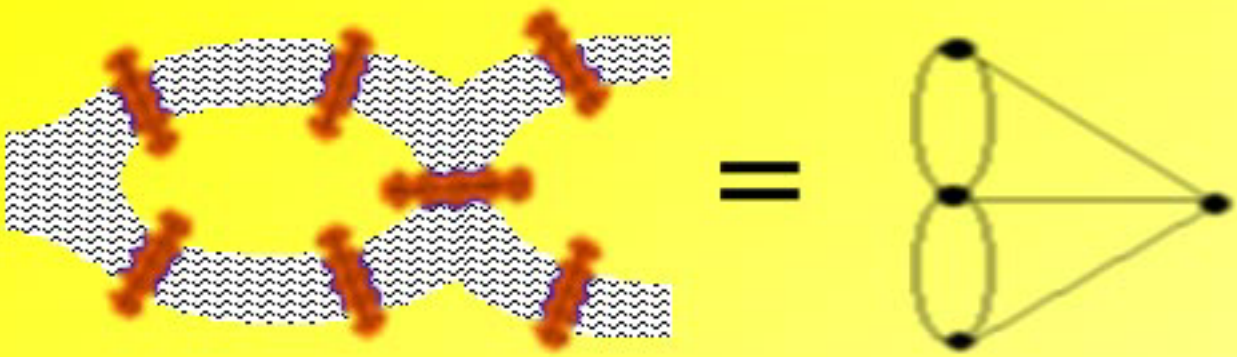
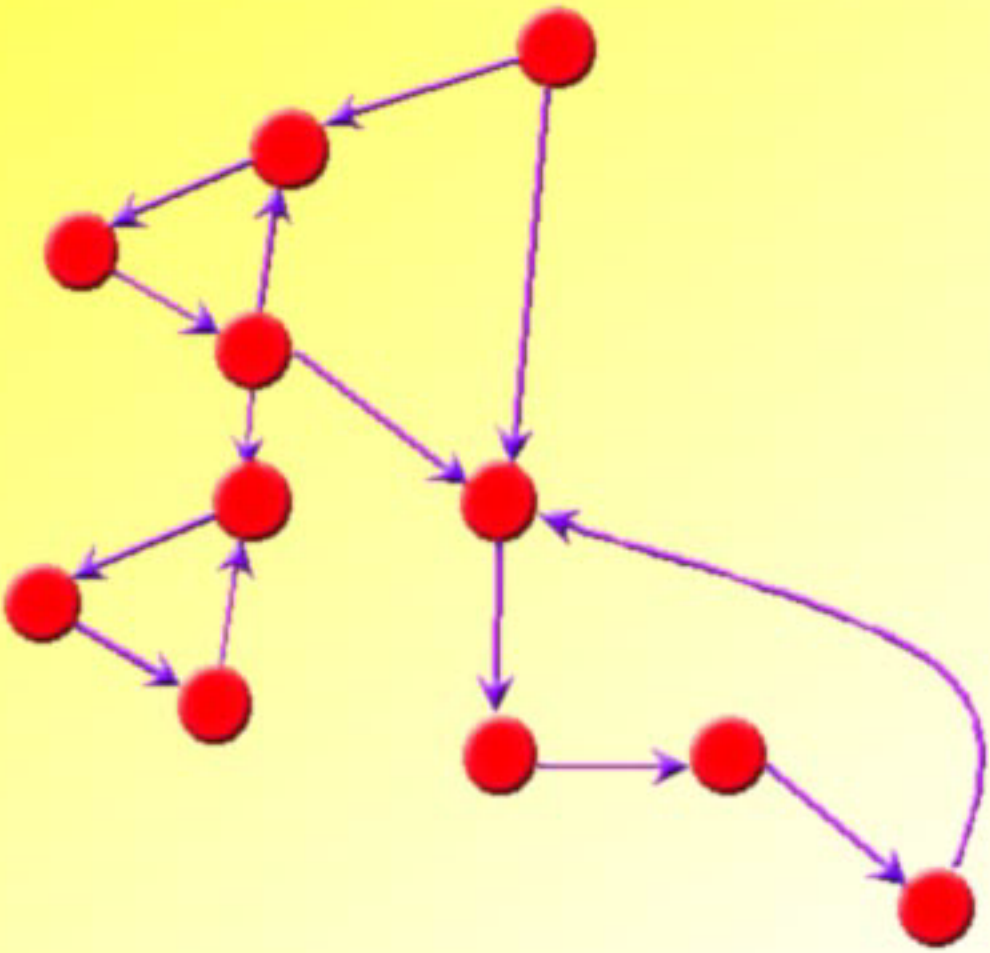
β test



LÝ THUYẾT ĐỒ THỊ



MỤC LỤC

§0. MỞ ĐẦU

Leonhard Euler
(1707 - 1783)

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

Mặc dù Lý thuyết đồ thị đã được khoa học phát triển từ rất lâu nhưng lại có nhiều ứng dụng hiện đại. Đặc biệt trong khoảng vài mươi năm trở lại đây, cùng với sự ra đời của máy tính điện tử và sự phát triển nhanh chóng của Tin học, Lý thuyết đồ thị càng được quan tâm đến nhiều hơn. Đặc biệt là các **thuật toán trên đồ thị** đã có nhiều ứng dụng trong nhiều lĩnh vực khác nhau như: Mạng máy tính, Lý thuyết mã, Tối ưu hoá, Kinh tế học v.v... Chẳng hạn như trả lời câu hỏi: Hai máy tính trong mạng có thể liên hệ được với nhau hay không ?; hay vấn đề phân biệt hai hợp chất hoá học có cùng công thức phân tử nhưng lại khác nhau về công thức cấu tạo cũng được giải quyết nhờ mô hình đồ thị. Hiện nay, môn học này là một trong những kiến thức cơ sở của bộ môn khoa học máy tính.

Trong phạm vi một chuyên đề, không thể nói kỹ và nói hết những vấn đề của lý thuyết đồ thị. Tập bài giảng này sẽ xem xét lý thuyết đồ thị dưới góc độ người lập trình, tức là khảo sát những **thuật toán cơ bản nhất** có thể **dễ dàng cài đặt trên máy tính** một số ứng dụng của nó. Các khái niệm trừu tượng và các phép chứng minh sẽ được diễn giải một cách hình thức cho đơn giản và dễ hiểu chứ không phải là những chứng minh chặt chẽ dành cho người làm toán. Công việc của người lập trình là đọc hiểu được ý tưởng cơ bản của thuật toán và cài đặt được chương trình trong bài toán tổng quát cũng như trong trường hợp cụ thể. Thông thường sau quá trình rèn luyện, hầu hết những người lập trình gần như phải **thuộc lòng** các mô hình cài đặt, để khi áp dụng có thể cài đặt đúng ngay và hiệu quả, không bị mất thời giờ vào các công việc gỡ rối. Bởi việc gỡ rối một thuật toán tức là phải dò lại từng bước tiến hành và tự trả lời câu hỏi: "Tại bước đó nếu đúng thì phải như thế nào ?", đó thực ra là tiêu phí thời gian vô ích để chứng minh lại tính đúng đắn của thuật toán trong trường hợp cụ thể, với một bộ dữ liệu cụ thể.

Trước khi tìm hiểu các vấn đề về lý thuyết đồ thị, trước hết bạn phải có **kỹ thuật lập trình khá tốt**, ngoài ra nếu đã có tìm hiểu trước về các kỹ thuật vét cạn, quay lui, một số phương pháp tối ưu hoá, các bài toán quy hoạch động thì sẽ giúp ích nhiều cho việc đọc hiểu các bài giảng này.

Tác giả xin chân thành cảm ơn các thầy giáo Nguyễn Đức Nghĩa, Nguyễn Thanh Tùng, Nguyễn Tô Thành (Khoa CNTT, ĐHBKHN), thầy giáo Nguyễn Xuân My (Khoa Toán - Cơ - Tin, ĐHKHTNHN), cô Hồ Cẩm Hà (Khoa Toán - Tin, ĐHSPHN) đã hỗ trợ rất nhiều kiến thức quý báu. Xin cảm ơn các bạn lớp chuyên tin ĐHBKHN khoá 1991 - 1994 đã đóng góp những ý kiến bổ ích. Mọi khiếm khuyết của tài liệu thuộc về trách nhiệm riêng của cá nhân tác giả.

Tác giả rất mong nhận được các ý kiến đóng góp về tài liệu này kể cả các vấn đề lý thuyết cũng như hệ thống bài tập ứng dụng. Xin liên lạc tại địa chỉ:
Lê Minh Hoàng - Khoa Toán - Tin học trường ĐHSPHN - ĐT: 04 7337983
e-mail: minhhoang_le@yahoo.com hoặc minhhoang_le@vol.vnn.vn

§1. CÁC KHÁI NIỆM CƠ BẢN

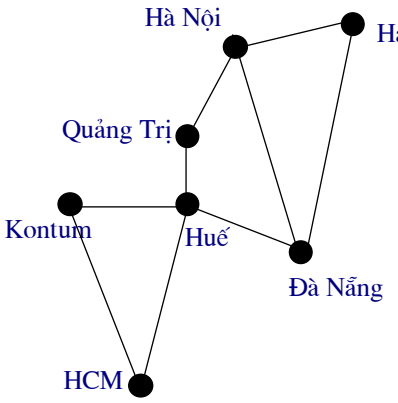
I. ĐỊNH NGHĨA ĐỒ THỊ (GRAPH)

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

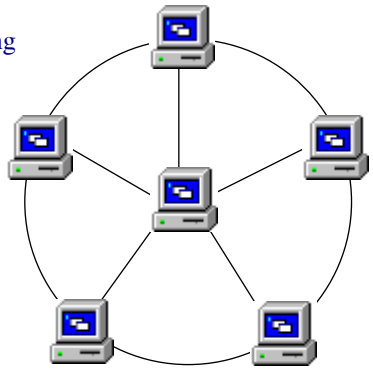
$$G = (V, E)$$

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

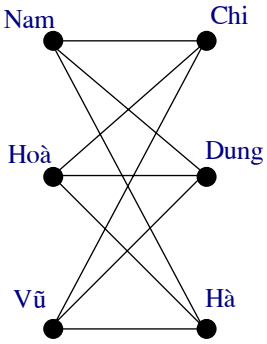
Một số hình ảnh của đồ thị:



Sơ đồ giao thông



Mạng máy tính



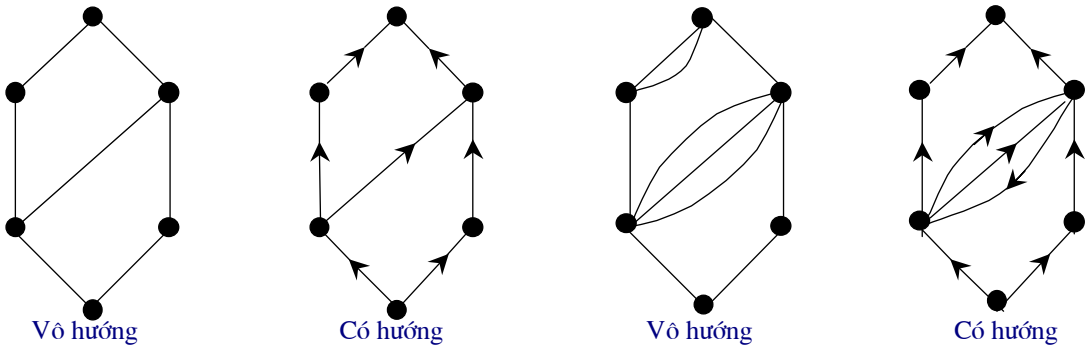
Quan hệ quen biết

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

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

1. G được gọi là **đơn đồ thị** nếu giữa hai đỉnh u, v của V có nhiều nhất là 1 cạnh trong E nối từ u tới v.
2. G được gọi là **đa đồ thị** nếu giữa hai đỉnh u, v của V có thể có nhiều hơn 1 cạnh trong E nối từ u tới v (Hiển nhiên đơn đồ thị cũng là đa đồ thị).
3. G được gọi là đồ thị **vô hướng** nếu các cạnh trong E là không định hướng, tức là cạnh nối hai đỉnh u, v bất kỳ cũng là cạnh nối hai đỉnh v, u. Hay nói cách khác, tập E gồm các cặp (u, v) không tính thứ tự. $(u, v) \equiv (v, u)$
4. G được gọi là đồ thị **có hướng** nếu các cạnh trong E là có định hướng, có thể có cạnh nối từ đỉnh u tới đỉnh v nhưng chưa chắc đã có cạnh nối từ đỉnh v tới đỉnh u. Hay nói cách khác, tập E gồm các cặp (u, v) có tính thứ tự: $(u, v) \neq (v, u)$. Trong đồ thị có hướng, các cạnh được gọi là các **cung**. Đồ thị vô hướng cũng có thể coi là đồ thị có hướng nếu như ta coi cạnh nối hai đỉnh u, v bất kỳ tương đương với hai cung (u, v) và (v, u).

