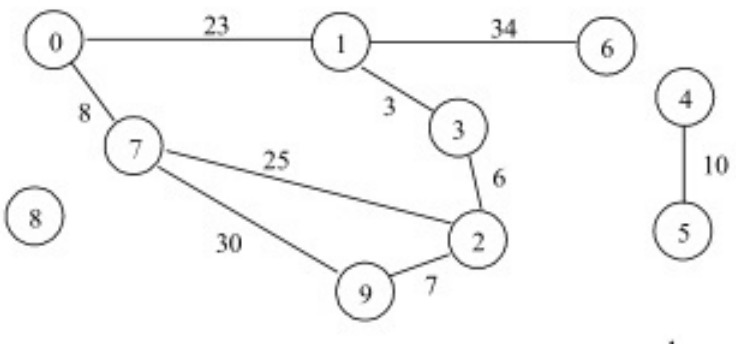
Thuật toán của Floyd tương tự như thuật toán của Ford-Bellman. Sự khác biệt cơ bản là khi nó kết thúc, chúng ta có độ dài của các đường đi nhỏ nhất giữa mỗi cặp đỉnh trong đồ thị mà không cần thêm bộ nhớ (sử dụng ma trận trọng số của nó). Điều này có nghĩa là nếu ngay từ đầu trong A[i][j] trọng số của cạnh (i, j) được viết, thì sau khi thực hiện thuật toán Floyd trên ma trận A[][], giá trị của A[i][j] sẽ là độ dài của đường đi nhỏ nhất giữa i và j.

Thuật toán bao gồm những điều sau: với mỗi hai đỉnh $i, j \in V$, A[i][j] được gán nhỏ hơn A[i][j] và A[i][k] + A[k][j], đối với mỗi $k \in V$ đỉnh. Ý nghĩa của đỉnh k, với tư cách là đỉnh trung gian để tối ưu hóa đường, sẽ trở nên rõ ràng hơn trong phần trình bày của đoạn tiếp theo (lược đồ tổng quát của Floyd).

Độ phức tạp của thuật toán Floyd là $\Theta(n^3)$. Như với Ford-Bellman, sự hiện diện của các cạnh âm không phải là vấn đề, miễn là đồ thị không chứa các vòng âm.

Chúng ta sẽ thực hiện kiểm tra ở trên với ba chu trình lồng nhau - k, i và j. Để thuận tiện, khi so sánh bước chính, thay vì 0, chúng ta sẽ sử dụng giá trị MAX_VALUE, nghĩa là không có cạnh trong ma trận trọng số (trong thực tế, số 0 có thể là giá trị trọng số hợp lệ).

Để chương trình hoạt động chính xác, MAX_VALUE phải có giá trị lớn hơn n.d max, trong đó d max là trọng số lớn nhất của một cạnh của đồ thị (yêu cầu này có thể được làm yếu đi: chỉ cần lấy tổng các cạnh dương là đủ). Cũng nên cẩn thận để làm tràn phần tổng kết: Vì chúng ta có hai phép cộng, MAX_VALUE không được vượt quá MAXINT/2, trong đó MAXINT là giá trị lớn nhất cho kiểu int trong trình biên dịch C tương ứng (trong hầu hết các môi trường, hằng số MAXINT được xác định trong tiêu đề tệp <values.h>).



Hình 5.11. Tối ưu hóa đường.

Sau đây là mã nguồn của chương trình. Dữ liệu đầu vào tương ứng với đồ thị không định hướng trong Hình 5.11 (Tất nhiên, thuật toán của Floyd cũng như chương trình bên dưới sẽ hoạt động chính xác với đồ thị có định hướng). Khi đặt dữ liệu đầu vào là hằng số, chúng ta đã sử dụng giá trị 0 để chỉ ra sự không có cạnh trong ma trận trọng số (để rõ ràng hơn trong mã nguồn). Sau đó (ở đầu hàm floyd() chính) tất cả các trường 0 đều được gán giá trị MAX_VALUE

Chương trình 5.7. Thuật toán floyd (507floyd.c)

```
#include <stdio.h>
/* Số đỉnh tối  đa trong đồ thị  */
#define MAXN 150
#define MAX_VALUE 10000
/* Số đỉnh trong đồ thị  */
const unsigned n = 10;
/* Ma trận trọng số của đồ thị    */
const unsigned A[MAXN][MAXN] = {
{ 0, 23, 0, 0, 0, 0, 0, 8, 0, 0 },
```

```
{23, 0, 0, 3, 0, 0, 34, 0, 0, 0 },
{ 0, 0, 0, 6, 0, 0, 0, 25, 0, 7 },
{ 0, 3, 6, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 10, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 10, 0, 0, 0, 0, 0 },
{ 0, 34, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 8, 0, 25, 0, 0, 0, 0, 0, 0, 30 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 7, 0, 0, 0, 0, 30, 0, 0 }
};

/* Tìm độ dài của đường đi nhỏ nhất giữa mỗi cặp đỉnh*/
void floyd(void)
{ unsigned i, j, k;

  /* các giá trị 0 thay đổi thành MAX_VALUE */
  for (i = 0; i < n; i++) for (j = 0; j < n; j++)
    if (0 == A[i][j]) A[i][j] = MAX_VALUE;

 /* Thuật toán Floyd */
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        if (A[i][j] > (A[i][k] + A[k][j]))
          A[i][j] = A[i][k] + A[k][j];

  for (i = 0; i < n; i++)
    A[i][i] = 0;
}

void printMinPaths(void)
{ unsigned i, j;
  printf("Ma trận trong số sau khi thực hiện Floyd:\n");
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
      printf("%3d ", (MAX_VALUE == A[i][j]) ? 0 : A[i][j]);
    printf("\n");
  }
}
```

```
int main() {
  floyd();
  printMinPaths();
  return 0;
}
```

Kết quả thực hiện thuật toán:

Ma trận trong số sau khi thực hiện Floyd:

```
 0 23 32 26  0  0 57  8 0 38
23  0  9  3  0  0 34 31 0 16
32  9  0  6  0  0 43 25 0  7
26  3  6  0  0  0 37 31 0 13
 0  0  0  0  0 10  0  0 0  0
 0  0  0  0 10  0  0  0 0  0
57 34 43 37  0  0  0 65 0 50
 8 31 25 31  0  0 65  0 0 30
 0  0  0  0  0  0  0  0 0  0
38 16  7 13  0  0 50 30 0  0
```

Chúng ta sẽ lưu ý rằng thứ tự của các chu trình là chủ yếu: chu trình của k phải ở ngoài cùng. Ý tưởng là đầu tiên chúng ta tìm các đường đi ngắn nhất với các đỉnh trung gian 1, sau đó là những con đường có các đỉnh trung gian 1 và 2, sau đó là những con đường có các đỉnh trung gian 1, 2 và 3, v.v. Rõ ràng là vị trí tương hỗ của các chu trình trên i và j là không có ý nghĩa. cả pháp lệnh k, i, j và k, j, i đều đúng. Nhưng mọi quy định khác đều dẫn chúng ta đến một thuật toán không chính xác. Hãy coi ma trận như một ví dụ (lưu ý rằng nó không đối xứng, tức là tương ứng với một đồ thị có định hướng):

```
const n = 4;
int A[n][n] =
{//  A B C D
  { 0, 0, 6, 2}, // A
  { 0, 0, 0, 0}, // B
  { 0, 1, 0, 0}, // C
  { 0, 0, 2, 0} // D
};
```

Với sự sắp xếp đúng, chúng ta thu được ma trận các đường dẫn tối

thiểu sau:

```
0 5 4 2
0 0 0 0
0 1 0 0
0 3 2 0
```

Tuy nhiên, theo thứ tự của các chu trình i, k, j ta có:

```
0 7 4 2
0 0 0 0
0 1 0 0
0 3 2 0
```

Trong trường hợp đầu tiên, đường đi ngắn nhất từ A đến B là (A, D, C, B) với độ dài là 5, và trong trường hợp thứ hai - chúng ta nhận được sai (A, C, B) với độ dài là 7. Chúng ta để người đọc quyết định tại sao nó lại xảy ra sự khác biệt này.

Lưu ý: Việc khôi phục các đường dẫn nhỏ nhất cụ thể, không chỉ tìm độ dài của chúng, sẽ được thảo luận bên dưới khi xem xét thuật toán Dijkstra. Người đọc có thể trực tiếp "chuyển" và áp dụng sơ đồ này cho thuật toán của Floyd.

Bài tập

▷ **5.16.** Trên một cặp đỉnh (i, j) hãy khôi phục một đường đi nhỏ nhất cụ thể.

▷ **5.17.** Trên một cặp đỉnh nhất định (i, j) hãy khôi phục tất cả các đường tối thiểu.

- Thuật toán tổng quát của Floyd

Thuật toán Floyd để tìm đường đi nhỏ nhất giữa tất cả các cặp đỉnh có thể được tóm tắt thành sơ đồ sau để tìm đường đi tối ưu:

$$\varphi^{(k)}(i, j) = F\{\varphi^{(k-1)}(i, j), \varphi^{(k-1)}(i, k) \oplus \varphi^{(k-1)}(k, j)\},$$

trong đó $\varphi^{(1)}(i, j) = f(i, j)$.

$\varphi^{(k)}$ là một hàm đại diện cho đường đi tối ưu giữa hai đỉnh, chứa không quá k đỉnh, có thể tối ưu hóa và $\oplus$ là một phép toán hai đối

số tính toán tổng giá trị của đường đi có được bằng cách ghép hai non -đường dẫn kết hợp. Biểu diễn đệ quy như vậy cho ta ấn tượng rằng chúng ta đang làm việc với n ma trận số: vì ma trận với các giá trị $\varphi^{(1)}(i,j)$ là ma trận trọng số của đồ thị và $\varphi^{(k)}(i,j)$ là ma trận với các đường dẫn tối ưu cần thiết. im lặng. Lược đồ trên được thực hiện lặp đi lặp lại với bộ nhớ bậc hai, như trong thuật toán của Floyd ở đoạn trước. Hàm F được sử dụng để tối ưu hóa giá trị thu được bằng phép toán $\oplus$.

Độ phức tạp của thuật toán tổng quát Floyd, nếu chúng ta giả định rằng độ phức tạp của việc thực hiện phép toán $\oplus$ và tính hàm F là hằng số, giống như tiêu chuẩn: $\Theta(n^3)$.

Hãy xem cách chúng ta có thể áp dụng sơ đồ trên để tìm độ dài tối thiểu của các con đường giữa tất cả các cặp đỉnh. Trong trường hợp này, phép toán $\oplus$ sẽ là tổng độ dài của hai đường đi. Vì chúng ta đang tìm một đường đi có độ dài nhỏ nhất, F sẽ là hàm nhỏ nhất.

Chúng ta sẽ sử dụng một ma trận đơn A[][] để lưu kết quả của các giá trị thứ k của hàm $\varphi^{(k)}(i,j)$. Lúc đầu ta khởi tạo A[][] với các trọng số cho trước của các cạnh của đồ thị (tức là A[][] là ma trận các trọng số). Khi F là cực đại và không tồn tại cạnh (i, j) thì ma trận A[i][j] được viết bằng 0. Giá trị của A[i][i] cho các vòng lặp sẽ là + vô cực (MAX_VALUE). Ngược lại, khi F là cực tiểu và không tồn tại cạnh (i,j) thì ma trận chứa MAX_VALUE và giá trị của A[i][i] là 0.