Ví dụ:



Vô hướng — Có hướng — Đơn đồ thị

Vô hướng — Có hướng — Đa đồ thị

II. CÁC KHÁI NIỆM

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

Cạnh liên thuộc, đỉnh kề, bậc

- Đối với đồ thị vô hướng G = (V, E). Xét một cạnh e ∈ E, nếu e = (u, v) thì ta nói hai đỉnh u và v là **kề nhau** (adjacent) và cạnh e này **liên thuộc** (incident) với đỉnh u và đỉnh v.
- Với một đỉnh v trong đồ thị, ta định nghĩa **bậc** (degree) của v, ký hiệu deg(v) là số cạnh liên thuộc với v. Dễ thấy rằng trên đơn đồ thị thì số cạnh liên thuộc với v cũng là số đỉnh kề với v.

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

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

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

Hệ quả: Trong đồ thị vô hướng, số đỉnh bậc lẻ là số chẵn

- Đối với đồ thị có hướng G = (V, E). Xét một cung e ∈ E, nếu e = (u, v) thì ta nói **u nối tới v** và **v nối từ u,** cung e là đi **ra khỏi đỉnh u và đi vào đỉnh v**. Đỉnh u khi đó được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của cung e.
- Với mỗi đỉnh v trong đồ thị có hướng, ta định nghĩa: **Bán bậc ra** của v ký hiệu $\deg^+(v)$ là số cung đi ra khỏi nó; **bán bậc vào** ký hiệu $\deg^-(v)$ là số cung đi vào đỉnh đó

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

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

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

Một số tính chất của đồ thị có hướng không phụ thuộc vào hướng của các cung. Do đó để tiện trình bày, trong một số trường hợp ta có thể không quan tâm đến hướng của các cung và coi các cung đó là các cạnh của đồ thị vô hướng. Và đồ thị vô hướng đó được gọi là **đồ thị vô hướng nền** của đồ thị có hướng ban đầu.

§2. BIỂU DIỄN ĐỒ THỊ TRÊN MÁY TÍNH

I. MA TRẬN LIỀN KỀ (MA TRẬN KỀ)

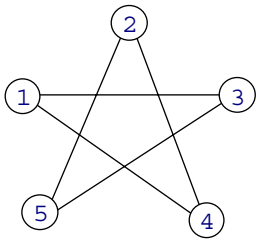
Giả sử G = (V, E) là một **đơn đồ thị** có số đỉnh (ký hiệu $|V|$) là n, Không mất tính tổng quát có thể coi các đỉnh được đánh số 1, 2, ..., n. Khi đó ta có thể biểu diễn đồ thị bằng một ma trận vuông $A = [a_{ij}]$ cấp n. Trong đó:

- $a_{ij} = 1$ nếu $(i, j) \in E$
- $a_{ij} = 0$ nếu $(i, j) \notin E$
- Quy ước $a_{ii} = 0$ với $\forall i$;

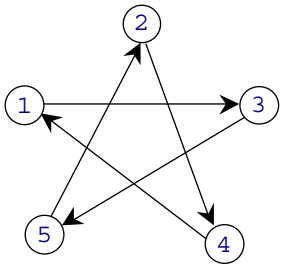
Đối với đa đồ thị thì việc biểu diễn cũng tương tự trên, chỉ có điều nếu như (i, j) là cạnh thì không phải ta ghi số 1 vào vị trí a_{ij} mà là ghi số cạnh nối giữa đỉnh i và đỉnh j

Ví dụ:

	1	2	3	4	5
1	0	0	**1**	**1**	0
2	0	0	0	**1**	**1**
3	**1**	0	0	0	**1**
4	**1**	**1**	0	0	0
5	0	**1**	**1**	0	0



	1	2	3	4	5
1	0	0	**1**	0	0
2	0	0	0	**1**	0
3	0	0	0	0	**1**
4	**1**	0	0	0	0
5	0	**1**	0	0	0



Các tính chất của ma trận liền kề:

1. Đối với đồ thị vô hướng G, thì ma trận liền kề tương ứng là ma trận đối xứng ($a_{ij} = a_{ji}$), điều này không đúng với đồ thị có hướng.
2. Nếu G là đồ thị vô hướng và A là ma trận liền kề tương ứng thì trên ma trận A:
   Tổng các số trên hàng i = Tổng các số trên cột i = Bậc của đỉnh i = deg(i)
3. Nếu G là đồ thị có hướng và A là ma trận liền kề tương ứng thì trên ma trận A:
   - Tổng các số trên hàng i = Bán bậc ra của đỉnh i = $deg^+(i)$
   - Tổng các số trên cột i = Bán bậc vào của đỉnh i = $deg^-(i)$
4. Trong trường hợp G là đơn đồ thị, ta có thể biểu diễn ma trận liền kề A tương ứng là các phần tử logic. a_{ij} = TRUE nếu $(i, j) \in E$ và a_{ij} = FALSE nếu $(i, j) \notin E$

Ưu điểm của ma trận liền kề:

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

Nhược điểm của ma trận liền kề:

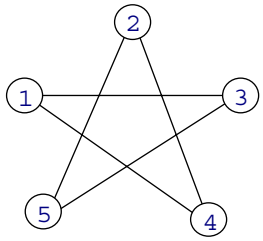
- Bất kể số cạnh của đồ thị là nhiều hay ít, ma trận liền kề luôn luôn đòi hỏi n^2 ô nhớ để lưu các phần tử ma trận, điều đó gây lãng phí bộ nhớ dẫn tới việc không thể biểu diễn được đồ thị với số đỉnh lớn.
- Với một đỉnh u bất kỳ của đồ thị, nhiều khi ta phải xét tất cả các đỉnh v khác kề với nó, hoặc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

II. DANH SÁCH CẠNH

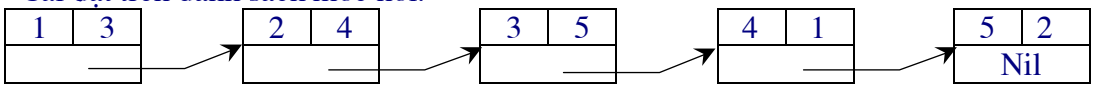
Trong trường hợp đồ thị có n đỉnh, m cạnh, ta có thể biểu diễn đồ thị dưới dạng danh sách cạnh, trong cách biểu diễn này, người ta liệt kê tất cả các cạnh của đồ thị trong một danh sách, mỗi phần tử của danh sách là một cặp (u, v) tương ứng với một cạnh của đồ thị. (Trong trường hợp đồ thị có hướng thì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

Cài đặt trên mảng:

1	1	3
2	2	4
3	3	5
4	4	1
5	5	2



Cài đặt trên danh sách móc nối:



Ưu điểm của danh sách cạnh:

- Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ: chẳng hạn m < 6n), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ, bởi nó chỉ cần 2m ô nhớ để lưu danh sách cạnh.
- Trong một số trường hợp, ta phải xét tất cả các cạnh của đồ thị thì cài đặt trên danh sách cạnh làm cho việc duyệt các cạnh dễ dàng hơn. (Thuật toán Kruskal chẳng hạn)

Nhược điểm của danh sách cạnh:

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

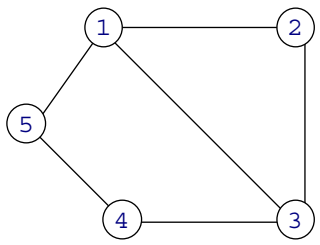
III. DANH SÁCH KỀ

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

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

Cách 1: (Forward Star) Dùng một mảng các đỉnh, mảng đó chia làm n đoạn, đoạn thứ i trong mảng lưu danh sách các đỉnh kề với đỉnh i: Ví dụ với đồ thị sau, danh sách kề sẽ là một mảng A gồm 12 phần tử:

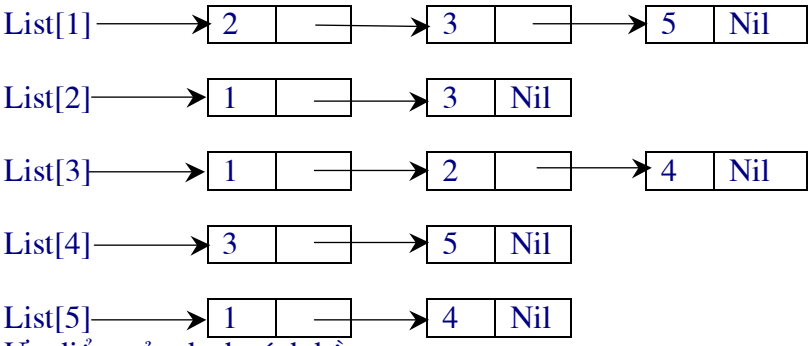
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
2	**3**	**5**	**1**	**3**	**1**	**2**	**4**	**3**	**5**	**1**	**4**
Đoạn 1			Đoạn 2		Đoạn 3			Đoạn 4		Đoạn 5	



Để biết một đoạn nằm từ chỉ số nào đến chỉ số nào, ta có một mảng lưu vị trí riêng. Để tiện trình bày ta gọi mảng lưu vị trí đó là mảng VT. VT[i] sẽ bằng chỉ số đầu đoạn thứ i. Quy ước VT[n + 1] sẽ bằng k + 1 với k là số phần tử của mảng A. Với đồ thị bên thì mảng VT sẽ là: (1, 4, 6, 9, 11, 13)

Như vậy đoạn từ vị trí VT[i] đến VT[i + 1] -1 trong mảng A sẽ chứa các đỉnh kề với đỉnh i. Lưu ý rằng với đồ thị có hướng gồm m cung thì cấu trúc Forward Star cần phải đủ chứa m phần tử, với đồ thị vô hướng m cạnh thì cấu trúc Forward Star cần phải đủ chứa 2m phần tử

Cách 2: Dùng các danh sách móc nối: Với mỗi đỉnh i của đồ thị, ta cho tương ứng với nó một danh sách móc nối các đỉnh kề với i, có nghĩa là tương ứng với một đỉnh i, ta phải lưu lại List[i] là chốt của một danh sách móc nối. Ví dụ với đồ thị trên, danh sách móc nối sẽ là:



Ưu điểm của danh sách kề:

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

Nhược điểm của danh sách kề

- Về lý thuyết, so với hai phương pháp biểu diễn trên, danh sách kề tốt hơn hẳn. Chỉ có điều, trong trường hợp cụ thể mà ma trận kề hay danh sách cạnh **không thể hiện nhược điểm** thì ta nên dùng ma trận kề (hay danh sách cạnh) bởi cài đặt danh sách kề có phần dài dòng hơn.