Thay vì A[i][i] == MAX_VALUE hoặc A[i][i] == 0, thích hợp hơn là thực hiện kiểm tra sau:

```
if (i! = j) && (i! = k) && (k! = j)) {...}
```

Điều này tránh việc sử dụng và khởi tạo các phần tử đường chéo A[i][i].

Chúng ta sẽ minh họa những gì đã được nói cho đến nay bằng một vài ví dụ.

Ví dụ 1. Độ tin cậy trên đường

Một đồ thị có trọng số đại diện cho một mạng máy tính được đưa ra, trong đó trọng số của mỗi cạnh $f(i,j)$ cho biết độ tin cậy của kết nối giữa các máy tính i và j (số từ 0 đến 1). Sau đó, độ tin cậy của đường dẫn trong đồ thị được xác định là tích của trọng lượng của các đường cạnh mà nó chứa. Bài toán là tìm cách đáng tin cậy

nhất để truyền thông tin giữa hai máy tính. Ở đây chương trình của Floyd được áp dụng như sau:

$$\varphi^{(k)}(i,j) = \max(\varphi^{(k-1)}(i,j), \varphi^{(k-1)}(i,k).\varphi^{(k-1)}(k,j))$$

Ma trận A[][], thực hiện lược đồ, chứa 0 nếu không có cạnh giữa hai đỉnh. Vì giá (độ tin cậy) thu được mỗi lần bằng cách nhân trọng lượng của các cạnh của nó, nên hoạt động của nó $\oplus$ là phép nhân. Vì đường đi với độ tin cậy cao nhất được tìm kiếm, F sẽ là *cực đại*.

Ví dụ 2. Đường có độ thấm lớn nhất

Trong đồ thị có trọng số đại diện cho hệ thống cấp nước, trọng lượng của mỗi cạnh $f(i,j)$ cho biết độ thấm của nó. Bài toán là tìm đường đi từ s đến t có độ thấm cực đại. Độ thấm của đường được định nghĩa là trọng lượng tối thiểu của cạnh nối hai đỉnh đường liên tiếp. Sơ đồ tổng quát của Floyd cho bài toán này sẽ là:

$$\varphi^{(k)}(i,j) = \max(\varphi^{(k-1)}(i,j), \min(\varphi^{(k-1)}(i,k), \varphi^{(k-1)}(k,j))),$$

Ma trận A[][] chứa 0 nếu không có cạnh giữa hai đỉnh. Vì giá (độ thấm) của mỗi đường đi được xác định bởi cạnh có độ thấm thấp nhất nên hoạt động của nó là nhỏ nhất. Vì chúng ta đang tìm đường dẫn có thông lượng cao nhất, F sẽ là đường dẫn lớn nhất.

Ví dụ 3. Tìm t con đường nhỏ nhất đầu tiên

Việc tìm đường đi nhỏ nhất t đầu tiên giữa mỗi hai đỉnh trong đồ thị có thể được thực hiện lại bằng cách sử dụng lược đồ tổng quát của Floyd. Ứng dụng của nó trong trường hợp này không trực tiếp như vậy, nhưng lược đồ được giữ nguyên: hàm $\varphi^{(k)}(i,j)$ được định nghĩa không phải là một số mà là một vectơ của các số có ý nghĩa là độ dài của các đường đi nhỏ nhất. Tương tự, F và $\oplus$ được xác định trên vectơ: cực tiểu của hai vectơ $(x_1, x_2, ..., x_t)$ và $(y_1, y_2, ..., y_t)$là vectơ gồm nhiều nhất số t nhỏ giữa các số trong hai vectơ và tổng của hai vectơ: như một vectơ gồm t nhỏ nhất giữa các số $x_1 + y_1, x_1 + y_2, ..., x_1 + y_t, x_2 + y_1, x_2 + y_2, ..., x_2 + y_t, ..., +x_t + y_1, x_t + y_2, ..., x_t + y_t$.

Có một số ví dụ khác (có ứng dụng thực tế) trong đó lược đồ Floyd tổng quát được áp dụng thành công: và những ví dụ khác.

Nếu chúng ta tìm khoảng cách tối thiểu từ một đỉnh cố định s đến tất cả các đỉnh khác, thì có một thuật toán tốt hơn Floyd.

Bài tập

▷ **5.18.** Viết chương trình tính toán độ tin cậy của đường (Ví dụ 1).

▷ **5.19.** Viết chương trình tính toán độ thấm của đường (Ví dụ 2).

▷ **5.20.** Viết chương trình để tìm p đường đi tối thiểu đầu tiên (Ví dụ 3).

- Thuật toán Dijkstra

Phương pháp hiệu quả nhất để tìm các đường đi nhỏ nhất từ một đỉnh cụ thể đến tất cả các đỉnh khác là thuật toán Dijkstra. Cho một đồ thị có hướng có trọng số $G(V, E)$ với n đỉnh được cho. Để áp dụng thuật toán, trọng số của các cạnh $f(i, j)$ phải là số dương (xem chú thích ở cuối đoạn). Hãy tìm đường đi nhỏ nhất từ một đỉnh cố định $s \in V$ đến tất cả các đỉnh khác của đồ thị. Với $\varphi(s, i)$, chúng ta sẽ biểu thị độ dài của đường đi nhỏ nhất từ s đến i. Nói chung, thuật toán Dijkstra dựa trên nguyên tắc sau: Để tìm $\varphi(s, i)$, chúng ta phải tìm giá trị nhỏ nhất trong số $\varphi(s, j) + f(j, i)$, với mỗi $j, j \neq i$. Do đó, mỗi đỉnh i của đồ thị được gán một giá trị thời gian d[i], là giới hạn trên của $\varphi(s, i)$. Trong quá trình hoạt động của thuật toán, giá trị này giảm dần cho đến khi cuối cùng d[i] trở thành chính xác bằng $\varphi(s, i)$:

Thuật toán Dijkstra

1) Khởi tạo mảng d[] như sau:

d[i] = A[s][i] cho mỗi $i \in V$ hàng xóm của s.

d[i] = MAX_VALUE, với mỗi đỉnh i, không kề với s.

Sau khi hoàn thành thuật toán d[i] == MAX_VALUE t.s.t.k. không có đường đi giữa s và i.

2) Chúng ta giới thiệu một tập hợp T, ban đầu chứa tất cả các đỉnh của đồ thị, không có $s : T = V \setminus \{s\}$.

3) Trong khi T chứa ít nhất một đỉnh i mà d[i] <MAX_VALUE:

3.1) Ta chọn đỉnh $j \in T$ sao cho d[j] là cực tiểu.

3.2) Chúng ta loại trừ j khỏi $T : T = T \setminus \{j\}$

3.3) Với mỗi $i \in T$ ta thực hiện d[i] = min (d[i], d[j] + A[j][i]);

Để khôi phục các đỉnh liên quan đến đường dẫn nhỏ nhất (và không chỉ độ dài đường dẫn), chúng ta sẽ giới thiệu một mảng bổ sung pred[]. Do đó, ở vị trí thứ i của mảng pred[], đỉnh j được viết, mà $\varphi(s,j)+f(j,i)$ là cực tiểu:

```
if (d[i] > d[j] + A[j][i]) {
  d[i] = d[j] + A[j][i];
  pred[i] = j;
}
```

Trên mảng này, sau khi hoàn thành thuật toán, đường dẫn có thể được khôi phục, ví dụ bằng cách sử dụng hàm sau:

```
void printPath(unsigned s, unsigned j) {
  if (pred[j] != s)
    printPath(s, pred[j]);
  printf("%d ", j); /* in được thực hiện sau khi quay trở lại từ đệ
    quy*/
}
```

Kỹ thuật được sử dụng trong chức năng trên để khôi phục và in đường đã xây dựng mà chúng ta đã biết từ 5.4.1 - ngầm định một ngăn xếp được sử dụng, bao gồm mỗi đỉnh kế tiếp và việc in được thực hiện sau khi kiểm tra đỉnh cuối cùng, trong quá trình đệ quy ngược lại.

Các đầu vào mẫu được sử dụng trong quá trình triển khai bên dưới được ghi lại dưới dạng hằng số ở đầu chương trình và tương ứng với cột trong Hình 5.11.

Chương trình 5.8. Thuật toán dijkstra (508dijkstra.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 150
#define MAX_VALUE 10000
#define NO_PARENT (unsigned)(-1)
/* Số đỉnh trong đồ thị */
const unsigned n = 10;
const unsigned s = 1;
/* Ma trận trọng số của đồ thị */
const unsigned A[MAXN][MAXN] = {
{ 0, 23, 0, 0, 0, 0, 0, 8, 0, 0 },
```

```
{23, 0, 0, 3, 0, 0, 34, 0, 0, 0 },
{ 0, 0, 0, 6, 0, 0, 0, 25, 0, 7 },
{ 0, 3, 6, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 10, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 10, 0, 0, 0, 0, 0 },
{ 0, 34, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 8, 0, 25, 0, 0, 0, 0, 0, 0, 30 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 7, 0, 0, 0, 0, 30, 0, 0 }
};

char T[MAXN];
unsigned d[MAXN];
int pred[MAXN];

/* Thuật toán Dijkstra - đường dẫn tối thiểu từ s đến các đỉnh khác *
   /
void dijkstra (unsigned s)
{ unsigned i;
  for (i = 0; i < n; i++) /*sự khởi tạo: d[i]=A[s][i], i trong V, i !
    = s */
    if (0 == A[s][i]) {
      d[i] = MAX_VALUE;
      pred[i] = NO_PARENT;
    }
    else {
      d[i] = A[s][i];
      pred[i] = s;
    }
  for (i = 0; i < n; i++) T[i] = 1; /* T chứa tất cả các ngọn*/
  T[s] = 0;
  pred[s] = NO_PARENT; /* từ đồ thị, ngoại trừ s */
  while (1) { /* trong khi T chứa i: d[i] < MAX_VALUE */
    /*chọn đỉnh j từ T mà d[j] là cực tiểu*/
    unsigned j = NO_PARENT;
    unsigned di = MAX_VALUE;
    for (i = 0; i < n; i++)
      if (T[i] && d[i] < di) {
        di = d[i];
        j = i;
```

```
      }
    if (NO_PARENT == j) break; /* d[i] = MAX_VALUE, với tất cả i:
      thoát*/
    T[j] = 0; /*Loại trừ j khỏi T */
    /*đối với mỗi i trong số T, thực hiện D[i] = min (d[i], d[j]+A[j][i
      ]) */
    for (i = 0; i < n; i++)
      if (T[i] && A[j][i] != 0)
        if (d[i] > d[j] + A[j][i]) {
          d[i] = d[j] + A[j][i];
          pred[i] = j;
        }
  }
}

void printPath(unsigned s, unsigned j)
{ if (pred[j] != s) printPath(s, pred[j]);
  printf("%u ", j+1);
}

/* In các đường dẫn tối thiểu được tìm thấy*/
void printResult(unsigned s)
{ unsigned i;
  for (i = 0; i < n; i++) {
    if (i != s) {
      if (d[i] == MAX_VALUE)
        printf("Không có đường đi giữa các đỉnh %u và %u\n", s+1,
    i+1);
      else {
        printf("Đường tối thiểu từ đỉnh %u đến %u: %u ", s+1, i+1, s
    +1);
        printPath(s, i);
        printf(", chiều dài con đường: %u\n", d[i]);
      }
    }
  }
}

int main() {
  dijkstra (s-1); printResult(s-1);
```

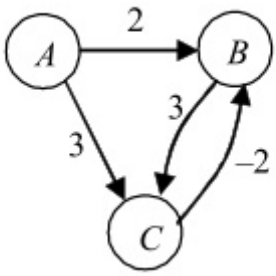
```
}
```

Kết quả thực hiện chương trình:

```
Đường tối thiểu từ đỉnh 1 đến đỉnh 2: 1 2, chiều dài đường: 23
Đường tối thiểu từ đỉnh 1 đến 3: 1 2 4 3, chiều dài đường: 32
Đường tối thiểu từ đỉnh 1 đến 4: 1 2 4, chiều dài đường: 26
Không có đường đi giữa các đỉnh 1 và 5
Không có đường đi giữa các đỉnh 1 và 6
Đường tối thiểu từ đỉnh 1 đến 7: 1 2 7, chiều dài đường: 57
Đường tối thiểu từ đỉnh 1 đến 8: 1 8, chiều dài đường: 8
Không có đường đi giữa các đỉnh 1 và 9
Đường tối thiểu từ đỉnh 1 đến 10: 1 8 10, chiều dài đường: 38
```

Độ phức tạp của thuật toán trong cách triển khai trên là $\Theta(n^2)$. Với việc lựa chọn cẩn thận hơn các cấu trúc dữ liệu (ví dụ, nếu chúng ta sử dụng kim tự tháp - xem 3.1.9., [Cormen, Leiserson, Rivest-1997]) thì độ phức tạp có thể giảm xuống $\Theta(n.\log_2 n)$.

Lưu ý: Thuật toán Dijkstra sẽ không hoạt động bình thường nếu có trọng số âm trong cột. Để minh họa điều này, chúng ta sẽ xem xét một ví dụ cụ thể (xem Hình 5.12).



Hình 5.12. Đồ thị có trọng số với cạnh âm.

Để tìm các đường đi nhỏ nhất A đến B và C, các bước sau sẽ được thực hiện:

1) Khởi tạo $d[A] = 0, d[B] = f(A,B) = 2, d[C] = f(A,C) = 3$.

2) Khởi tạo tập hợp $T = \{A,B,C\} \setminus \{A\} = \{B,C\}$.

3) Chọn một đỉnh $i \in T$ mà $d[i]$ là cực tiểu, đây là đỉnh B: Loại trừ B khỏi T. Với mỗi đỉnh từ T (và trong T chỉ có đỉnh C) được thực

hiện:

$$d[C] = \min(d[C], d[B] + f(B,C)) = \min(3, 2+3) = 3.$$

Trong bước tiếp theo, chỉ có C trong T, và sau khi tắt C, tập T vẫn trống và thuật toán kết thúc.

Vì vậy, chúng ta có $d[A] = 0, d[B] = 2, d[C] = 3$. Tuy nhiên, rõ ràng, đường đi ngắn nhất từ A đến B không phải là độ dài 2, mà là độ dài 1 (đây là đường đi ACB, vì $3 + (-2) = 1$).

Bài tập

▷ **5.21.** So sánh thuật toán của Dijkstra với thuật toán của Ford-Bellman và Floyd.

▷ **5.22.** Có thể khôi phục đường đi nhỏ nhất giữa một cặp đỉnh chỉ từ mảng d[] mà không cần sử dụng một mảng bổ sung như pred[] không?

- Lũy thừa ma trận của phần tử lân cận

Có một mối liên hệ thú vị giữa phép toán lũy thừa ma trận lên bậc k và số đường có độ dài k giữa hai đỉnh trong một đồ thị vô hướng. Cho một đồ thị vô hướng $G(V,E)$ với ma trận lân cận A[][]. Cho A' là ma trận sao cho A'[i][j] chứa số đường đi có độ dài k giữa các đỉnh i và j. Khi đó trong ma trận $A'' = A'.A$ giá trị của A"[i][j] sẽ chính xác là số đường có độ dài $k+1$ giữa hai đỉnh. Sự thật của câu lệnh này dựa trên quy tắc nhân ma trận:

$$A''[i][j] = \sum_{i=1}^{n} A'[i][t].A[t][j]$$

Vì A'[i][t] là số đường có độ dài k giữa i và t, nên số đường có độ dài $k+1$ giữa i và j sẽ tăng lên A' [i] [t] với điều kiện là cạnh (t,j) là từ đồ thị (tương đương với A[t][j] == 1). Vì vậy, chúng ta đã đến với những điều sau

Định lý 5.2. *Cho một đồ thị vô hướng $G(V,E)$ với một ma trận lân cận A. Nếu chúng ta nâng A lên lũy thừa của k, thì $C[i][j] = A^k[i][j]$ sẽ là số*

đường có độ dài k, nằm giữa các đỉnh i và j. Nếu $C[i][j] = 0$, thì không có đường đi nào giữa i và j có độ dài k.

Định lý cũng có giá trị trong trường hợp đồ thị có định hướng (và thậm chí là đồ thị đa phương), nhưng chỉ khi các giá trị trong ma trận lân cận không bao gồm các giá trị -1, biểu thị cạnh "quay lại".

Kết quả thu được không chỉ hữu ích trong việc giải các bài toán tìm số đường mà còn cả "đạo hàm" của chúng: chu trình, các dạng thành phần khác nhau và hơn thế nữa. Hơn nữa, do đồ thị "ma trận lân cận", nhiều bài toán cổ điển của đại số liên quan đến ma trận được chuyển thành các bài toán trên đồ thị, kể cả những bài toán vẫn chưa giải được.

Bài tập

▷ **5.23.** Hãy viết chương trình thực hiện công việc dựa trên định lý trên.

- Thuật toán Warshal và ma trận khả năng tiếp cận

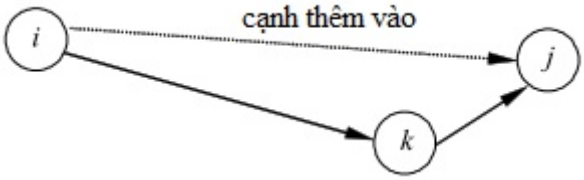
Bài toán: Giả sử một đồ thị có hướng $G(V, E)$ với một ma trận lân cận A[][]. Chúng ta đang tìm một ma trận A′[][] sao cho:

- A′[i][j] == 1, nếu và chỉ khi có một đường đi (có độ dài bất kỳ) giữa các đỉnh i và j.
- A′[i] [j] == 0, khi không có đường đi giữa hai đỉnh.

Thuật toán giải bài toán này được đặt tên theo Warshal và kết quả ứng dụng của nó - ma trận A′[][], được gọi là *ma trận khả năng phản ứng* của đồ thị. Đồ thị $G'(V, E')$ tương ứng với ma trận A′[][] (nếu chúng ta coi A′[][] là *ma trận lân cận*) được gọi là đồ thị chuyển tiếp đóng của G (nó sẽ được thảo luận lại trong 5.5.1.).

Thuật toán của Worschal là phức tạp $\Theta(n^3)$ và tiến hành như sau: với mỗi ba đỉnh $k, i, j \in V$, nếu $(i, k) \in E$ và $(k, j) \in E$, thì chúng ta thêm vào tập từ các cạnh của đồ thị và cạnh (i, j) (Hình 5.13).

Việc xem xét ba đỉnh được thực hiện bởi ba chu trình lồng nhau như sau:



Hình 5.13. Bước đóng chuyển tiếp.

Ba chu trình lồng nhau

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++) {
    if (A[i][k])
      for (j = 0; j < n; j++)
    if (A[k][j]) A[i][j] = 1;
}
```

Sau khi thực hiện đoạn chương trình được hiển thị, ma trận A[][] sẽ được sửa đổi thành công thành ma trận có khả năng truy xuất yêu cầu A′[][]. (Tại sao?) Người đọc chú ý có lẽ đã nhận thấy sự giống nhau giữa thuật toán trên và Floyd. Đây không phải là ngẫu nhiên: Thuật toán Worshal là một trường hợp đặc biệt của Floyd.

Bài tập

▷ **5.24.** Viết trong C/C++ thuật toán Worshal.

▷ **5.25.** Chứng minh rằng thuật toán Worshal tìm đúng ma trận khả năng tiếp cận.

- đường đi dài nhất trong thị chu trình

Nhiều bài toán thực tế được giải quyết, rút gọn thành bài toán tìm đường đi dài nhất trong đồ thị mạch hở. Điều kiện của sự nhạy bén là điều cần thiết, vì cách tiếp cận duy nhất để giải quyết vấn đề trong trường hợp chung là thông qua sự kiệt sức hoàn toàn. Mặt khác, khi không có chu trình trong đồ thị, một số nguyên tắc tối ưu được đáp ứng (điều này cũng cần thiết cho tất cả các bài toán được giải bằng tối ưu hóa động - Chương 8) và bài toán có thể được giải

quyết hiệu quả hơn nhiều.

Chúng ta sẽ xem xét một phát biểu thực tế về bài toán tìm ra con đường tối đa.

Bài toán: Một nhóm lập trình viên phát triển một sản phẩm phần mềm bao gồm các bài toán riêng lẻ. Mỗi bài toán có một thời lượng cụ thể và kết nối hai giai đoạn phát triển sản phẩm: ban đầu và cuối cùng. Một bài toán không thể được bắt đầu nếu giai đoạn đầu tiên của nó chưa được hoàn thành. Để hoàn thành một giai đoạn, tất cả các bài toán cuối cùng của nó phải được hoàn thành. Để xác định thời gian tối thiểu (với điều kiện chúng ta có số lượng lập trình viên không giới hạn) đủ để hoàn thành toàn bộ dự án. Dự án được coi là hoàn thành khi tất cả các giai đoạn của nó đã được hoàn thành.

Nếu chúng ta sử dụng đồ thị hướng có trọng số trong mô hình bài toán, trong đó đỉnh là giai đoạn và cạnh là bài toán, thì thời gian tối thiểu cần thiết sẽ *bằng độ dài của số đỉnh lớn nhất* trong đồ thị. Trong tài liệu, con đường này còn được gọi là *con đường phê bình*.

Thuật toán dưới đây dựa trên kỹ thuật tối ưu hóa động (xem Chương 8).

Thuật toán:

Cho đồ thị có hướng chu trình $G(V, E)$ và $A[N][N]$ là ma trận các trọng số của G.

1) Chúng ta nhập một mảng maxDist[], trong đó chúng ta sẽ giữ khoảng cách tối đa từ mỗi đỉnh trong cột, tức là maxDist[i] sẽ bằng độ dài của đường đi lớn nhất bắt đầu với đỉnh i. Lúc đầu, chúng ta khởi tạo tất cả các phần tử của maxDist[] bằng các số không. Trong mảng thứ hai savePath[], chúng ta sẽ giữ các ý định là đường dẫn dài nhất, vì mảng được khởi tạo bằng -1.