IV. NHẬN XÉT

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

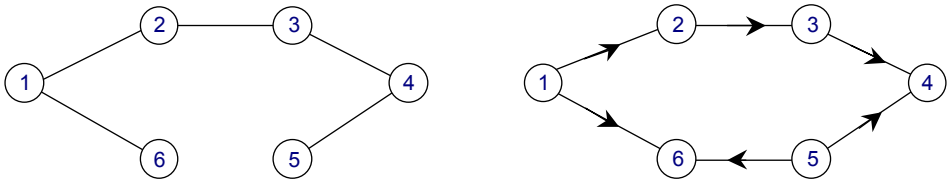
```
program Nhap_Do_Thi;
var
 A: array[1..100, 1..100] of Byte; {Ma trận kề của đồ thị}
 n, i, j: Byte;
begin
 Write('Cho số đỉnh của đồ thị: '); Readln(n);
 FillChar(A, SizeOf(A), 0);
 repeat
  Write('Nhập cạnh (i, j)- (i = 0 để thoát): ');
  Readln(i, j); {Nhập một cặp (i, j) tưởng như là nhập danh sách cạnh}
  if i <> 0 then
   begin {nhưng lưu trữ trong bộ nhớ lại theo kiểu ma trận kề}
    Inc(A[i, j]);
    Inc(A[j, i]);
   end;
 until i = 0; {Nếu người sử dụng nhập giá trị i = 0 thì dừng quá trình nhập, nếu không thì tiếp tục}
end.
```

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

§3. CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

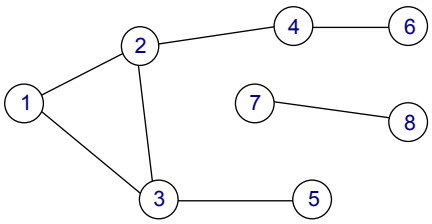
I. BÀI TOÁN

Cho đồ thị G = (V, E). u và v là hai đỉnh của G. Một **đường đi** (path) độ dài l từ đỉnh u đến đỉnh v là dãy $(u = x_0, x_1, ..., x_l = v)$ thoả mãn $(x_i, x_{i+1}) \in E$ với $\forall i: (0 \le i < l)$.
Đường đi nói trên còn có thể biểu diễn bởi dãy các cạnh: $(u = x_0, x_1), (x_1, x_2), ..., (x_{l-1}, x_l = v)$
Đỉnh u được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của đường đi. Đường đi có đỉnh đầu trùng với đỉnh cuối gọi là **chu trình** (Circuit), đường đi không có cạnh nào đi qua hơn 1 lần gọi là **đường đi đơn,** tương tự ta có khái niệm **chu trình đơn.**
Ví dụ: Xét một đồ thị vô hướng và một đồ thị có hướng dưới đây:



Trên cả hai đồ thị, (1, 2, 3, 4) là đường đi đơn độ dài 3 từ đỉnh 1 tới đỉnh 4. Bởi (1, 2) (2, 3) và (3, 4) đều là các cạnh (hay cung). (1, 6, 5, 4) không phải đường đi bởi (6, 5) không phải là cạnh (hay cung).
Một bài toán quan trọng trong lý thuyết đồ thị là bài toán duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó. Vấn đề này đưa về một bài toán liệt kê mà yêu cầu của nó là không được bỏ sót hay lặp lại bất kỳ đỉnh nào. Chính vì vậy mà ta phải xây dựng những thuật toán cho phép **duyệt một cách hệ thống** các đỉnh, những thuật toán như vậy gọi là những thuật toán **tìm kiếm trên đồ thị** và ở đây ta quan tâm đến hai thuật toán cơ bản nhất: **thuật toán tìm kiếm theo chiều sâu** và **thuật toán tìm kiếm theo chiều rộng** cùng với một số ứng dụng của nó. Lưu ý: Thứ nhất: những cài đặt dưới đây là cho đơn đồ thị vô hướng, muốn làm với đồ thị có hướng hay đa đồ thị cũng không phải sửa đổi gì nhiều. Thứ hai: để tiết kiệm thời gian nhập liệu, chương trình quy định dữ liệu về đồ thị sẽ được nhập từ file văn bản GRAPH.INP. Trong đó:

- Dòng 1 ghi số đỉnh n và số cạnh m của đồ thị cách nhau 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng hai số nguyên dương u, v cách nhau một dấu cách, thể hiện có cạnh nối đỉnh u và đỉnh v trong đồ thị.



Đồ thị G và file GRAPH.INP tương ứng

```
GRAPH.INP
8 7
1 2
1 3
2 3
2 4
4 6
3 5
7 8
```

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

1. Cài đặt đệ quy

Vào: Đơn đồ thị vô hướng G = (V, E) gồm n đỉnh, m cạnh. Các đỉnh được đánh số từ 1 đến n. Đỉnh xuất phát S, đỉnh đích F.
Ra:
a) Tất cả các đỉnh có thể đến được từ S
b) Một đường đi đơn (nếu có) từ S đến F
Tư tưởng của thuật toán có thể trình bày như sau: Trước hết, mọi đỉnh x kề với S tất nhiên sẽ đến được từ S. Với mỗi đỉnh x kề với S đó thì tất nhiên những đỉnh y kề với x cũng đến được từ S... Điều đó gợi ý cho ta viết một thủ tục đệ quy DFS(u) mô tả việc duyệt từ đỉnh u bằng cách thông báo tới được u và tiếp tục quá trình duyệt DFS(v) với v là một đỉnh kề với u.

- Để không một đỉnh nào bị liệt kê tới hai lần, ta sử dụng kỹ thuật đánh dấu, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại để các bước duyệt đệ quy kế tiếp không duyệt lại đỉnh đó nữa
- Để lưu lại đường đi từ đỉnh xuất phát S, trong thủ tục DFS(u), trước khi gọi đệ quy DFS(v) với v là một đỉnh kề với u mà chưa đánh dấu, ta lưu lại vết đường đi từ u tới v bằng cách đặt TRACE[v] := u, tức là TRACE[v] lưu lại đỉnh liền trước v trong đường đi từ S tới v. Khi quá trình tìm kiếm theo chiều sâu kết thúc, đường đi từ S tới F sẽ là:

$$F \leftarrow p1 = Trace[F] \leftarrow p2 = Trace[p1] \leftarrow ... \leftarrow S.$$

```
procedure DFS(u);
begin
 < 1. Thông báo tới được u >
 < 2. Đánh dấu u >
 < 3. Xét mọi đỉnh v kề với u mà chưa bị đánh dấu, với mỗi đỉnh v đó >
  begin
   Trace[v] := u;        {Lưu vết đường đi, đỉnh mà từ đó tới v là u}
   DFS(v);                   {Gọi đệ quy duyệt tương tự đối với v}
  end;
end;

begin {Chương trình chính}
 < Nhập dữ liệu: đồ thị, đỉnh xuất phát S, đỉnh đích F >
 < Khởi tạo: Tất cả các đỉnh đều chưa bị đánh dấu >
 DFS(S);
 < Nếu F chưa bị đánh dấu thì không thể có đường đi từ S tới F >
 < Nếu F đã bị đánh dấu thì truy theo vết để tìm đường đi từ S tới F >
end.
```

Chương trình cài đặt thuật toán tìm kiếm theo chiều sâu dưới đây biểu diễn đơn đồ thị vô hướng bởi ma trận kề A, muốn làm trên đồ thị có hướng hoặc đa đồ thị cũng không phải sửa đổi gì nhiều. Lưu ý rằng đường đi từ S tới F sẽ được in ngược từ F về S theo quá trình truy vết.

```
program Depth_First_Search_1; {Tìm kiếm theo chiều sâu}
const
 max = 100;
var
 A: array[1..max, 1..max] of Boolean; {Ma trận kề của đồ thị}
 Free: array[1..max] of Boolean; {Mảng đánh dấu Free[i] = True nếu đỉnh i chưa được thăm}
 Trace: array[1..max] of Byte; {Trace[i] = đỉnh liền trước i trong đường đi S → i}
 n, S, F: Byte;

procedure Enter;{Nhập dữ liệu: số đỉnh và ma trận kề của đồ thị từ file GRAPH.INP, đỉnh xuất phát và đích}
var
 DataFile: Text;
 i, u, v: Byte;
 m: Word;
begin
 FillChar(A, SizeOf(A), False);   {Khởi tạo ma trận kề toàn False: đồ thị chưa có cạnh nào}
 Assign(DataFile, 'GRAPH.INP'); Reset(DataFile);
 Readln(DataFile, n, m);          {Đọc dòng đầu tiên ra số đỉnh n và số cạnh m}
 for i := 1 to m do
  begin
   Readln(DataFile, u, v);        {Đọc dòng thứ i trong số m dòng tiếp theo ra hai đỉnh u, v}
   A[u, v] := True;               {Đặt phần tử tương ứng trong ma trận kề := True}
   A[v, u] := True;               {Đồ thị vô hướng nên cạnh (u, v) cũng là cạnh (v, u) }
  end;
 Close(DataFile);
 Write('Start, Finish: '); Readln(S, F);
end;

procedure DFS(u: Byte); {Tìm kiếm theo chiều sâu bắt đầu từ đỉnh u}
var
```

```
 v: Byte;
begin
 Write(u, ', '); {Thông báo thăm u}
 Free[u] := False; {Đánh dấu đỉnh u là đã thăm}
 for v := 1 to n do {Xét mọi đỉnh của đồ thị}
  if Free[v] and A[u, v] then {Lọc ra những đỉnh v chưa được thăm mà kề với u}
   begin
    Trace[v] := u; {Ghi vết đường đi, đỉnh liền trước v trong đường đi S → v là u}
    DFS(v); {Gọi đệ quy, tìm kiếm theo chiều sâu bắt đầu từ đỉnh v}
   end;
end;

procedure Result; {In đường đi từ S → F}
begin
 if Free[F] then {Nếu F chưa được thăm thì không có đường S→F}
  Writeln('Not found any path from ', S, ' to ', F)
 else {Nếu không}
  begin
   while F <> S do
    begin
     Write(F, '<--'); {In ra F}
     F := Trace[F]; {Trace[F] là đỉnh liền trước F trong đường đi S→F nên sẽ truy tiếp Trace[F]}
    end;
   Writeln(S);
  end;
end;

begin
 Enter;
 FillChar(Free, n, True);
 DFS(S);
 Writeln;
 Result;
end.
```

Chú ý:

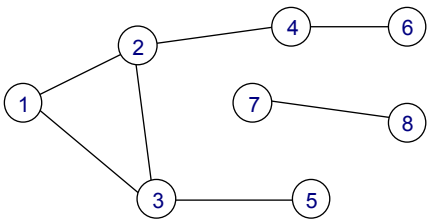
a) Vì có kỹ thuật đánh dấu, nên thủ tục DFS sẽ được gọi ≤ n lần (n là số đỉnh), không có hiện tượng bùng nổ tổ hợp ở đây.

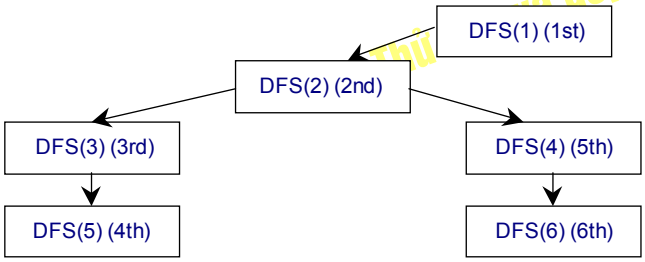
b) Đường đi từ S tới F có thể có nhiều, ở trên chỉ là một trong số các đường đi. Cụ thể là đường đi có thứ tự từ điển nhỏ nhất.

c) Có thể chẳng cần dùng mảng đánh dấu Free, ta khởi tạo mảng lưu vết Trace ban đầu toàn 0, mỗi lần từ đỉnh u thăm đỉnh v, ta có thao tác gán vết Trace[v] := u, khi đó Trace[v] sẽ khác 0. Vậy việc kiểm tra một đỉnh v là chưa được thăm ta có thể kiểm tra Trace[v] = 0. Chú ý: ban đầu khởi tạo Trace[S] := n + 1 (Chỉ là để cho khác 0 thôi).

```
procedure DFS(u: Byte); {Cải tiến}
var
 v: Byte;
begin
 Write(u, ', ');
 for v := 1 to n do
  if (Trace[v] = 0) and A[u, v] then {Trace[v] = 0 thay vì Free[v] = True}
   begin
    Trace[v] := u; {Lưu vết cũng là đánh dấu luôn}
    DFS(v);
   end;
end;
```

Ví dụ: Với đồ thị sau đây, đỉnh xuất phát S = 1: quá trình duyệt đệ quy có thể vẽ trên cây sau:





Hỏi: *Đỉnh 2 và 3 đều kề với đỉnh 1, nhưng tại sao DFS(1) chỉ gọi đệ quy tới DFS(2) mà không gọi DFS(3) ?.*