2) Xét một đỉnh $i \in V$ mà từ đó không có cạnh nào nhô ra. Một đỉnh như vậy nhất thiết phải tồn tại, bởi vì đồ thị là dòng chu trình (hãy để người đọc xem xét điều này đến từ đâu!). Chúng ta xóa đỉnh này khỏi đồ thị và gán cho từng đỉnh trước đó của nó k:

$$maxDist[k] = \max\{maxDist[k], maxDist[i] + A[k][i]\}$$

3) Thực hiện bước 2) cho đến khi không còn đỉnh nào trong đồ thị. Khi đó giá trị lớn nhất của maxDist [i], với $i = 1, 2, ...n$ sẽ là độ

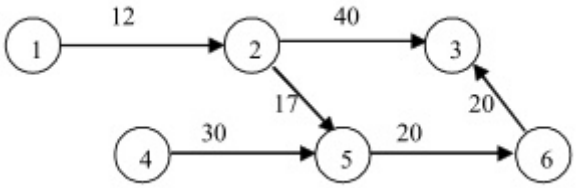
dài của đường dẫn tối đa được yêu cầu.

Thuật toán có độ phức tạp $\Theta(n+m)$, và khi biểu diễn đồ thị bằng ma trận lân cận - $\Theta(n^2)$. Chúng ta sẽ minh họa cách nó được áp dụng cho một đồ thị cụ thể. Hình 5.13. hiển thị thứ tự các đỉnh bị xóa khỏi đồ thị và các giá trị trong maxDist[] thay đổi như thế nào.

Thực hiện thuật toán:

Chúng ta sẽ thực hiện thuật toán được trình bày đệ quy ở trên. Chúng ta sẽ sử dụng một sửa đổi của thu thập thông tin chuyên sâu. Mỗi lần chúng ta quay trở lại từ một lệnh gọi $DFS(i)$, chúng ta thấy mình đang ở trong một tiền nhiệm j ở đầu i. Ở đó, chúng ta sẽ giữ $maxDist[j] + A[j][i]$ tối đa cho mỗi thứ j kế thừa của i và cuối cùng, trước khi thoát khỏi hàm DFS, chúng ta sẽ gán nó cho $maxDist[i]$. Hàm $DFS(i)$ chỉ được thực thi khi $maxDist[i]$ chưa được tính, tức là. `if` maxDist[i] == 0 (khởi tạo ban đầu toàn bộ mảng bằng các số 0). Hàm DFS được khởi động tuần tự cho mỗi đỉnh của đồ thị.

Để lưu và in tất cả các đỉnh của đường tìm kiếm, không chỉ độ dài của nó, trong mảng savePath[] ở vị trí thứ i, chúng ta sẽ viết đoạn kế tiếp này là j, mà maxDist[i] + A[i][j] lớn nhất là thiết lập



Hình 5.14. Tìm đường quan trọng trong đồ thị.

1. Khởi tạo $maxDist[i] = 0$, cho mỗi i.

2. Xóa đỉnh 3 $\Rightarrow$ $maxDist[6] = max\{maxDist[6], maxDist[3] + 20\} = max\{0, 20\} = 20$; Tương tự, $maxDist[2] = 40$

3. Xóa đỉnh 6 $\Rightarrow$ $maxDist[5] = 40$;

4. Xóa đỉnh 5 $\Rightarrow$ $maxDist[4] = 70, maxDist[2] = \max\{40, 40 + 17\} = 57$

5. Xóa đỉnh 4

6. Xóa đỉnh 2 $\Rightarrow$ $maxDist[1] = 69$

7. Xóa đỉnh 1

Thời gian dài nhất: 70($maxDist[4]$)

Sau đây là mã nguồn của chương trình. Số lượng đỉnh và ma trận trọng số $A[][]$ của đồ thị được đặt dưới dạng hằng số.

Chương trình 5.9. Đường dài nhất (509longpath.c)

```
#include <stdio.h>
/*  Số đỉnh tối đa trong đồ thị */
#define MAXN 150
/* Số đỉnh trong đồ thị*/
const unsigned n = 6;
/*Ma trận trọng số của đồ thị*/
const unsigned A[MAXN][MAXN] = {
{ 0, 12, 0, 0, 0, 0 },
{ 0, 0, 40, 0, 17, 0 },
{ 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 30, 0 },
{ 0, 0, 0, 0, 0, 20 },
{ 0, 0, 20, 0, 0, 0 }
};

int savePath[MAXN], maxDist[MAXN];

void DFS(unsigned i)
{ int max, d;
  unsigned j;

  if (maxDist[i] > 0)
    return;
  max = maxDist[i];
  for (j = 0; j < n; j++)
    if (A[i][j]) {
      DFS(j);
      d = maxDist[j] + A[i][j];
      if (d > max) {
        max = d;
        savePath[i] = j;
      }
    }
  maxDist[i] = max;
```

```
}

void solve(void)
{ unsigned i, maxi;
  /*sự khởi tạo*/
  for (i = 0; i < n; i++) {
    maxDist[i] = 0;
    savePath[i] = -1;
  }

  for (i = 0; i < n; i++)
    if (maxDist[i] == 0) DFS(i);

  maxi = 0;
  for (i = 0; i < n; i++)
    if (maxDist[i] > maxDist[maxi])
      maxi = i;

  printf("Độ dài của đường dẫn tới hạn là% d \nĐường dẫn là: ",
    maxDist[maxi]);
  while (savePath[maxi] >= 0) {
    printf("%u ", maxi + 1);
    maxi = savePath[maxi];
  }
  printf("%d\n", maxi + 1);
}

int main() {
  solve();
  return 0;
}
```

Kết quả thực hiện chương trình:

```
Chiều dài của đường tới hạn là 70
Đường là: 4 5 6 3
```

Bài tập

▷ **5.26.** Để chứng minh rằng trong một đồ thị mạch hở nhất thiết phải tồn tại:

- ít nhất một đỉnh không có đỉnh trước
- ít nhất một đỉnh không có người thừa kế

▷ **5.27.** Để triển khai một biến thể của thuật toán, trong đó ở mỗi bước, một đỉnh không có đỉnh trước bị loại bỏ, thay vào đó là đỉnh không có đỉnh kế tiếp.

- Đường đơn dài nhất giữa hai đỉnh trong bất kỳ đồ thị nào

Chúng ta sẽ nhắc lại một lần nữa rằng việc tìm kiếm đường đi dài nhất trong đồ thị chu trình là một bài toán NP đầy đủ, mà chúng ta sẽ xem xét trong **??**. Cạn kiệt hoàn toàn được áp dụng để giải quyết nó.

Bài tập

▷ **5.28.** Đưa ra một ví dụ về một đồ thị có hướng và một cặp đỉnh của nó là i và j, trong đó thuật toán trước đó không tìm ra chính xác đường đi dài nhất giữa i và j.

▷ **5.29.** Có những trường hợp nào khi thuật toán tìm đường đi dài nhất trong đồ thị chu trình sẽ hoạt động chính xác đối với một cặp đỉnh trong đồ thị tuần hoàn? Nếu có - khi nào? Nếu không - tại sao? Đưa ra các ví dụ có liên quan.

5.4.3. Chu trình

Tiếp tục logic của chủ đề từ 5.4.2 cho đường là việc tìm kiếm và nghiên cứu các chu trình của một đồ thị.

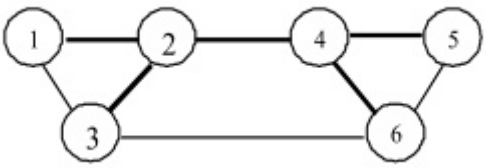
- tìm kiếm một tập hợp các chu trình cơ bản

Cho một đồ thị liên thông vô hướng $G(V, E)$ với n đỉnh và m cạnh. Cho $D(V, T)$ là một cây bao tùy ý của G. Thêm vào D một cạnh không thuộc T sẽ đóng một vòng lặp đơn giản. Chúng ta sẽ nói rằng chu trình này thuộc *tập các chu trình cơ bản* của đồ thị G đối với cây bao phủ D của nó.

Mỗi cây phủ D có đúng $q = n - 1$ cạnh. Các cạnh của G không thuộc D có tổng là $m - n + 1$. Vì khi thêm mỗi chúng vào T sẽ thu

được đúng một chu trình, nên số chu trình trong tập cơ bản là $m - n + 1$.

Xác định tập hợp các chu trình cơ bản, chúng ta xác định duy nhất cấu trúc chu trình của đồ thị, vì bất trình chu trình nào khác trong G đều có thể được biểu diễn bằng các chu trình "dán" từ một số tập cơ bản. Ví dụ, đối với các kết quả đọc trong Hình 5.15 đồ thị và cây che phủ đã chọn (các cạnh tham gia được hiển thị bằng màu tối) các chu trình đơn giản là $A = (1,2,3)$, $B = (4,5,6)$ và $C = (2,3,4,6)$. Bất kỳ chu trình đơn giản nào khác đều có thể được xây dựng bằng cách nối (dán) một số chu trình A, B và C, có ít nhất một cạnh chung (chỉ riêng một đỉnh / đỉnh chung là không đủ!). Bằng cách nối hai chu trình, chúng ta có nghĩa là loại bỏ các cạnh chung của chúng (và tương ứng là các đỉnh cô lập sau khi loại bỏ các cạnh này) - vì vậy chúng ta có lại các chu trình đơn giản. Ví dụ, nối A và C, ta được chu trình $(1,2,4,6,3)$.



Hình 5.15. Tập hợp cơ bản của các chu trình trong đồ thị.

Chúng ta sẽ thực hiện việc tìm kiếm một tập hợp các chu trình cơ bản, một lần nữa bằng cách sử dụng sửa đổi chức năng thu thập thông tin độ sâu DFS. Trong bước đầu tiên, chúng ta sẽ tìm một cây bao phủ tùy ý của đồ thị, sau đó mỗi cạnh không tham gia vào cây bao phủ đã xây dựng sẽ đóng một chu trình và chúng ta sẽ tìm chu trình này với một đường đi ngang khác của đồ thị, được thực hiện bên dưới hàm $DFS2()$. Tổng độ phức tạp của thuật toán là $\Theta(m.(M+n))$.

Trong mã nguồn được đưa ra bên dưới, đồ thị được biểu diễn bằng ma trận lân cận $A[][]$, với $A[i][j]$ cho biết có cạnh giữa i và j hay không. Sau đó, trong quá trình thực thi chương trình, $A[i][j] == 2$, nếu cạnh tham gia vào cây bao phủ đã xây dựng.

Chương trình 5.10. Chu trình đơn (510allcyc.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 150
/* Số đỉnh trong đồ thị */
const unsigned n = 10;
/ * Đồ thị được biểu diễn bởi một ma trận lân cận: 0 - không có cạnh
    ; 1 - có;
* Sau đó với 2 chúng ta sẽ đánh dấu các cạnh của cây của đồ thị.
* /
char A[MAXN][MAXN] = {
  { 0, 1, 1, 0, 0, 0 },
  { 1, 0, 1, 1, 0, 0 },
  { 1, 1, 0, 0, 0, 1 },
  { 0, 1, 0, 0, 1, 1 },
  { 0, 0, 0, 1, 0, 1 },
  { 0, 0, 1, 1, 1, 0 }
};

char used[MAXN];
unsigned cycle[MAXN], d;
/* Tìm bất kỳ cây phủ nào */
void DFS(unsigned v)
{ unsigned i;
  used[v] = 1;
  for (i = 0; i < n; i++)
    if (!used[i] && A[v][i]) {
      A[v][i] = 2;
      A[i][v] = 2;
      DFS(i);
    }
}
/*In một chu trình tìm thấy*/
void printCycle(void)
{ unsigned k;
  for (k = 0; k < d; k++)
    printf("%u ", cycle[k] + 1);
  printf("\n");
}
/* Tìm kiếm một chu trình theo cây phủ được tìm thấy */
```

```
void DFS2(unsigned v,unsigned u)
{ unsigned i;
  if (v == u) {
    printCycle();
    return;
  }
used[v] = 1;
for (i = 0; i < n; i++)
  if (!used[i] && 2==A[v][i]) {
    cycle[d++] = i;
    DFS2(i, u);
    d--;
  }
}

int main() {
  unsigned i, j, k;
  DFS(0);
  printf("Các chu trình đơn trong đồ thị là: \n");
  for (i = 0; i < n - 1; i++)
    for (j = i + 1; j < n; j++)
      if (1 == A[i][j]) {
        for (k = 0; k < n; k++) used[k] = 0;
        d = 1;
        cycle[0] = i;
        DFS2(i, j);
      }
  return 0;
}
```

Kết quả thực hiện chương trình:

```
Các chu trình đơn trong đồ thị là:
1 2 3
2 3 6 4
5 4 6
```

Bài tập

▷ **5.30.** Thuật toán được mô tả trên (có / không sửa đổi) có hoạt động đối với đồ thị đa phương không?

▷ **5.31.** Thuật toán được mô tả trên (có / không sửa đổi) có hoạt động đối với đồ thị có định hướng không? Và cho một đồ thị nhiều định hướng?

▷ **5.32.** Thuật toán được thảo luận ở trên có độ phức tạp $\Theta(m.(M+n))$, và việc triển khai, do sử dụng ma trận lân cận để biểu diễn đồ thị - độ phức tạp $\Theta(n^4)$. Có thể cải thiện kết quả này không? Chúng ta đã chỉ ra rằng số chu trình đơn có bậc là $\Theta(m+n)$, và độ dài của mỗi chu trình phụ thuộc vào chiều cao h của cây phủ đã xây dựng, tức là độ phức tạp tối thiểu để tìm và các vòng là $\Theta((m+n).h)$. Rõ ràng là trong trường hợp xấu nhất, chiều cao bằng với số đỉnh của đồ thị, tức là. độ phức tạp $\Theta((m+n).n)$ thu được.

Bạn có thể nghĩ ra một thuật toán để tìm các vòng lặp hoạt động với độ phức tạp tối thiểu được chỉ định $\Theta((m+n).h)$ không? Bạn có thể đề xuất chiến lược nào để xây dựng cây che phủ để nó có chiều cao tối thiểu? Có thể nói rằng với một chiến lược được lựa chọn tốt, h sẽ thuộc về $O(\log_2 n)$?

- chu trình tối thiểu qua đỉnh

Bài toán: Cho đồ thị có hướng trọng số $G(V,E)$ và đỉnh $i \in V$. Tìm một vòng lặp (không nhất thiết phải đơn) chứa đỉnh i và tổng trọng lượng của các cạnh tham gia là nhỏ nhất.

Lời giải: Với mỗi $j \in V$ ta tìm được đường đi nhỏ nhất $\varphi(i,j)$ từ i đến j và đường đi nhỏ nhất $\varphi(j,i)$ từ j đến i. Cho $S_k = \varphi(i,k) + \varphi(k,i), k \in V$. Khi đó S_k nhỏ nhất là độ dài của chu trình nhỏ nhất mà chúng ta đang tìm.

Điều quan trọng là chu trình được tìm thấy không nhất thiết phải đơn. Bài toán đặt ra một ràng buộc bổ sung cho chu trình để tránh các đỉnh lặp lại cũng khá thú vị. Nếu chúng ta xem xét một đồ thị không có trọng lượng của các cạnh, vấn đề được giải quyết bằng cách thu thập thông tin theo chiều rộng: một sửa đổi nhỏ của thuật toán để tìm số đỉnh tối thiểu giữa hai đỉnh. Giải pháp trong trường hợp tổng quát (đối với đồ thị có trọng số) được thảo luận trong Bài toán 5.5.

Một bài toán thú vị khác về mặt thuật toán của các vòng lặp là

bài toán tìm một vòng lặp đơn tối thiểu chứa ít nhất k đỉnh (xem Bài toán 5.93).

Trường hợp "đặc biệt" của nó, tại $k = n$, được xem xét trong đoạn tiếp theo đây.

Bài tập

▷ **5.33.** Triển khai thực hiện thuật toán trên được mô tả và tính độ phức tạp của nó.

▷ **5.34.** Một đồ thị có hướng có trọng số và một đỉnh từ nó đã cho. Tìm một chu trình đơn chứa đỉnh mà tổng trọng lượng của các cạnh liên quan là nhỏ nhất.

5.4.4. Các chu trình Hamilton. Bài toán đường thương mại

Định lý 5.3. *Chu trình Hamilton (HC) trong đồ thị được gọi là chu trình chứa mỗi đỉnh của đếm chính xác một lần. Một đồ thị chứa một chu trình như vậy được gọi là một Hamilton.*

Mệnh đề 5.1. *Trong đồ thị Hamilton, chu trình Hamilton tương ứng có thể được xây dựng bắt đầu từ bất kỳ đỉnh ban đầu nào.*

Hai bài toán cổ điển là kiểm tra xem một biểu đồ có phải là Hamilton hay không và bài toán dành cho khách du lịch thương mại, đó là tìm một chu trình Hamilton có độ dài tối thiểu trong một biểu đồ có trọng số. Cả hai bài toán được định nghĩa là NP đầy đủ (xem **??**), Có nghĩa là độ phức tạp là cấp số nhân trong trường hợp xấu nhất. Trong đoạn này chúng ta sẽ xem xét và giải quyết vấn đề thứ hai.

Thuật toán giải quyết công việc cho khách du lịch

Chúng ta sẽ giải quyết vấn đề bằng cách vét cạn hoàn toàn. Trong 5.4.1 chúng ta đã xem xét cách tìm tất cả các đường đi đơn giữa hai đỉnh. Chúng ta sẽ áp dụng cách tiếp cận tương tự, muốn các đỉnh bắt đầu và kết thúc trùng khớp. Chúng ta chọn bất kỳ đỉnh i nào và bắt đầu từ nó, chúng ta sẽ xây dựng tất cả các tuyến có thể theo thứ tự. Chúng ta sẽ thay đổi độ dài *curSum* của tuyến được xây dựng

cho đến nay trong phần thân của đệ quy. Khi chúng ta tìm thấy một chu trình Hamilton mới, chúng ta sẽ kiểm tra xem độ dài của nó có nhỏ hơn chu trình nhỏ nhất được tìm thấy cho đến nay hay không (chúng ta sẽ giữ nó trong biến *minSum*) và nếu nó nhỏ hơn - chúng ta sẽ lưu nó. Nếu trọng số của tất cả các cạnh của đồ thị là dương, chúng ta có thể làm cho thuật toán của mình hiệu quả hơn một chút bằng cách áp dụng "cắt đệ quy" (xem **??**).

Kỹ thuật như sau: nếu tại bất kỳ bước nào chiều dài *curSum* của con đường chúng ta đang xây dựng lớn hơn *minSum*, chúng ta sẽ làm gián đoạn việc xây dựng tuyến đường hiện tại. Điều này được thực hiện bởi vì ngay cả khi nó kéo dài sau chu trình Hamilton, thì độ dài của nó chắc chắn sẽ lớn hơn chiều dài nhỏ nhất. Tuy nhiên, để tận dụng tối ưu hóa này, đồ thị không được chứa các cạnh âm làm giảm *curSum* sau đó! Để thấy điều này, chúng ta hãy nhìn vào ma trận lân cận mẫu sau:

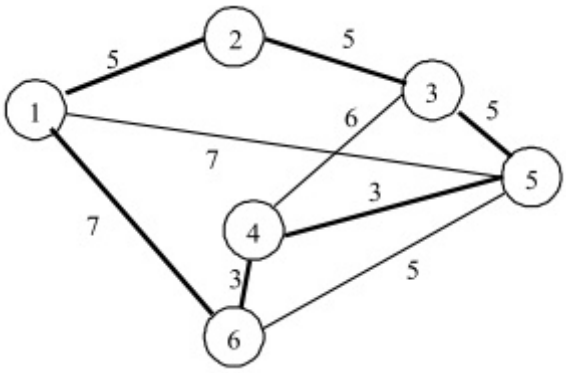
Ma trận lân cận mẫu

```
int A[n][n] =
{ // A B C D
{  0, 1, 2, 0}, // A
{ -2, 0, 1, 0}, // B
{  0, 0, 0, 1}, // C
{  1, 2, 0, 0} // D
};
```

Nếu ở một bước nào đó, chúng ta đã tìm thấy một chu trình Hamilton nhỏ nhất hiện tại (A, B, C, D, A) với độ dài $1 + 1 + 1 + 1 = 4$, thì khi chúng ta đạt đến đỉnh B trong chu trình Hamilton (A, C, D, B, A) tổng hiện tại sẽ là $2 + 1 + 2 = 5$. Và vì $5 > 4$, và 4 là độ dài tối thiểu cho đến nay, việc xem xét ứng cử viên mới, là tối thiểu, chấm dứt, bởi vì khi thêm cạnh, A) với trọng lượng - 2, độ dài của chu trình trở thành 3, và 3 <4.

Trong cách triển khai đề xuất, đồ thị được biểu diễn bằng ma trận trọng số A[][], và dữ liệu đầu vào được ghi lại dưới dạng hằng số ở đầu chương trình. Đồ thị được sử dụng làm ví dụ về dữ liệu đầu vào trong quá trình thực hiện được thể hiện trong Hình 5.16.

Chương trình 5.11. Chu trình Hamilton nhỏ nhất (510tps.c)



Hình 5.16. Chu trình Hamilton nhỏ nhất

```
#include <stdio.h>
/* Số đỉnh có khả năng lớn nhất trong đồ thị*/
#define MAXN 150
#define MAX_VALUE 10000
/* Số đỉnh thật trong đồ thị */
const unsigned n = 6;
/*Ma trận trọng số của đồ thị*/
const int A[MAXN][MAXN] = {
  { 0, 5, 0, 0, 7, 7 },
  { 5, 0, 5, 0, 0, 0 },
  { 0, 5, 0, 6, 5, 0 },
  { 0, 0, 6, 0, 3, 3 },
  { 7, 0, 5, 3, 0, 5 },
  { 7, 0, 0, 3, 5, 0 }
};
char used[MAXN];
unsigned minCycle[MAXN], cycle[MAXN];
int curSum, minSum;

void printCycle(void)
{ unsigned i;
  printf("Chu trình Hamilton nhỏ nhất: 1");
  for (i = 0; i < n - 1; i++) printf(" %u", minCycle[i] + 1);
  printf(" 1, độ dài %d\n", minSum);
}
```