Trả lời: *Đúng là cả 2 và 3 đều kề với 1, nhưng DFS(1) sẽ tìm thấy 2 trước và gọi DFS(2). Trong DFS(2) sẽ xét tất cả các đỉnh kề với 2 mà chưa đánh dấu thì dĩ nhiên trước hết nó tìm thấy 3 và gọi DFS(3), khi đó* ***3 đã bị đánh dấu*** *nên khi kết thúc quá trình đệ quy gọi DFS(2), lùi về DFS(1) thì đỉnh 3 đã được thăm (đã bị đánh dấu) nên DFS(1) sẽ không gọi DFS(3) nữa.*

Hỏi: *Nếu F = 5 thì đường đi từ 1 tới 5 trong chương trình trên sẽ in ra thế nào ?.*

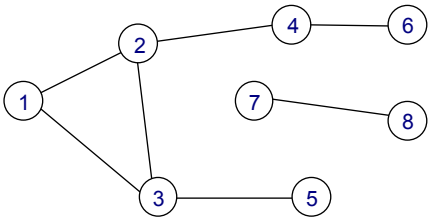
Trả lời: *DFS(5) do DFS(3) gọi nên Trace[5] = 3. DFS(3) do DFS(2) gọi nên Trace[3] = 2. DFS(2) do DFS(1) gọi nên Trace[2] = 1. Vậy đường đi là: 5 ← 3 ← 2 ←1.*

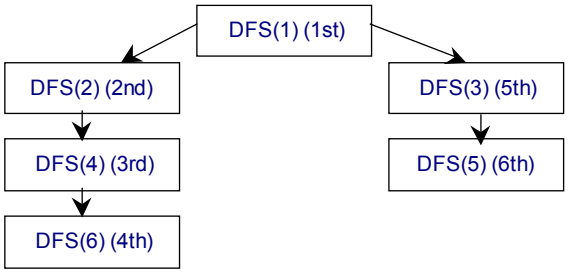
Hỏi: *Dựa vào thứ tự duyệt từ 1st đến 6th, cho biết tại sao người ta lại gọi là Depth First Search?.*

Có cách cài đặt đệ quy khác cũng **tạm** gọi là Depth First Search: thủ tục DFS(u) sẽ xét tất cả những đỉnh v kề với u mà chưa bị đánh dấu, đánh dấu tất cả những đỉnh v đó lại, sau đó mới gọi đệ quy từng DFS(v) sau. Ta chỉ cần sửa tiếp thủ tục DFS như sau:

```
procedure DFS(u: Byte); {Sửa đổi}
var
 v: Byte;
begin
 Write(u, ', ');
 for v := 1 to n do {Trước hết lưu vết (≡đánh dấu) tất cả những đỉnh v kề với u mà chưa được thăm}
  if (Trace[v] = 0) and A[u, v] then Trace[v] := u;
 for v := 1 to n do {Rồi mới gọi DFS(v) với từng đỉnh v đó}
  if Trace[v] = u then DFS(v);
end;
```

Với đồ thị dưới đây (S = 1) thì quá trình duyệt đệ quy có thể vẽ như cây sau (có khác trước)





Hỏi: *Tại sao ở đây DFS(1) lại gọi cả DFS(2) và DFS(3), khác với trên?*

Trả lời: *Bởi trong DFS(1) đã gán cả Trace[2] và Trace[3] := 1 rồi mới gọi DFS(2) nên các thủ tục đệ quy bắt đầu từ DFS(2) sẽ không "quét" phải đỉnh 3 nữa (Nó chỉ quét những đỉnh nào có Trace = 0 thôi)*

Hỏi: *Nếu S = 1 và F = 5 thì đường đi sẽ được in ra thế nào ?*

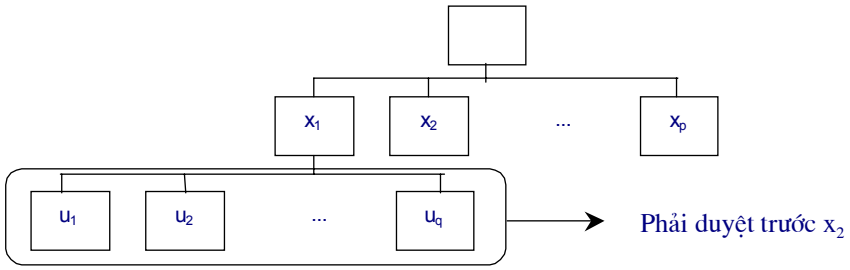
Trả lời: *5 ← 3 ← 1*

Với cây thể hiện quá trình đệ quy DFS ở trên, ta thấy nếu dây chuyền đệ quy là: DFS(S) → DFS (u_1) → DFS(u_2) ... Thì thủ tục DFS nào gọi cuối dây chuyền sẽ được thoát ra đầu tiên, thủ tục DFS(S) gọi đầu dây chuyền sẽ được thoát cuối cùng. Vậy nên chăng, ta có thể mô tả dây chuyền đệ quy bằng một ngăn xếp (Stack)

2. Cài đặt không đệ quy

Cơ sở của phép khử đệ quy là: "lập lịch" duyệt các đỉnh. Việc thăm một đỉnh sẽ lên lịch duyệt các đỉnh kề nó sao cho thứ tự duyệt là ưu tiên độ sâu.

Giả sử phải duyệt các đỉnh theo thứ tự $x_1, x_2, ..., x_p$. Việc thăm x_1 sẽ phát sinh yêu cầu duyệt những đỉnh kề nó $u_1, u_2, ..., u_q$. Để đảm bảo quá trình duyệt ưu tiên độ sâu, các đỉnh u này phải được duyệt trước x_2 tức là thứ tự duyệt đỉnh sau khi đã thăm x_1 sẽ phải là $(u_1, u_2, ..., u_q, x_2, ..., x_p)$;



Như vậy, có hai việc chủ yếu: **duyệt đỉnh** và **lên lịch**. Vì đỉnh "lên lịch" sau sẽ được duyệt trước nên ta sẽ mô tả "lịch" bằng một ngăn xếp. Thứ tự thăm đỉnh là thứ tự lấy ra từ ngăn xếp, việc lên lịch tương đương với việc đẩy một đỉnh vào ngăn xếp. Việc đánh dấu đỉnh sẽ theo hai trạng thái: đã lên lịch và chưa lên lịch. Đã lên lịch (đã đánh dấu) tức là đỉnh đó đã được thăm hoặc đang ở trong ngăn xếp (sớm muộn gì cũng được thăm)

Ta sẽ dựng giải thuật như sau:

Bước 1: Khởi tạo:

- Các đỉnh đều ở trạng thái chưa đánh dấu, ngoại trừ đỉnh xuất phát S là đã đánh dấu
- Một ngăn xếp (Stack), ban đầu chỉ có một phần tử là S. Ngăn xếp dùng để chứa các đỉnh sẽ được duyệt theo thứ tự ưu tiên độ sâu

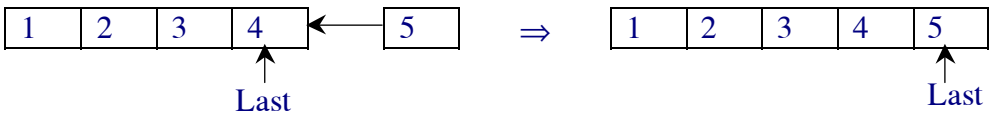
Bước 2: Lặp các bước sau đến khi ngăn xếp rỗng:

- Lấy u khỏi ngăn xếp, thông báo thăm u (Bắt đầu việc duyệt đỉnh u)
- Xét tất cả những đỉnh v kề với u mà chưa được đánh dấu, với mỗi đỉnh v đó:
  1. Đánh dấu v.
  2. Ghi nhận vết đường đi từ u tới v (Có thể làm chung với việc đánh dấu)
  3. Đẩy v vào ngăn xếp (Lên lịch thực hiện duyệt đỉnh v)

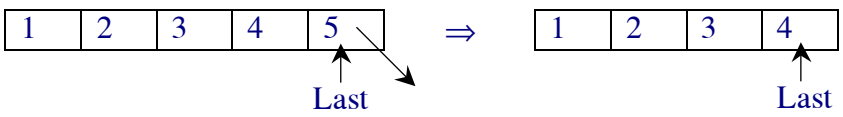
Bước 3: Giống như cài đặt đệ quy: truy vết tìm đường đi.

Việc mô tả ngăn xếp có thể bằng một mảng. Lưu ý rằng số phần tử của ngăn xếp không bao giờ vượt quá n (số đỉnh). Giả sử ta dùng một mảng Stack và một số nguyên Last lưu số phần tử thực sự trong ngăn xếp. Ta có hai thao tác cơ bản trên ngăn xếp:

Đưa một giá trị vào ngăn xếp ⇔ Thêm một phần tử vào cuối mảng Stack:



Lấy một phần tử khỏi ngăn xếp ⇔ Bỏ phần tử cuối của mảng Stack:

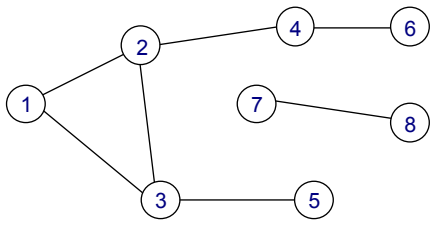



```
program Depth_First_Search_2; {Tìm kiếm theo chiều sâu không đệ quy}
const
 max = 100;
var
```

```
 A: array[1..max, 1..max] of Boolean; {Ma trận kề của đồ thị}
 Free: array[1..max] of Boolean; {Free[i] = False nếu i đã thăm hoặc đã lên lịch}
 Trace: array[1..max] of Byte; {Vết đường đi}
 Stack: array[1..max] of Byte; {Ngăn xếp = mảng chứa các phần tử đã lên lịch nhưng chưa thăm}
 n, S, F, Last: Byte; {Last là số phần tử thực sự của mảng Stack = số đỉnh trong ngăn xếp}

(*procedure Enter; Như trên*)

procedure Init; {Khởi tạo}
begin
 FillChar(Free, n, True); {Các đỉnh đều chưa lên lịch- chưa bị đánh dấu}
 Free[S] := False; {Ngoại trừ đỉnh xuất phát là lên lịch duyệt đầu tiên}
 Last := 1; {Ngăn xếp chỉ gồm 1 phần tử}
 Stack[1] := S; {Đó là đỉnh xuất phát}
end;

procedure Push(V: Byte); {Đẩy một đỉnh V vào ngăn xếp}
begin
 Inc(Last);
 Stack[Last] := V;
end;

function Pop: Byte; {Lấy một đỉnh khỏi ngăn xếp, trả về đỉnh lấy ra trong kết quả hàm}
begin
 Pop := Stack[Last];
 Dec(Last);
end;

procedure DFS; {Tìm kiếm theo chiều sâu không đệ quy}
var
 u, v: Byte;
begin
 repeat
   u := Pop; {Bao giờ cũng đi thăm đỉnh u lấy ra từ ngăn xếp}
   Write(u, ', ');
   for v := n downto 1 do
    if Free[v] and A[u, v] then {Sau đó xét những đỉnh v kề u mà chưa đánh dấu}
     begin
      Push(v); {Lên lịch duyệt v}
      Free[v] := False; {Đánh dấu v đã lên lịch, tránh việc một đỉnh lên lịch 2 lần}
      Trace[v] := u; {Ghi vết đường đi}
     end;
 until Last = 0; {Cho tới khi ngăn xếp rỗng - tất cả các đỉnh  trong lịch đều đã thăm}
 Writeln;
end;

(*procedure Result;  In kết quả, giống trên*)

begin
 Enter;
 Init;
 DFS;
 Result;
end.
```

Lưu ý: Việc thêm vào/ lấy ra một phần tử của ngăn xếp đều được thực hiện ở cuối mảng Stack.

Ví dụ: với đồ thị bên (S = 1), Ta thử theo dõi quá trình thực hiện thủ tục tìm kiếm theo chiều sâu dùng ngăn xếp và đối



sánh thứ tự các đỉnh ra khỏi ngăn xếp với thứ tự từ 1st đến 6th trong cây tìm kiếm của thủ tục DFS sửa đổi dùng đệ quy.