```
/* Tìm Chu trình Hamilton nhỏ nhất */
void hamilton(unsigned i, unsigned level)
{ unsigned k;
  if ((0 == i) && (level > 0)) {
    if (level == n) {
      minSum = curSum;
      for (k = 0; k < n; k++)
        minCycle[k] = cycle[k];
    }
    return;
  }
  if (used[i])
    return;
  used[i] = 1;
  for (k = 0; k < n; k++)
    if (A[i][k] && k != i) {
      cycle[level] = k;
      curSum += A[i][k];
      if (curSum < minSum) /*Gián đoạn vòng lặp*/
        hamilton(k, level + 1);
      curSum -= A[i][k];
    }
  used[i] = 0;
}

int main() {
  unsigned k;
  for (k = 0; k < n; k++) used[k] = 0;
  minSum = MAX_VALUE;
  curSum = 0;
  cycle[0] = 1;
  hamilton(0, 0);
  printCycle();
  return 0;
}
```

Kết quả thực hiện chương trình:

Chu trình Hamilton nhỏ nhất: 1 2 3 5 4 6 1, độ dài 28.

Thuật toán được trình bày và cách triển khai trên thực tế không thể áp dụng cho các đồ thị lớn: với khả năng tính toán hiện tại cho

đồ thị có hơn 50 đỉnh, có thể tìm thấy các ví dụ mà thuật toán sẽ hoạt động lâu đến mức không thể chấp nhận được. Do ứng dụng rộng rãi của nó, bài toán đã được nghiên cứu rất chi tiết trong các tài liệu [Reinelt-1994]. Có hàng tá (!) Trong số các thuật toán cho giải pháp của nó, một số trong số đó tìm ra nghiệm của độ phức tạp đa thức cho hầu hết các đồ thị (tất cả ngoại trừ logarit phụ thuộc vào n số đồ thị).

Trước khi chuyển sang loại chu trình thú vị tiếp theo trong đồ thị, chúng ta sẽ xem xét một câu hỏi khác liên quan đến các chu trình Hamilton và tìm kiếm của chúng. Đó là một câu hỏi làm thế nào để xác suất tồn tại của một chu trình Hamilton trong một đồ thị tùy ý thay đổi so với kết nối của đồ thị. Rõ ràng là một đồ thị đầy đủ (xem định nghĩa 5.8) Luôn luôn là Hamilton (bất kỳ sự sắp xếp nào của các đỉnh mà chúng ta lấy, chúng sẽ luôn tạo thành HC). Khi đồ thị gần như ầy đủ (chính xác hơn là gần như liên thông hoàn toàn), xác suất tồn tại HC là rất cao, số chu trình Hamilton cũng vậy. Con số này giảm mạnh khi số lượng cạnh giảm dần. Ở một cực khác, khi khả năng kết nối của đồ thị gần bằng 2, xác suất tồn tại của một chu trình Hamilton là nhỏ. Tại điểm tới hạn giữa hai điểm cực trị này, xác suất tồn tại của chu trình Hamilton phân bố đều giữa 0 và 1. Lý thuyết cho thấy rằng phân bố xác suất đồng đều này được thỏa mãn trong trường hợp chúng ta có kết nối trung bình của một đồ thị $\ln n + \ln \ln n$ [Christofides-1975].

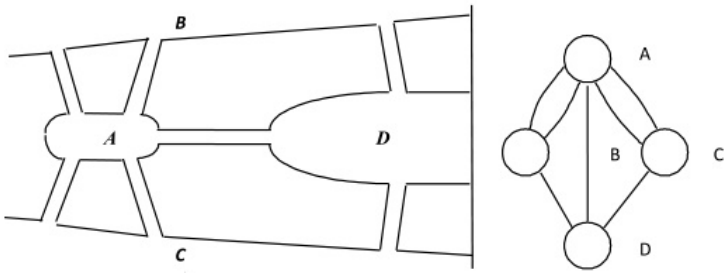
Bài tập

▷ **5.35.** Hãy tìm một số lớp của đồ thị "đặc biệt" mà việc tìm kiếm chu trình Hamilton là "dễ dàng" (được thực hiện với độ phức tạp đa thức). Các ví dụ về vấn đề này được thảo luận trong **??**.

5.4.5. Chu trình Euler

Sự khởi đầu của lý thuyết đồ thị được coi là sự xem xét của một loại chu trình đặc biệt, được đặt theo tên nhà nghiên cứu đầu tiên L. Euler. Thành phố Königsberg (nay là Kaliningrad) có bảy cây cầu bắc qua sông Pregel, được xây dựng như trong Hình 5.17.

Mọi người ở đó thường tự hỏi liệu có thể đi bộ qua thành phố để người ta có thể đi qua chỉ một lần trong những cây cầu và chuyển



Hình 5.17. 7 cầu ở Königsberg và đồ thị của chúng

tham quan sẽ kết thúc ở vị trí xuất phát hay không. Euler đã trình bày các cây cầu bằng đồ thị đa điểm (Hình 5.17) và chỉ ra rằng điều này là không thể.

Định nghĩa 5.16. Một đồ thị được kết nối được đưa ra. Chu trình mà mỗi cạnh tham gia đúng một lần được gọi là *chu trình Euler*. Một đa đồ thị được gọi là một đồ thị Euler nếu có một chu trình Euler trong đó.

Đường dẫn của Euler trong đồ thị cũng được xác định tương tự.

Định lý 5.4 (Euler)**.** *Một đồ thị đa hướng không định hướng được kết nối chứa một chu trình Euler, khi và chỉ khi tất cả các đỉnh của đồ thị có bậc chẵn. (được coi là định lý đầu tiên trong lý thuyết đồ thị được biểu diễn bởi Euler mà không cần chứng minh).*

Bổ đề 5.1. *Một đồ thị đa hướng không định hướng liên kết chứa một đường Euler khi và chỉ khi có đúng hai đỉnh có bậc lẻ.*

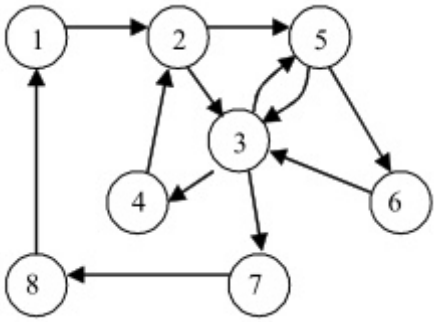
Định lý 5.5. *Một đồ thị đa phương liên kết lỏng chứa chu trình Euler khi và chỉ khi bậc độ của đầu vào $d^+(i)$ tại mỗi đỉnh i bằng bậc độ của đầu ra $d^-(i)$, tức là: $d^+(i) = d^-(i)$, với mỗi $i \in V$.*

Bổ đề 5.2. *Một đồ thị có định hướng được ghép nối lỏng lẻo chứa đường đi Euler khi và chỉ khi có đúng hai đỉnh i và j sao cho $d^+(i) = d^-(i) + 1$, $d^+(j) = d^-(j)) - 1$ và $d^+(k) = d^-(k)$, với mỗi $k \in V, k \in i, k \neq j$.*

Mệnh đề 5.2. *Trong đa đồ thị Euler, một chu trình Euler có thể được xây dựng bắt đầu từ bất kỳ đỉnh ban đầu nào.*

Bổ đề 5.3. *Nếu có một đường Euler trong một đồ thị đa phương được ghép nối lỏng lẻo, thì đường ban đầu là đỉnh có bậc độ ở đầu ra lớn hơn bậc độ ở đầu vào.*

Một ví dụ về đồ thị Euler có định hướng được thể hiện trong Hình **??**. Một chu trình Euler có thể có trong nó là $(1,2)$, $(2,3)$, $(3,4)$, $(4,2)$, $(2,5)$, $(5,3)$, $(3,5)$, $(5,6)$, (6.3), (3.7), (7.8), (8.1).



Hình 5.18. Chu trình Euler

Thuật toán tìm chu trình Euler

Chúng ta bắt đầu duyệt đồ thị từ đỉnh i bất kỳ. Chúng ta vô tình tìm thấy một cạnh (i,j) với nó và đánh dấu nó là đã thăm. Chúng ta tiếp tục với đỉnh j: đối với nó, chúng ta tìm thấy một cạnh không được chờ đợi (j,k), đánh dấu nó là đã thăm và chuyển đến k. Tiếp tục theo cách này, đến một lúc nào đó chúng ta sẽ thấy mình ở đỉnh ban đầu i, tức là. chúng ta sẽ đóng chu trình (Tại sao?).

Nếu tất cả các cạnh của đồ thị đã được đánh dấu, thì chu trình này là Euler. Nếu có các cạnh không được thăm thì ta tìm được đỉnh x, thuộc chu trình mới tìm được và có ít nhất một cạnh không được thăm. Từ thực tế là mỗi đỉnh phải có mức độ chẵn, theo đó x là sự cố với một số chẵn (tức là ít nhất hai) các cạnh không được chọn. Từ x, chúng ta bắt đầu xây dựng một vòng lặp theo cách đã được mô tả, cho đến khi chúng ta quay trở lại nó. Chúng ta nhận được một chu trình thứ hai, mà chúng ta kết hợp với chu trình đầu tiên (điểm chung của chúng sẽ là đỉnh x). Do đó, sau một số bước hữu hạn, tất cả các sườn sẽ được đưa vào một chu trình chung - chu trình Euler mong muốn.

Nếu chúng ta tìm đường đi Euler, chúng ta có thể áp dụng cách sửa đổi đơn giản sau của thuật toán trên: chúng ta nối các đỉnh có bậc lẻ với một cạnh (từ Hệ quả 1 theo đó có đúng hai đỉnh như vậy) và tìm chu trình Euler bằng cách thuật toán được mô tả ở trên. Sau khi loại bỏ phần sườn đã thêm, chúng ta sẽ có được đường dẫn Euler mong muốn.

Cho đồ thị được biểu diễn bằng ma trận lân cận A[][]. Để thực hiện, chúng ta sẽ sử dụng hai ngăn xếp: stack[], cho vòng lặp hiện được xây dựng và cStack[] - để hợp nhất tất cả các vòng được xây dựng cho đến nay. Cả hai ngăn xếp ban đầu sẽ trống. Mã giả sau đây mô tả chi tiết hơn việc triển khai thuật toán:

```
<thêm đỉnh bất kỳ i vào stack>;
while (<stack khác trống>) {
  <Lấy một đỉnh i ở đỉnh stack không loại nó>;
    if (<i có đỉnh kề>) {
      <Lấy đỉnh kề bất kỳ j của i>;
      <đặt j stack>;
      A[j][i] = 0; A[i][j] = 0; /* Loại bỏ cạnh của đồ thị */
  } else {
     <Loại i khỏi stack>;
     <Đưa i vào cStack>;
  }
}
```

Triển khai đầy đủ sau đây. Đồ thị mẫu là của Hình 5.18.

Chương trình 5.12. Chu trình Euler (511eurler.c)

```
#include <stdio.h>
/* Khả năng ớn nhất đỉnh của đồ thị */
#define MAXN 100
/* Số đỉnh thật của đồ thị*/
const unsigned n = 8;
/* Ma trận cạnh kề trong đồ thị */
char A[MAXN][MAXN] = {
 { 0, 1, 0, 0, 0, 0, 0, 0 },
 { 0, 0, 1, 0, 1, 0, 0, 0 },
 { 0, 0, 0, 1, 1, 0, 1, 0 },
 { 0, 1, 0, 0, 0, 0, 0, 0 },
 { 0, 0, 1, 0, 0, 1, 0, 0 },
```

```
  { 0, 0, 1, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 1 },
  { 1, 0, 0, 0, 0, 0, 0, 0 } };
/*Kiểm tra xem có chu trình Euler không (theo định lý Euler)*/
char isEulerGraph(void)
{ unsigned i, j;
  for (i = 0; i < n; i++) {
    int din = 0, dout = 0;
    for (j = 0; j < n; j++) {
      if (A[i][j]) din++;
      if (A[j][i]) dout++;
    }
    if (din != dout) return 0;
  }
  return 1;
}

/*Tìm chu trình Euler*/
void findEuler(int i)
{ unsigned cStack[MAXN * MAXN], stack[MAXN * MAXN];
  unsigned k, j, cTop = 0, sTop = 1;
  stack[sTop] = i;
  while (sTop > 0) {
    i = stack[sTop];
    for (j = 0; j < n; j++)
      if (A[i][j]) {
      A[i][j] = 0; i = j;
      break;
    }
  if (j < n)
    stack[++sTop] = i;
  else
    cStack[++cTop] = stack[sTop--];
}

  printf("Chu trình Euler là: ");
  for (k = cTop; k > 0; k--) {
    printf("%u ", cStack[k] + 1);
  }
  printf("\n");
```

```
}

int main(){
  if (isEulerGraph()) findEuler(0);
  else
    printf("Đồ thị không phải là Euler!");
  return 0;
}
```

Kết quả thực hiện chương trình:

```
chu trình Euler là: 1 2 3 4 2 5 3 5 6 3 7 8 1
```

Bài tập

▷ **5.36.** Hãy chứng minh định lý Euler và các hệ quả của nó.

▷ **5.37.** So sánh bài toán tìm chu trình Hamilton với bài toán tìm chu trình Euler. Đâu là lý do cho việc đầu tiên khó hơn đáng kể so với lần thứ hai?

5.4.6. Luồng trong đồ thị

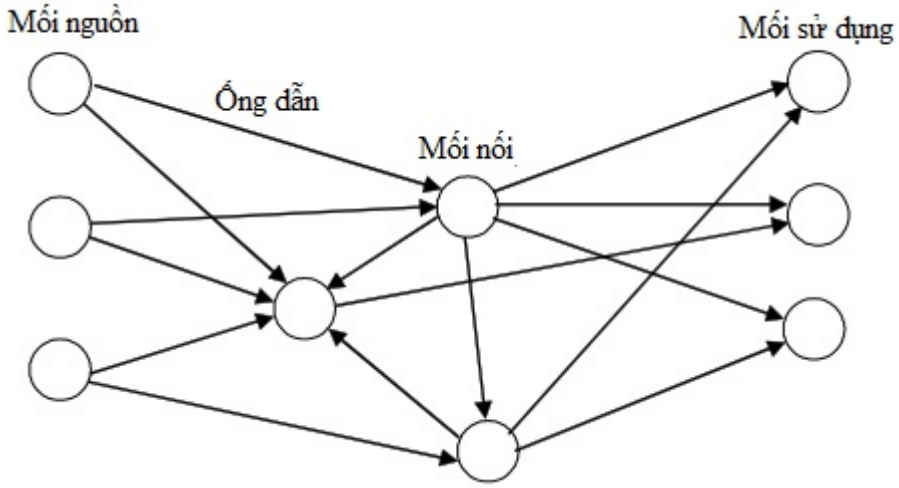
Giả sử một đồ thị có hướng G (V, E) trong đó các đỉnh được chia thành ba tập hợp:

- nhiều nguồn nguyên liệu;
- nhiều người tiêu dùng;
- tập hợp các đỉnh trung gian - các kết nối mà qua đó vật liệu có thể được phân phối đến các kết nối hoặc người tiêu dùng khác.

Mỗi nguồn được so sánh với một con số - lượng nguyên liệu tối đa mà nó có thể phân phối và đối với mỗi người tiêu dùng được cung cấp số lượng nguyên liệu tối đa mà nó có thể nhận được. Các cạnh định hướng (i, j)của đồ thị có thể được hiểu là "đường ống" nối các đỉnh i và j, và đối với mỗi cạnh định hướng, chúng ta xác định:

1) Hàm c: $c(i, j)$ giới hạn lượng vật liệu có thể đi qua cạnh $(i, j), c(i, j) \geq 0$, với mọi $(i, j) \in E$.

2) Hàm t: $t(i, j)$ thể hiện chi phí (chi phí) vận chuyển vật liệu qua sườn (i, j).



Hình 5.19. Mối chuyển đổi

Bài toán chung của việc tìm ra bài toán dòng chảy mạng chi phí tối thiểu là tìm ra phương án vận chuyển nguyên liệu từ các nguồn đến người tiêu dùng sao cho tổng chi phí vận chuyển là nhỏ nhất. Năm 1960, Ford và Fulkerson đã đề xuất một thuật toán (gọi là kilter) để giải quyết vấn đề này. Bài toán có nhiều cách hiểu khác nhau và các trường hợp đặc biệt hữu ích, ví dụ:

Bài toán giá trị tối ưu

Cho trước là n máy, n bộ phận và ma trận vuông $A_{n\times n}$: a_{ij} cho biết chi phí chế tạo bộ phận i từ máy j. Tìm sơ đồ tạo các chi tiết (hoán vị p của các phần tử từ 1 đến n) sao cho tổng chi phí là nhỏ nhất (tức là tổng nhỏ nhất $\sum_{i\in V} a_{ip(i)}$.

Tác vụ này có thể được giảm xuống một tác vụ luồng tối thiểu như sau:

- Chúng ta coi máy móc là nguồn cung cấp một đơn vị vật chất.
- Chúng ta coi các chi tiết là người tiêu dùng yêu cầu một đơn vị vật liệu.
- Mỗi sườn có công suất đơn vị và giá bằng giá trị tương ứng trong bảng cho sẵn trong điều kiện.
- Không có đỉnh trung gian (kết nối).

Biến thể ngược lại của nguyên công cũng có thể xảy ra: bảng vuông cho thấy thu nhập từ việc sản xuất các chi tiết (tùy thuộc vào loại máy nào sẽ sản xuất nó) và chúng ta đang tìm kiếm lợi nhuận tối đa.

Ngoài việc giảm đến một luồng tối ưu, vấn đề thứ hai cũng có thể được giải quyết bằng cái gọi là thuật toán Hungary [Shishkov-1995].

- Lưu lượng luồng cực đại

Một biến thể của bài toán trên là tìm dòng chảy tối đa. Các bài toán kiểu này được sử dụng trong các mạng thông tin liên lạc, trong việc lập kế hoạch cho các mạng chuyển vật chất (khí đốt, nước, nhiên liệu) và các mạng khác. Thuật toán giải nó đã được biết đến từ năm 1950 và một lần nữa thuộc về Ford và Fulkerson.

Chúng ta xem xét một đồ thị có trọng số $G(V, E)$, trong đó trên mỗi cạnh (i, j) một số không âm $c(i, j)$ được ánh xạ, nghĩa là dung lượng (thông lượng). Hai đỉnh đặc biệt được cố định trong đồ thị: nguồn s và người tiêu dùng t.

Định nghĩa 5.17. Luồng trong đồ thị được gọi là hàm $f : E \to Z$, phù hợp với số $f(i, j)$ trên mỗi cạnh (i, j) và có ba tính chất sau:

1) Lưu lượng không vượt quá khả năng của "đường ống", tức là. với mỗi cạnh $(i, j) \in E$, $f(i, j) \leq c(i, j)$ được thỏa mãn.

2) Nếu dòng chảy là âm, có nghĩa là có một "dòng chảy ngược". Nói chung, nếu luồng từ i đến j bằng $f(i, j)$, thì nhất thiết phải tuân theo luồng từ j đến i là $-f(i, j)$, với mỗi $(i, j) \in E$.

3) Độ lớn của dòng vào và dòng ra bằng nhau, tức là. với mỗi đỉnh$i \in V \setminus \{s, t\}$ thỏa mãn $\sum\limits_{j:(i,j)\in E} f(i, j) = 0$.

Bài toán tìm giá trị của luồng cực đại từ s đến t, tức là một hàm f như vậy (thỏa mãn ba điều kiện trên) mà $\sum\limits_{i:(i,t)\in E} f(i, t) = 0$là cực đại.

Thuật toán Ford-Fulkerson

Ý tưởng chính của thuật toán là liên tục tăng luồng khi có thể:

1) Chúng ta bắt đầu với luồng không: $f(i,j) = 0$, với mỗi $(i,j) \in E$.

2) Chúng ta tìm thấy một *đường đi tăng* trong đồ thị. *Đường đi tăng dần* ($s = v_0, v_1, ..., v_k = t$) là đường đi từ s đến t sao cho mọi hai đỉnh kề nhau v_i, $v_{i+1} (i = 0, 1, ..., k-1)$ của đường đi được hoàn thành $c(v_i, v_{i+1}) > 0$.

3) Nếu một đường dẫn như vậy không tồn tại, thì theo đó chúng ta đã tìm thấy luồng cực đại trong đồ thị.

4) Nếu không, chúng ta tăng các luồng âm và dương lên số dương lớn nhất p, cho phép tìm được đường dẫn, tức là:

$$p = \min_{i=0,...,k-1} \{c(v_i, v_{i+1})\}$$

$$f(v_i, v_{i+1}) = f(v_i, v_{i+1}) + p, i = 0, 1, ..., k-1$$

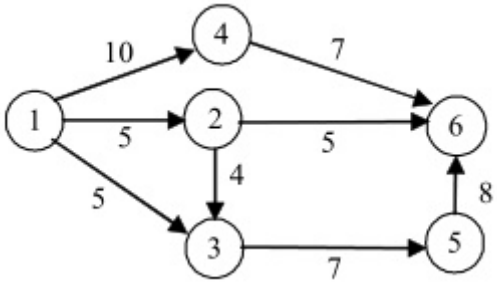
$$f(v_{i+1}, v_i) = f(v_{i+1}, v_i) - p, i = 0, 1, ..., k-1$$

Chúng ta sửa đổi trọng số (băng thông) của các cạnh của biểu đồ như sau:

$$c(v_i, v_{i+1}) = c(v_i, v_{i+1}) - p, i = 0, 1, ..., k-1$$

$$c(v_{i+1}, v_i) = c(v_{i+1}, v_i) + p, i = 0, 1, ..., k-1$$

Chúng ta quay lại bước 2, nơi chúng ta tìm kiếm một con đường tăng mới, v.v.



Hình 5.20. Luồng cực đại trong đồ thị

Hãy nhìn vào đồ thị trong Hình 5.20., Và chọn nguồn $s = 1$ và người tiêu dùng $t = 6$. Thuật toán sẽ tìm ra 5 đường tăng liên tiếp:

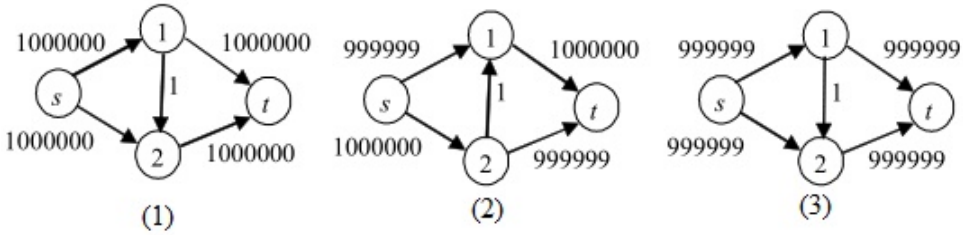
1. $(1, 2, 3, 5, 6)$: sườn tối thiểu của đường là $(2, 3)$ và trọng lượng của nó xác định mức tăng của tổng lưu lượng: trong trường

hợp là 4. Chúng ta sửa đổi trọng lượng của các sườn - ví dụ, từ đỉnh 2 đến đỉnh 3, chúng ta thu được dòng ngược chiều dài 4, dòng chảy này (xuất hiện ở bước 3) được sử dụng trong việc xây dựng đường tăng tiếp theo.

2. $(1,2,6)$: cạnh nhỏ nhất của con đường là $(1,2)$ với trọng lượng 1 (vì chúng ta đã giảm trọng lượng của nó đi 4 ở bước 1).
3. $(1,3,2,6)$: cạnh nhỏ nhất của đường là $(3,2)$ với trọng số 4.
4. $(1,3,5,6)$: cạnh nhỏ nhất của đường là $(1,3)$ với trọng số 1.
5. $(1,4,6)$: cạnh nhỏ nhất của đường là $(4,6)$ với trọng số 7.

Như vậy tổng kích thước của luồng lớn nhất là 17.

Có thể chứng minh rằng nếu không có đường đi tăng trong biểu đồ, thì dòng kết quả là cực đại [Christofides-1975]. Tuy nhiên, có thể xảy ra (nếu trong bước 2 của thuật toán bạn luôn chọn một con đường tăng dần tùy ý), độ phức tạp trong trường hợp xấu nhất sẽ đạt đến độ lớn của luồng tối đa, tức là. sẽ phụ thuộc vào trọng số của các cạnh của đồ thị. Do đó, ngay cả đối với các đồ thị rất đơn giản (xem Hình 5.21), việc tìm ra lưu lượng cực đại có thể mất một thời gian dài ngoài dự kiến. Với ví dụ trong hình, độ phức tạp nhỏ nhất bằng độ lớn của luồng cực đại: Trong bước đầu tiên, chúng ta chọn một đường đi tăng dần $s-1-2-t$ và tăng luồng một. Chúng ta nhận được đồ thị (2), sau đó chúng ta đi theo đường tăng dần $s-2-1-t$ - chúng ta nhận được đồ thị (3), tương tự như đồ thị mà chúng ta đã bắt đầu, v.v. Giải pháp sẽ tìm ra 1.000.000 con đường gia tăng [Cormen, Leiserson, Rivest-1997].



Hình 5.21. Một ví dụ về sự kém hiệu quả trong việc chọn ngẫu nhiên một đường dẫn tăng.

Để giải quyết vấn đề thứ hai, chúng ta sẽ sử dụng định lý sau [Cormen, Leiserson, Rivest-1997]:

Định lý 5.6 (Luồng cực đại). *Nếu đường được chọn ở bước 2) luôn là số lượng đỉnh tối thiểu liên quan, thì độ phức tạp của thuật toán trong trường hợp xấu nhất là $\Theta(n^5)$, tức là. sẽ có nhiều nhất n 3 đường tăng, mỗi đường có độ phức tạp $\Theta(n^2)$.*

Trong phương án sau đây, các đường đi tăng dần được định vị bởi truyền theo độ sâu, đồ thị được biểu diễn bằng ma trận trọng số A[][] và dòng chảy được giữ trong ma trận F[][].

Chương trình 5.13. Thuật toán Ford-Fulkerson (512fordfulk.c)