Ngăn xếp	**Đỉnh u (lấy ra từ ngăn xếp)**	Ngăn xếp (sau khi lấy u ra)	Các đỉnh v kề u mà chưa lên lịch	Ngăn xếp sau khi đẩy những đỉnh v vào
(1)	**1**	∅	3, 2	(3, 2)
(3, 2)	**2**	(3)	4	(3, 4)
(3, 4)	**4**	(3)	6	(3, 6)
(3, 6)	**6**	(3)	Không có	(3)
(3)	**3**	∅	5	(5)
(5)	**5**	∅	Không có	∅
∅				

(NX: Đúng theo thứ tự duyệt đỉnh của thủ tục sửa đổi mà ta tạm thời cũng gọi nó là Depth Firsh Search)

Tại sao khi xét các đỉnh v kề với u để lên lịch duyệt v, ta lại xét v theo thứ tự từ chỉ số lớn đến chỉ số nhỏ (for v := n downto 1 do) ?. Bởi vì khi đó thứ tự đẩy các đỉnh v vào ngăn xếp sẽ theo thứ tự từ chỉ số lớn đến chỉ số nhỏ. Tức là khi lấy ra sẽ lấy theo thứ tự từ chỉ số nhỏ đến chỉ số lớn. Ví dụ với đồ thị trên khi đẩy hai đỉnh 2 và 3 (kề với 1) vào ngăn xếp, ta đẩy 3 trước, 2 sau. Để quá trình duyệt sẽ duyệt 2 trước, 3 sau, ta được kết quả "đẹp" hơn. Đẩy 2 trước, 3 sau cũng được, khi đó thứ tự duyệt các đỉnh sẽ là: (1, 3, 5, 2, 4, 6)

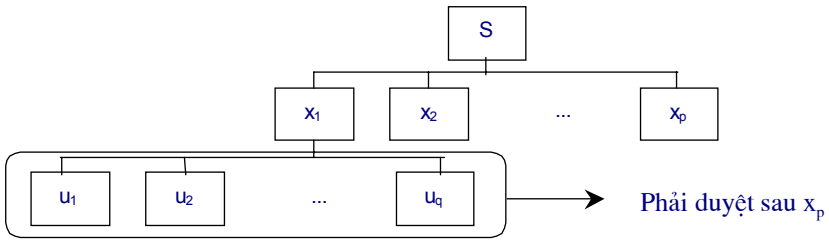
Có thể có câu hỏi: Ngăn xếp (Stack) và hàng đợi (Queue) là hai cấu trúc có quan hệ chặt chẽ, có thể nói chúng là hai cấu trúc đối ngẫu. Vậy điều gì sẽ xảy ra nếu ta thay ngăn xếp bằng hàng đợi ?

III. THUẬT TOÁN TÌM KIẾM THEO CHIỀU RỘNG (BREADTH FIRST SEARCH)

1. Cài đặt bằng hàng đợi

Thuật toán tìm kiếm theo chiều sâu dựa trên ngăn xếp có tính chất: đỉnh nào vào ngăn xếp sau sẽ được duyệt trước. Nếu thay ngăn xếp bằng hàng đợi thì chỉ có một sự thay đổi duy nhất: đỉnh nào vào hàng đợi trước sẽ được duyệt trước. Khi đó ta cũng sẽ duyệt được đầy đủ các đỉnh có thể đến từ S nhưng theo thứ tự ưu tiên chiều rộng, tức là đỉnh nào gần S hơn sẽ được duyệt trước (gần hơn theo nghĩa đường đi từ S đến nó qua ít cạnh hơn)

Bắt đầu ta sẽ thăm đỉnh S. Việc thăm đỉnh S sẽ phát sinh thứ tự duyệt những đỉnh (x_1, x_2, ..., x_p) kề với S (những đỉnh gần S nhất). Khi thăm đỉnh x_1 sẽ lại phát sinh yêu cầu duyệt những đỉnh (u_1, u_2 ..., u_q) kề với x_1. Nhưng rõ ràng các đỉnh u này "xa" S hơn những đỉnh x nên chúng chỉ được duyệt khi tất cả những đỉnh x đã duyệt xong. Tức là thứ tự duyệt đỉnh sau khi đã thăm x_1 sẽ là: (x_2, x_3..., x_p, u_1, u_2, ..., u_q)



Ta sẽ dựng giải thuật như sau:

Bước 1: Khởi tạo:

- Các đỉnh đều ở trạng thái chưa đánh dấu, ngoại trừ đỉnh xuất phát S là đã đánh dấu
- Một hàng đợi (Queue), ban đầu chỉ có một phần tử là S. Hàng đợi dùng để chứa các đỉnh sẽ được duyệt theo thứ tự ưu tiên chiều rộng

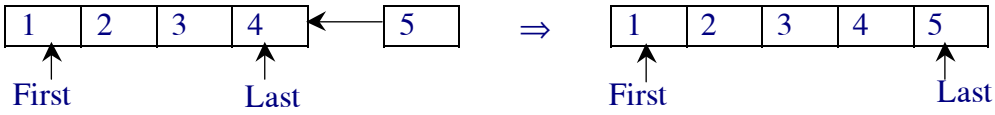
Bước 2: Lặp các bước sau đến khi hàng đợi rỗng:

- Lấy u khỏi hàng đợi, thông báo thăm u (Bắt đầu việc duyệt đỉnh u)
- Xét tất cả những đỉnh v kề với u mà chưa được đánh dấu, với mỗi đỉnh v đó:
  1. Đẩy v vào hàng đợi (Lên lịch thực hiện duyệt đỉnh v)
  2. Đánh dấu v.
  3. Ghi nhận vết đường đi từ u tới v (Có thể làm chung với việc đánh dấu)

Bước 3: Truy vết tìm đường đi.

Việc mô tả hàng đợi có thể bằng một mảng. Tương tự trên, số phần tử của hàng đợi không bao giờ vượt quá n (số đỉnh). Giả sử ta dùng một mảng Queue, một số nguyên Last lưu chỉ số cuối hàng đợi, một số nguyên First lưu chỉ số đầu hàng đợi. Ta có hai thao tác cơ bản trên hàng đợi:

Đưa một giá trị V vào hàng đợi ⇔ Thêm một phần tử vào cuối mảng Queue. Chỉ số đầu hàng đợi giữ nguyên, chỉ số cuối hàng đợi tăng 1:



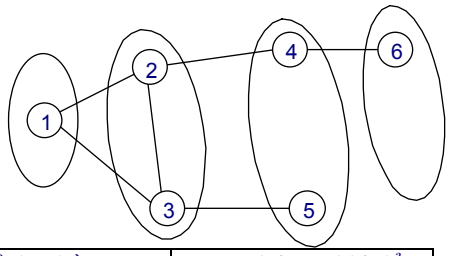
Lấy một phần tử khỏi hàng đợi ⇔ Lấy phần tử thứ First của mảng Queue, phần tử kế tiếp trở thành đầu. Chỉ số đầu hàng đợi tăng 1, chỉ số cuối hàng đợi giữ nguyên:

(Ta có thể dùng số nQueue lưu số phần tử, thao tác lấy phần tử ra luôn lấy ở vị trí 1 rồi "dồn toa" các phần tử từ Queue[2] đến Queue[nQueue] lên trước 1 vị trí)

```
program Breadth_First_Search_1; {Thuật toán tìm kiếm theo chiều rộng dùng hàng đợi}
const
 max = 100;
var
 A: array[1..max, 1..max] of Boolean;
 Free: array[1..max] of Boolean;
 Trace: array[1..max] of Byte;
 Queue: array[1..max] of Byte; {Hàng đợi = mảng chứa các đỉnh đã lên lịch nhưng chưa thăm}
 n, S, F, First, Last: Byte; {First: Chỉ số đầu, Last: Chỉ số cuối hàng đợi}

(*procedure Enter; Như trên *)

procedure Init;
begin
 FillChar(Free, n, True); {Ban đầu các đỉnh đều chưa đánh dấu}
 Free[S] := False; {Ngoại trừ đỉnh S đã bị đánh dấu (lên lịch thăm đầu tiên)}
 Queue[1] := S; {Khởi tạo hàng đợi ban đầu chỉ có mỗi đỉnh S}
 Last := 1;  {Khi đó chỉ số đầu hay chỉ số cuối đều là 1}
 First := 1;
end;

procedure Push(V: Byte); {Đẩy đỉnh V vào hàng đợi}
begin
 Inc(Last);
 Queue[Last] := V;
end;

function Pop: Byte; {Lấy một đỉnh ra khỏi hàng đợi, trả về đỉnh đó trong kết quả hàm}
begin
 Pop := Queue[First];
 Inc(First);
end;

procedure BFS; {Thuật toán tìm kiếm theo chiều rộng}
var
 u, v: Byte;
```

```
begin
 repeat
  u := Pop; {Bao giờ cũng thăm đỉnh u lấy ra từ hàng đợi}
  Write(u, ', ');
  for v := 1 to n do
   if Free[v] and A[u, v] then {Xét các đỉnh v kề với u mà chưa đánh dấu}
    begin
     Push(v); {Lên lịch thăm v}
     Free[v] := False; {Đánh dấu v đã lên lịch, tránh việc một đỉnh lên lịch 2 lần}
     Trace[v] := u; {Lưu vết đường đi}
    end;
 until First > Last; {Cho tới khi hàng đợi rỗng}
 Writeln;
end;

(*procedure Result; Như trên*)

begin
 Enter;
 Init;
 BFS;
 Result;
end.
```

Ví dụ: Xét đồ thị bên, Đỉnh xuất phát S = 1.
Ta thử áp dụng thuật toán xem quá trình các đỉnh vào, ra hàng đợi như thế nào. Lưu ý rằng mảng Queue ở đây có tính chất: Vào ở cuối, ra ở đầu và trước khi áp dụng thuật toán, hàng đợi được khởi tạo chỉ gồm mỗi đỉnh xuất phát.

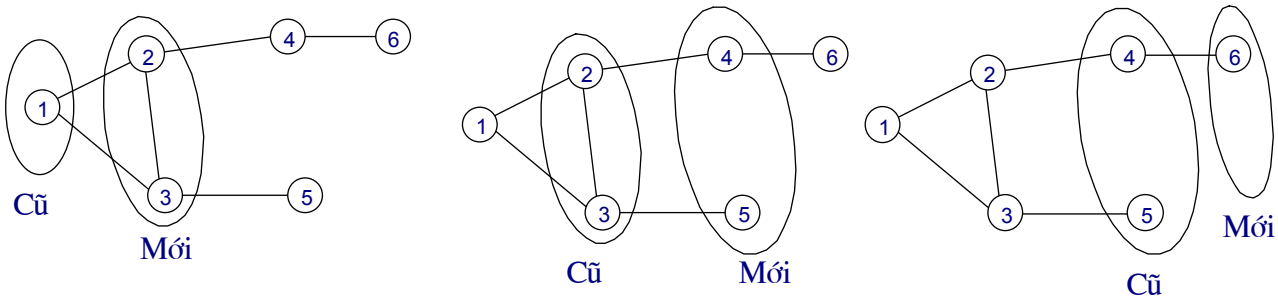


Hàng đợi	**Đỉnh u (lấy ra từ hàng đợi)**	Hàng đợi (sau khi lấy u ra)	Các đỉnh v kề u mà chưa lên lịch	Hàng đợi sau khi đẩy những đỉnh v vào
(1)	**1**	∅	2, 3	(2, 3)
(2, 3)	**2**	(3)	4	(3, 4)
(3, 4)	**3**	(4)	5	(4, 5)
(4, 5)	**4**	(5)	6	(5, 6)
(5, 6)	**5**	(6)	Không có	(6)
(6)	**6**	∅	Không có	∅

Để ý thứ tự các phần tử lấy ra khỏi hàng đợi, ta thấy trước hết là 1; sau đó đến 2, 3; rồi mới tới 4, 5; cuối cùng là 6. Rõ ràng là đỉnh gần S hơn sẽ được duyệt trước. Và như vậy, ta có nhận xét: nếu kết hợp lưu vết tìm đường đi thì **đường đi từ S tới F sẽ là đường đi ngắn nhất** (theo nghĩa qua ít cạnh nhất)

2. Cài đặt bằng thuật toán loang

Cách cài đặt này sử dụng hai tập hợp, một tập "cũ" chứa những đỉnh "đang xét", một tập "mới" chứa những đỉnh "sẽ xét". Ban đầu tập "cũ" chỉ gồm mỗi đỉnh xuất phát, tại mỗi bước ta sẽ dùng tập "cũ" tính tập "mới", tập "mới" sẽ gồm những đỉnh chưa được thăm mà kề với một đỉnh nào đó của tập "cũ". Lặp lại công việc trên (sau khi đã gán tập "cũ" bằng tập "mới") cho tới khi tập cũ là rỗng:



Giải thuật loang có thể dựng như sau:

Bước 1: Khởi tạo
Các đỉnh khác S đều chưa bị đánh dấu, đỉnh S bị đánh dấu, tập "cũ" Old := {S}
Bước 2: Lặp các bước sau đến khi Old = ∅

- Đặt tập "mới" New = ∅, sau đó dùng tập "cũ" tính tập "mới" như sau:
- Xét các đỉnh u ∈ Old, với mỗi đỉnh u đó:
  - Thông báo thăm u
  - Xét tất cả những đỉnh v kề với u mà chưa bị đánh dấu, với mỗi đỉnh v đó:
    - Đánh dấu v
    - Lưu vết đường đi, đỉnh liền trước v trong đường đi S→v là u
    - Đưa v vào tập New
- Gán tập "cũ" Old := tập "mới" New

Bước 3: Truy vết tìm đường đi.

```
program Breadth_First_Search_2;
const
 max = 100;
var
 A: array[1..max, 1..max] of Boolean;
 Free: array[1..max] of Boolean;
 Trace: array[1..max] of Byte;
 Old, New: set of Byte;
 n, S, F: Byte;

(*procedure Enter; Như trên*)

procedure Init;
begin
 FillChar(Free, n, True);
 Free[S] := False; {Các đỉnh đều chưa đánh dấu, ngoại trừ đỉnh S đã đánh dấu}
 Old := [S]; {Tập "cũ" khởi tạo ban đầu chỉ có mỗi S}
end;

procedure BFS; {Thuật toán loang}
var
 u, v: Byte;
begin
 repeat {Lặp: dùng Old tính New}
  New := [];
  for u := 1 to n do
   if u in Old then {Xét những đỉnh u trong tập "cũ", với mỗi đỉnh u đó:}
    begin
     Write(u, ', '); {Thông báo thăm u}
     for v := 1 to n do
      if Free[v] and A[u, v] then {Quét tất cả những đỉnh v chưa bị đánh dấu mà kề với u}
       begin
        Free[v] := False; {Đánh dấu v và lưu vết đường đi}
        Trace[v] := u;
        New := New + [v]; {Đưa v vào tập New}
       end;
    end;
  Old := New; {Gán tập cũ := tập mới và lặp lại}
 until Old = []; {Cho tới khi không loang được nữa}
 Writeln;
end;
```

```
(*procedure Result; chẳng khác gì trước*)

begin
 Enter;
 Init;
 BFS;
 Result;
end.
```


IV. CHÚ Ý QUAN TRỌNG

Quay lại mục II.2. với kỹ thuật khử đệ quy, ta nhận thấy, thứ tự duyệt các đỉnh **không trùng** với thuật toán tìm kiếm theo chiều sâu dùng đệ quy. Tại sao như vậy ?

Như trên đã nói, kỹ thuật khử đệ quy đó là khử đệ quy của một thủ tục khác mà ta tạm gọi nó cũng là Depth First Search để sau này dễ dàng chuyển thành thuật toán tìm kiếm theo chiều rộng bằng cách thay cơ chế vào / ra ngăn xếp bởi cơ chế vào / ra hàng đợi.

Nhưng thuật toán tìm kiếm theo chiều sâu dùng đệ quy mà ta cài đặt đầu tiên có những điểm tốt riêng và có nhiều ứng dụng trong các bài toán, thuật toán khác, trong đó có một số thuật toán mà chỉ cần thay đổi thứ tự duyệt đỉnh của DFS là sẽ gây ra kết quả sai lầm. Vậy nên chăng, ta cố gắng tìm cách khử đệ quy mà không làm thay đổi thứ tự duyệt đỉnh của thuật toán tìm kiếm theo chiều sâu.

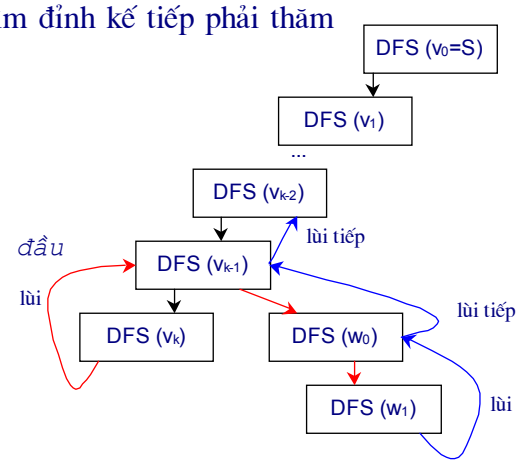
Mô hình Depth First Search dùng đệ quy:

```
procedure DFS(u);
begin
 < 1. Thông báo thăm u >
 < 2. Đánh dấu u đã thăm>
 < 3. Xét mọi đỉnh v kề với u mà chưa thăm, với mỗi đỉnh v đó: >
   DFS(v); {Gọi đệ quy duyệt tương tự đối với v, trước đó có việc lưu vết}
end;
begin {Chương trình chính}
 < Nhập dữ liệu: đồ thị, đỉnh xuất phát S, đỉnh đích F >
 < Khởi tạo: Tất cả các đỉnh đều chưa thăm>
 DFS(S);
 < Nếu F chưa thăm thì không thể có đường đi từ S tới F >
 < Nếu F đã thăm thì truy theo vết để tìm đường đi từ S tới F >
end.
```

Nhận xét:

Quá trình tìm kiếm theo chiều sâu dùng đệ quy trước hết đi thăm đỉnh xuất phát v_0 = S. Sau đó đi thăm đỉnh v_1 là đỉnh chưa thăm đầu tiên trong danh sách kề với v_0, rồi lại đi thăm đỉnh v_2 là đỉnh chưa thăm đầu tiên trong danh sách kề với v_1... dây chuyền đệ quy cứ tiến sâu như vậy cho tới khi thăm đến đỉnh v_k mà không có đỉnh chưa thăm nào kề nó. Đến đây xảy ra sự lùi lại của dây chuyền đệ quy (được thể hiện rất tự nhiên qua sự thoát khỏi thủ tục DFS(v_k) trở về thủ tục gọi nó: DFS(v_{k-1})); khi quay trở lại xét đỉnh v_{k-1}, nếu còn đỉnh kề nó chưa thăm, thì ta chọn w_0 là đỉnh đầu tiên chưa thăm trong danh sách kề v_{k-1} để thăm tiếp, quá trình lặp lại tương tự với sự tiến sâu hơn của dây chuyền đệ quy theo một hướng khác; còn nếu như mọi đỉnh kề với v_{k-1} đều đã thăm thì lại lùi tiếp xét đỉnh v_{k-2} v.v... Thuật toán sẽ kết thúc khi dây chuyền đệ quy lùi về xét đến tận đỉnh v_0=S, mà mọi đỉnh kề S đều đã thăm. Ta sẽ viết một thủ tục FindNext, tìm đỉnh kế tiếp phải thăm sau đỉnh u như sau:

```
function FindNext(u∈V): ∈V;
begin
 repeat
  for ∀v ∈ V
   if (v ∈ Kề(u)) and (v chưa thăm) then
   {Nếu u có đỉnh kề chưa thăm thì chọn đỉnh kề đầu
tiên chưa thăm để thăm tiếp}
    begin
     Trace[ v]  := u; {Lưu vết}
     FindNext := v;
     Exit;
```



```
    end;
  u := Trace[u]; {Nếu không, lùi về một bước}
 until <Lùi về tới tận u = S mà mọi đỉnh kề S đều đã thăm>;
 <ở trên không Exit được tức là mọi đỉnh tới được từ S đã duyệt xong>
end;
program Depth_First_Search_3;
const
 max = 100;
var
 A: array[1..max, 1..max] of Boolean;
 {Trace[v] = đỉnh liền trước v trong đường đi từ S tới v, nếu đệ quy thì tức là Trace[v] = u nếu DFS(u) gọi DFS(v)}
 Trace: array[1..max] of Byte;
 n, S, F: Byte;

procedure Enter;
var
 DataFile: Text;
 i, u, v: Byte;
 m: Word;
begin
 FillChar(A, SizeOf(A), False);
 Assign(DataFile, 'GRAPH.INP'); Reset(DataFile);
 Readln(DataFile, n, m);
 for i := 1 to m do
  begin
   Readln(DataFile, u, v);
   A[u, v] := True;
   A[v, u] := True;
  end;
 Close(DataFile);
 Write('Start, Finish: '); Readln(S, F);
 FillChar(Trace, SizeOf(Trace), 0); {Trace = 0 có nghĩa là đỉnh chưa thăm}
 Trace[S] := n + 1;  {Riêng đỉnh S đã thăm, đặt một giá trị đặc biệt = n + 1}
end;

function FindNext(u: Byte): Byte; {Sau đỉnh u thì thăm đỉnh nào ?}
var
 v: Byte;
begin
 repeat
  for v := 1 to n do
   if A[u, v] and (Trace[v] = 0) then {Nếu u có đỉnh kề chưa thăm}
    begin
     Trace[v] := u; {Lưu vết ≡ đánh dấu ≠ 0 luôn}
     FindNext := v;
     Exit; {Chọn lấy đỉnh kề đầu tiên chưa thăm và thoát luôn}
    end;
  u := Trace[u]; {Nếu mọi đỉnh kề u đã thăm thì lùi một bước}
 until u = n + 1; {Nếu đã về đến tận S mà còn lùi nữa thì sẽ được đỉnh n+1}
 FindNext := 0; {Tức là mọi đỉnh kề S đều đã thăm, hết}
end;

procedure DFS;
var
 u: Byte;
begin
 u := S;
 repeat {Cứ FindNext liên tiếp bắt đầu từ u}
  Write(u, ', ');
  u := FindNext(u);
 until u = 0; {Đến khi duyệt hết các đỉnh có thể thăm được}
end;
```

```
procedure Result;
begin
 if Trace[F] = 0 then
  Writeln('Not found any path from ', S, ' to ', F)
 else
  begin
   while F <> S do
    begin
     Write(F, '<--'); {In ra F}
     F := Trace[F];
    end;
   Writeln(S);
  end;
end;

begin
 Enter;
 DFS;
 Writeln;
 Result;
end.
```

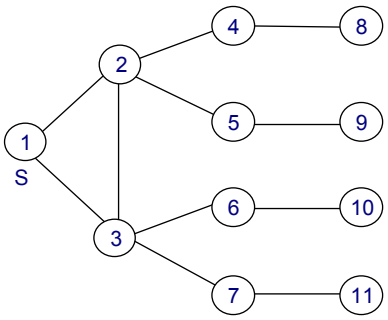
Đây là một cách dựa vào đặc thù của thuật toán tìm kiếm theo chiều sâu để khử đệ quy bằng hàm FindNext. Ta hoàn toàn có thể cải tiến phương pháp dùng ngăn xếp để khử đệ quy sao cho vẫn giữ nguyên thứ tự duyệt đỉnh như sau: Khi lấy một đỉnh u ra khỏi ngăn xếp, ta sẽ kiểm tra u đã thăm hay chưa, nếu chưa thì ta thông báo chính thức thăm u rồi mới đánh dấu u. (Khác với trước: khi lên lịch duyệt u là đã đánh dấu rồi). Nhược điểm của phương pháp này là có thể ngăn xếp phải chứa nhiều hơn n đỉnh (do có những đỉnh được đẩy vào ngăn xếp 2 lần).

Khử đệ quy là một kỹ thuật quan trọng cần nắm vững, nó giúp ích cho ta khi cài đặt trên những ngôn ngữ không cho phép đệ quy. Ngay cả với PASCAL là ngôn ngữ cho phép đệ quy (tự sinh và tương hỗ) nhưng kỹ thuật này cũng rất có ích, bởi nó thay thế **chi phí về bộ nhớ Stack vốn ít ỏi dành cho chương trình con để lưu mã lệnh và biến địa phương** bằng **bộ nhớ toàn cục rộng rãi và linh hoạt hơn.** Dùng ngăn xếp để khử đệ quy là một kỹ thuật được dùng phổ biến trong các thuật toán đệ quy nói chung (khử đệ quy QuickSort khi sắp xếp mảng cỡ 20000 số nguyên chẳng hạn).