```
#include <stdio.h>
/* Số đỉnh cực đại có thể trong đồ thị*/
#define MAXN 100
#define MAX_VALUE 10000
const unsigned n = 10; /* Số đỉnh của đồ thị*/
const unsigned s = 1; /* Đỉnh nguồn*/
const unsigned t = 6; /* Đỉnh người sử dụng*/
/*Ma trận trong số của đồ thị */
int A[MAXN][MAXN] = {
  { 0, 5, 5, 10, 0, 0 },
  { 0, 0, 4, 0, 0, 5 },
  { 0, 0, 0, 0, 7, 0 },
  { 0, 0, 0, 0, 0, 7 },
  { 0, 0, 0, 0, 0, 8 },
  { 0, 0, 0, 0, 0, 0 }
};
int F[MAXN][MAXN];
unsigned path[MAXN];
char used[MAXN], found;

void updateFlow(unsigned pl)
{ int incFlow = MAX_VALUE;
  unsigned i;
  printf("Đường tăng tìm được: ");
  for (i = 0; i < pl; i++) {
    unsigned p1 = path[i];
    unsigned p2 = path[i + 1];
    printf("%u, ", p1+1);
    if (incFlow > A[p1][p2]) incFlow = A[p1][p2];
  }
```

```
    printf("%u \n", path[pl]+1);
    for (i = 0; i < pl; i++) {
      unsigned p1 = path[i];
      unsigned p2 = path[i + 1];
      F[p1][p2] += incFlow;
      F[p2][p1] -= incFlow;
      A[p1][p2] -= incFlow;
      A[p2][p1] += incFlow;
    }
}

void DFS(unsigned i, unsigned level)
{ unsigned k;
  if (found) return;
  if (i == t-1) {
    found = 1;
    updateFlow(level - 1);
  }
  else
    for (k = 0; k < n; k++)
      if (!used[k] && A[i][k] > 0) {
        used[k] = 1;
        path[level] = k;
        DFS(k, level + 1);
        if (found) return;
      }
  }

int main() {
  unsigned i, j;
  int flow;
  /* 1) một luồng trống được khởi tạo*/
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) F[i][j] = 0;
/* 2) tìm thấy một đường ngày càng tăng trong khi có thể*/
do {
  for (i = 0; i < n; i++) used[i] = 0;
  found = 0;
  used[s-1] = 1;
  path[0] = s-1;
```

```
    DFS(s-1, 1);
  } while (found);
    /* In luồng ra */
    printf("Lưu lượng tối đa qua đồ thị: \n");
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) printf("%4d", F[i][j]);
      printf("\n");
   }
    printf("\n");
    flow = 0;
    for (i = 0; i < n; i++) flow += F[i][t-1];
    printf("Với độ lớn : %d\n", flow);
    return 0;
}
```

Kết quả thực hiện chương trình:

```
Đã tìm thấy đường dẫn tăng: 1, 2, 3, 5, 6
Đã tìm thấy đường dẫn tăng: 1, 2, 6
Đã tìm thấy đường dẫn tăng: 1, 3, 2, 6
Đã tìm thấy đường dẫn tăng: 1, 3, 5, 6
Đã tìm thấy đường dẫn tăng: 1, 4, 6
Lưu lượng tối đa qua đồ thị:
 0  5  5  7  0  0  0  0  0  0
-5  0  0  0  0  5  0  0  0  0
-5  0  0  0  5  0  0  0  0  0
-7  0  0  0  0  7  0  0  0  0
 0  0 -5  0  0  5  0  0  0  0
 0 -5  0 -7 -5  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
Với kích thước: 17
```

Chúng ta để người đọc sửa đổi chương trình trên để nó tìm ra các đường đi tăng dần bằng cách đi ngang theo chiều rộng, điều này sẽ đảm bảo độ dài đường đi tối thiểu, vì nó sẽ thỏa mãn điều kiện của định lý.

Ngoài ra, chúng ta sẽ nói rằng có các thuật toán để tìm luồng cực

đại trong đồ thị có độ phức tạp $\Theta(n^4)$ và thậm chí $\Theta(n^3)$ [Cormen, Leiserson, Rivest-1997].

Bài tập

▷ **5.38.** Chứng minh rằng nếu không có đường đi tăng trong đồ thị, thì dòng kết quả là cực đại.

- Nhiều nguồn và người tiêu dùng

Nếu có nhiều hơn một nguồn và người tiêu dùng được đưa ra trong bài toán, chúng ta có thể dễ dàng giảm nó xuống nguồn vừa được xem xét. Gọi các nguồn là $s_1, s_2, ..., s_p$ và người tiêu dùng $t_1, t_2, ..., t_q$. Chúng ta thêm một đỉnh mới s trong cột được gọi là siêu nguồn và một đỉnh mới t - siêu tích lũy. Ngoài hai mẹo mới này, chúng ta thêm cạnh:

- (s, s_i), sao cho $c(s, s_i) = +\infty$, với $i = 1, 2, ..., p$.
- (t_j, t), sao cho $c(t_j, t) = +\infty$, với $j = 1, 2, ..., q$.

Sử dụng thuật toán Ford-Fulkerson từ đoạn trước, chúng ta tìm thấy luồng lớn nhất từ s đến t, sẽ bằng với tìm kiếm trong phiên bản gốc của bài toán. (Tại sao?)

Bài tập

▷ **5.39.** Chứng minh rằng lưu lượng tối đa được tìm thấy bởi thuật toán Ford-Fulkerson trên đồ thị đã sửa đổi là lưu lượng tối đa cho đồ thị đầu ra, có một số nguồn và một số khách hàng.

- Công suất của các đỉnh

Có thể làm phức tạp thêm bài toán bằng cách đặt giới hạn cho lưu lượng đi qua mỗi đỉnh. Cho một hàm $v(i), v(i) \geq 0, i \in V$ sao cho:

$$\sum_{j \in V} f(i, j) \leq v(i), i \in V$$

Chúng ta đang tìm kiếm một dòng chảy tối đa từ nguồn s đến người tiêu dùng t. Ở đây, chúng ta cũng có thể giải quyết vấn đề

bằng cách giảm nó xuống bài toán tiêu chuẩn là tìm dòng chảy tối đa. Với mục đích này, chúng ta sẽ xây dựng một đồ thị mới $G'(V', E')$, trong đó:

1) Hai đỉnh của V' sẽ tương ứng với mỗi đỉnh $i \in V$: i_1 và i_2, sẽ được nối với nhau bởi một cạnh (i_1, i_2). Độ đi qua $c(i_1, i_2)$ của cạnh này sẽ bằng $v(i)$.

2) Mỗi cạnh (i, j) của G được chuyển đến G' là sườn (i_2, j_1).

3) Trong đồ thị mới G' sẽ không có hàm số v giới hạn dòng chảy qua các đỉnh.

Như trong đoạn trước, luồng tối đa trong G' được tìm thấy bởi thuật toán Ford-Fulkerson sẽ là mức tối đa cho đồ thị G. (Tại sao?)

Các ứng dụng khác của bài toán luồng cực đại sẽ được thảo luận trong **??** và **??**.

Bài tập

▷ **5.40.** Chứng minh rằng lưu lượng tối đa được tìm thấy bởi thuật toán Ford-Fulkerson trên đồ thị đã sửa đổi là lưu lượng tối đa cho đồ thị đầu ra.