Tuy nhiên, phương pháp nào cũng có nhược điểm, khử đệ quy để có thể chạy với dữ liệu lớn thì lại chậm hơn về thời gian. Chương trình đệ quy với những **lệnh máy** cấp phát vùng nhớ Stack và gọi hàm đệ quy thực thi với tốc độ nhanh hơn hẳn so với phương pháp giả lập. Hơn thế nữa, trên các môi trường lập trình 32 bit hiện nay đang rất phổ dụng (Delphi, C++ Builder, Visual C++ v.v...), người ta cũng dần quên đi phương pháp khử đệ quy và khái niệm mảng cấp phát động vì lý do: rào cản 64KB dành cho một đoạn (Segment) bộ nhớ, (tức là giới hạn không thể vượt qua của không gian các biến địa phương và cũng là kích thước tối đa của một biến) giờ đây không còn nữa. Mỗi chương trình có 4GB bộ nhớ về mặt lý thuyết để lưu mã lệnh và dữ liệu của mình.

Bài tập:

Có thể còn thắc mắc "Tại sao làm thế này thì là duyệt theo chiều sâu, thế kia thì lại là duyệt theo chiều rộng". Cách tốt nhất có thể làm là lấy một ví dụ, dò theo các bước của thuật toán, theo dõi các giá trị và cố gắng giải thích "vì sao lại thế ?". Vậy hãy hoàn thành nốt các bảng sau: (với đồ thị bên S = 1)



a) Depth_First_Search_2. Stack ≡ mảng thì vào/ ra Stack đều ở cuối mảng. Các đỉnh v kề với u được liệt kê từ chỉ số lớn tới chỉ số nhỏ

Ngăn xếp	**Đỉnh u (lấy ra từ ngăn xếp)**	Các đỉnh v kề u mà chưa bao giờ bị đẩy vào ngăn xếp	Ngăn xếp (sau khi lấy u ra)	Ngăn xếp sau khi đẩy những đỉnh v đó vào
(1)	**1**	3, 2	∅	(3, 2)
(3, 2)	**2**	5, 4	(3)	(3, 5, 4)

(3, 5, 4)	**4**	8	(3, 5)	(3, 5, 8)
(3, 5, 8)	**8**	không có	(3, 5)	(3, 5)
(3, 5)				
				∅

b) Breadth_First_Search_1. Queue ≡ mảng thì vào Queue ở cuối mảng, ra Queue ở đầu mảng. Các đỉnh v kề với u được liệt kê từ chỉ số nhỏ tới chỉ số lớn.

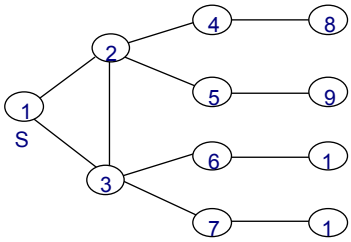
Hàng đợi	**Đỉnh u (lấy ra từ hàng đợi)**	Các đỉnh v kề u mà chưa bao giờ bị đẩy vào hàng đợi	Hàng đợi (sau khi lấy u ra)	Hàng đợi sau khi đẩy những đỉnh v đó vào
(1)	**1**	2, 3	∅	(2, 3)
(2, 3)	**2**	4, 5	(3)	(3, 4, 5)
(3, 4, 5)	**3**	6, 7	(4, 5)	(4, 5, 6, 7)
(4, 5, 6, 7)				
				∅

c) Breadth_First_Search_2:

Tập cũ	Tập mới gồm những đỉnh v chưa xét mà v kề với một đỉnh u nào đó của tập cũ	Tập cũ sau khi:= tập mới
{1}	{2, 3}	{2, 3}
{2, 3}		
		∅

d) Depth_First_Search_3

u	Tập những đỉnh chưa thăm	FindNext(u)
1 = S	{2..11}	2
2	{3..11}	3
3	{4..11}	6
6	{4, 5, 7, 8, 9, 10, 11}	10
10	{4, 5, 7, 8, 9, 11} Lùi từ 10→6→3 duyệt các đỉnh kề 3 chưa thăm	7
7	{4, 5, 8, 9, 11}	11
11	{4, 5, 8, 9} Lùi về tận 2...	
		12 = n + 1

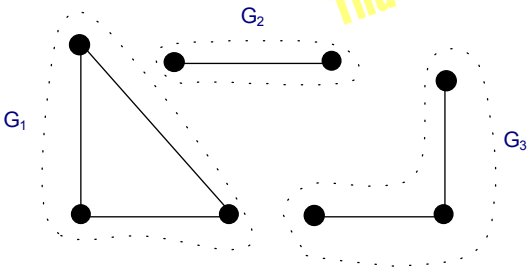


§4. TÍNH LIÊN THÔNG CỦA ĐỒ THỊ

I. ĐỊNH NGHĨA

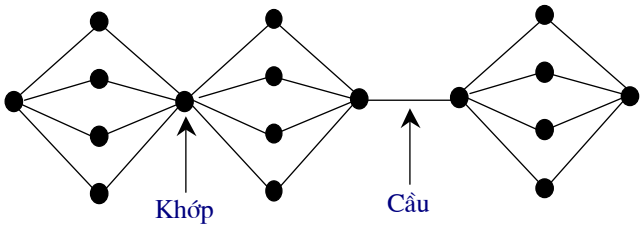
1. Đối với đồ thị vô hướng G = (V, E)

G gọi là **liên thông** (connected) nếu luôn tồn tại đường đi giữa mọi cặp đỉnh phân biệt của đồ thị. Nếu G không liên thông thì chắc chắn nó sẽ là hợp của hai hay nhiều đồ thị con[1] liên thông, các đồ thị con này đôi một không có đỉnh chung. Các đồ thị con liên thông rời nhau như vậy được gọi là các thành phần liên thông của đồ thị đang xét (Xem ví dụ bên).



Đồ thị G và các thành phần liên thông G_1, G_2, G_3 của nó

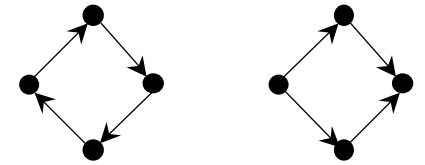
Đôi khi, việc xoá đi một đỉnh và tất cả các cạnh liên thuộc với nó sẽ tạo ra một đồ thị con mới có nhiều thành phần liên thông hơn đồ thị ban đầu, các đỉnh như thế gọi là **đỉnh cắt** hay **điểm khớp**. Hoàn toàn tương tự, những cạnh mà khi ta bỏ nó đi sẽ tạo ra một đồ thị có nhiều thành phần liên thông hơn so với đồ thị ban đầu được gọi là một **cạnh cắt** hay một **cầu.**



2. Đối với đồ thị có hướng G = (V, E)

Có hai khái niệm về tính liên thông của đồ thị có hướng tuỳ theo chúng ta có quan tâm tới hướng của các cung không.

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



Liên thông mạnh Liên thông yếu

II. TÍNH LIÊN THÔNG TRONG ĐỒ THỊ VÔ HƯỚNG

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

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

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

- Đánh dấu đỉnh 1 và những đỉnh có thể đến từ 1, thông báo những đỉnh đó thuộc thành phần liên thông thứ nhất.
- Nếu tất cả các đỉnh đều đã bị đánh dấu thì G là đồ thị liên thông, nếu không thì sẽ tồn tại một đỉnh v nào đó chưa bị đánh dấu, ta sẽ đánh dấu v và các đỉnh có thể đến được từ v, thông báo những đỉnh đó thuộc thành phần liên thông thứ hai.
- Và cứ tiếp tục như vậy cho tới khi tất cả các đỉnh đều đã bị đánh dấu

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

begin
 for ∀ v ∈ V do <khởi tạo v chưa đánh dấu>;
 Count := 0;
```

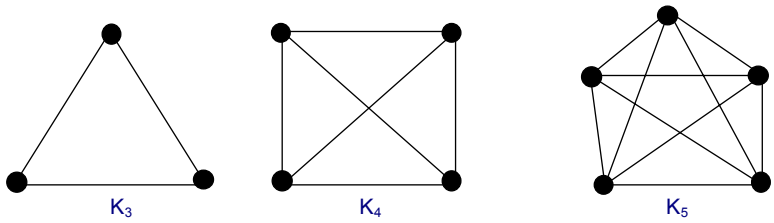
[1] Đồ thị G = (V, E) là con của đồ thị G' = (V', E') nếu G là đồ thị và $V \subseteq V'$ và $E \subseteq E'$

```
 for u := 1 to n do
  if <u chưa đánh dấu> then
   begin
    Count := Count + 1;
    Writeln('Thành phần liên thông thứ ', Count, ' gồm các đỉnh : ');
    Duyệt(u);
   end;
end.
```


III. ĐỒ THỊ ĐẦY ĐỦ VÀ THUẬT TOÁN WARSHALL

1. Định nghĩa:

Đồ thị đầy đủ với n đỉnh, ký hiệu K_n, là một đơn đồ thị vô hướng mà giữa hai đỉnh bất kỳ của nó đều có cạnh nối.



Đồ thị đầy đủ K_n có đúng:

$C_n^2 = \frac{n.(n-1)}{2}$ cạnh và bậc của mọi đỉnh đều bằng n - 1.

2. Bao đóng đồ thị:

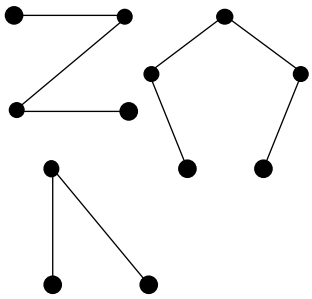
Với đồ thị G = (V, E), người ta xây dựng đồ thị G' = (V, E') cũng gồm những đỉnh của G còn các cạnh xây dựng như sau: (ở đây quy ước giữa u và u luôn có đường đi)

Giữa đỉnh u và v của G' có cạnh nối ⇔ Giữa đỉnh u và v của G có đường đi

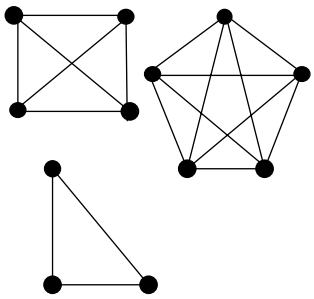
Đồ thị G' xây dựng như vậy được gọi là bao đóng của đồ thị G.

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

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



Đơn đồ thị vô hướng



Và bao đóng của nó

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

3. Thuật toán Warshall

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

Từ ma trận kề A của đơn đồ thị vô hướng G (a_{ij} = True nếu (i, j) là cạnh của G) ta sẽ sửa đổi A để nó trở thành ma trận kề của bao đóng bằng cách: **Với mọi đỉnh k xét theo thứ tự từ 1 tới n, ta xét tất cả các cặp đỉnh (u, v); nếu có cạnh nối (u, k) (a_{uk} = True) và có cạnh nối (k, v) (a_{kv} = True) thì ta tự nối thêm cạnh (u, v) nếu nó chưa có (đặt a_{uv} := True)**. Tư tưởng này dựa trên một quan sát

đơn giản như sau: Nếu từ u có đường đi tới k và từ k lại có đường đi tới v thì tất nhiên từ u sẽ có đường đi tới v.


```
{n là số đỉnh của đồ thị}
for k := 1 to n do
  for u := 1 to n do
    if a[u, k] then
      for v := 1 to n do
        if a[k, v] then a[u, v] := True;
```

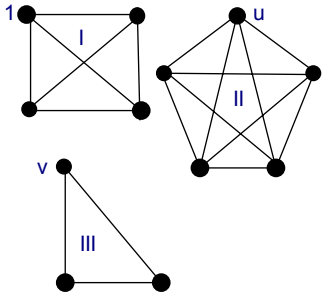
hoặc

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

Việc chứng minh tính đúng đắn của thuật toán đòi hỏi phải lật lại các lý thuyết về bao đóng bắc cầu và quan hệ liên thông, ta sẽ không trình bày ở đây mà sẽ thử cài đặt thuật toán Warshall tìm bao đóng của đơn đồ thị vô hướng sau đó đếm số thành phần liên thông của đồ thị:

Việc cài đặt thuật toán sẽ qua những bước sau:

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



Chương trình nhập dữ liệu về đồ thị từ file văn bản GRAPH.INP với khuôn dạng như trong các thuật toán tìm kiếm trên đồ thị ở trên.

```
program Connectivity;
const
 max = 100;
var
 a: array[1..max, 1..max] of Boolean; {Ma trận kề của đồ thị}
 Free: array[1..max] of Boolean;
 k, u, v, n: Byte;
 Count: Byte;

procedure Enter;
var
 f: Text;
 i, u, v: Byte;
 m: Word;
begin
 FillChar(a, SizeOf(a), False);
 Assign(f, 'GRAPH.INP'); Reset(f);
 Readln(f, n, m);
 for v := 1 to n do a[v, v] := True;
 for i := 1 to m do
  begin
   Readln(f, u, v);
   a[u, v] := True;
   a[v, u] := True;
  end;
 Close(f);
end;

begin
 Enter;
 for k := 1 to n do {Thuật toán Warshall}
```

```
  for u := 1 to n do
   for v := 1 to n do
    a[u, v] := a[u, v] or a[u, k] and a[k, v];
 Count := 0;
 FillChar(Free, n, True); {Các đỉnh đều chưa bị đánh dấu}
 for u := 1 to n do {Quét danh sách đỉnh}
  if Free[u] then {Nếu thấy một đỉnh u chưa bị đánh dấu (chưa liệt kê vào tp liên thông nào)}
   begin
    Inc(Count);
    Write('Connected Component ', Count, ': '); {Thành phần liên thông thứ Count gồm:}
    for v := 1 to n do
     if a[u, v] then {Các đỉnh v kề với u (tất nhiên có cả u)}
      begin
       Write(v, ', ');
       Free[v] := False; {Liệt kê đỉnh nào đánh dấu đỉnh đó}
      end;
    Writeln;
   end;
end.
```


IV. CÁC THÀNH PHẦN LIÊN THÔNG MẠNH

Đối với đồ thị có hướng, người ta quan tâm đến bài toán kiểm tra tính liên thông mạnh, hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông mạnh của đồ thị có hướng. Đối với bài toán đó ta có một phương pháp khá hữu hiệu dựa trên thuật toán tìm kiếm theo chiều sâu Depth First Search.

1. Phân tích

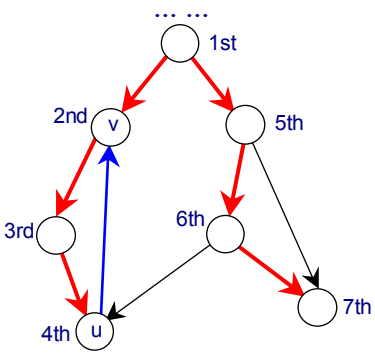
Thêm vào đồ thị một đỉnh x và nối x với tất cả các đỉnh còn lại của đồ thị bằng các cung định hướng. Khi đó quá trình tìm kiếm theo chiều sâu bắt đầu từ x có thể coi như một quá trình xây dựng cây tìm kiếm theo chiều sâu (cây DFS) gốc x.

```
procedure Visit(u∈V)
begin
 <Thêm u vào cây tìm kiếm DFS>
 for (∀v: (u, v)∈E) do
  if <v không thuộc cây DFS> then Visit(v);
end;

begin
 <Thêm vào đồ thị đỉnh x và các cung định hướng (x, v) với mọi v>
 <Khởi tạo cây tìm kiếm DFS := ∅>
 Visit(x)
end.
```

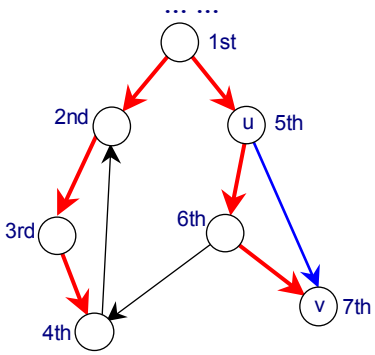
Để ý thủ tục thăm đỉnh đệ quy Visit(u). Thủ tục này xét tất cả những đỉnh v nối từ u, nếu v chưa được thăm thì đi theo cung đó thăm v, tức là bổ sung cung (u, v) vào cây tìm kiếm DFS. Nếu **v đã thăm** thì có ba khả năng xảy ra đối với vị trí của u và v trong cây tìm kiếm DFS:

1. v là tiền bối (ancestor - tổ tiên) của u, tức là v được thăm trước u và thủ tục Visit(u) do dây chuyền đệ quy từ thủ tục Visit(v) gọi tới. Cung (u, v) khi đó được gọi là **cung ngược** (Back edge)
2. v là hậu duệ (descendant - con cháu) của u, tức là u được thăm trước v, nhưng thủ tục Visit(u) sau khi tiến đệ quy theo một hướng khác đã gọi Visit(v) rồi. Nên khi dây chuyền đệ quy lùi lại về thủ tục Visit(u) sẽ thấy v là đã thăm nên không thăm lại nữa. Cung (u, v) khi đó gọi là **cung xuôi** (Forward edge).
3. v thuộc một nhánh của cây DFS đã duyệt trước đó, tức là sẽ có một đỉnh w được thăm trước cả u và v. Thủ tục Visit(w) gọi trước sẽ rẽ theo một nhánh nào đó thăm v trước, rồi khi lùi lại, rẽ sang một nhánh khác thăm u. Cung (u, v) khi đó gọi là **cung chéo** (Cross edge)

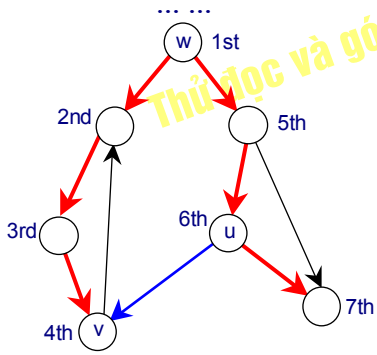
(Rất tiếc là từ điển thuật ngữ tin học Anh-Việt quá nghèo nàn nên không thể tìm ra những từ tương đương với các thuật ngữ ở trên. Ta có thể hiểu qua các ví dụ)



TH1: (v là tiền bối của u)
(u, v) là cung ngược



TH2: (v là hậu duệ của u)
(u, v) là cung xuôi



TH3: (v nằm ở nhánh đã duyệt trước u)
(u, v) là cung chéo

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

2. Cây tìm kiếm DFS và các thành phần liên thông mạnh

Định lý 1: Nếu a, b là hai đỉnh thuộc thành phần liên thông mạnh C thì với mọi đường đi từ a tới b cũng như từ b tới a. Tất cả đỉnh trung gian trên đường đi đó đều phải thuộc C.

Chứng minh

Nếu a và b là hai đỉnh thuộc C thì tức là có một đường đi từ a tới b và một đường đi khác từ b tới a. Suy ra với một đỉnh v nằm trên đường đi từ a tới b thì **a tới được v**, v tới được b, mà b có đường tới a nên **v cũng tới được a**. Vậy v nằm trong thành phần liên thông mạnh chứa a tức là $v \in C$. Tương tự với một đỉnh nằm trên đường đi từ b tới a.

Định lý 2: Với một thành phần liên thông mạnh C bất kỳ, sẽ tồn tại một đỉnh $r \in C$ sao cho mọi đỉnh của C đều thuộc nhánh DFS gốc r.

Chứng minh:

Trước hết, nhắc lại một thành phần liên thông mạnh là một đồ thị con liên thông mạnh của đồ thị ban đầu thoả mãn tính chất tối đại tức là việc thêm vào thành phần đó một tập hợp đỉnh khác sẽ làm mất đi tính liên thông mạnh.

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

$$(r = x_0, x_1, ..., x_k = v)$$

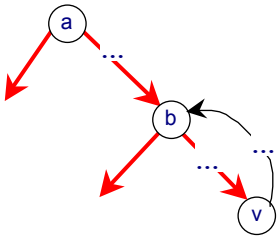
Từ định lý 1, tất cả các đỉnh $x_0, x_1, ..., x_k$ đều thuộc C nên chúng sẽ phải thăm sau đỉnh r. Khi thủ tục Visit(r) được gọi thì tất cả các đỉnh $x_0, ..., x_k=v$ đều chưa thăm; vì thủ tục Visit(r) sẽ liệt kê tất cả những đỉnh chưa thăm đến được từ r bằng cách xây dựng nhánh gốc r của cây DFS, nên các đỉnh $x_0, x_1, ..., x_k = v$ sẽ thuộc nhánh gốc r của cây DFS. Bởi chọn v là đỉnh bất kỳ trong C nên ta có điều phải chứng minh.

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

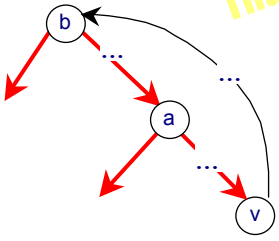
Định lý 3: Luôn tìm được đỉnh chốt a thoả mãn: Quá trình tìm kiếm theo chiều sâu bắt đầu từ a không thăm được bất kỳ một chốt nào khác. (Tức là nhánh DFS gốc a không chứa một chốt nào ngoài a) chẳng hạn ta chọn a là chốt được thăm sau cùng trong một dây chuyền đệ quy hoặc chọn a là chốt thăm sau tất cả các chốt khác. Với chốt a như vậy thì các đỉnh thuộc **nhánh DFS gốc a chính là thành phần liên thông mạnh** chứa a.

Chứng minh:

Với mọi đỉnh v nằm trong nhánh DFS gốc a, xét b là chốt của thành phần liên thông mạnh chứa v. Ta sẽ chứng minh a ≡ b. Thật vậy, theo định lý 2, v nằm trong nhánh DFS gốc b. Vậy v nằm trong cả nhánh DFS gốc a và nhánh DFS gốc b. Giả sử phản chứng rằng a≠b thì sẽ có hai khả năng xảy ra:



Khả năng 1: a→b→v



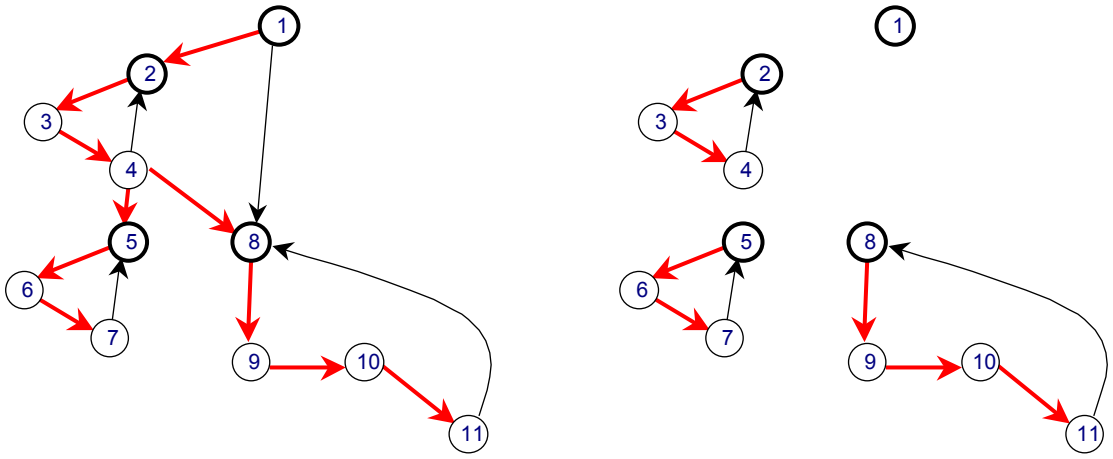
Khả năng 2: b→a→v

- Khả năng 1: Nhánh DFS gốc a chứa nhánh DFS gốc b, có nghĩa là thủ tục Visit(b) sẽ do thủ tục Visit(a) gọi tới, điều này mâu thuẫn với giả thiết rằng a là chốt mà quá trình tìm kiếm theo chiều sâu bắt đầu từ a không thăm một chốt nào khác.
- Khả năng 2: Nhánh DFS gốc a nằm trong nhánh DFS gốc b, có nghĩa là a nằm trên một đường đi từ b tới v. Do b và v thuộc cùng một thành phần liên thông mạnh nên theo định lý 1, a cũng phải thuộc thành phần liên thông mạnh đó. Vậy thì thành phần liên thông mạnh này có hai chốt a và b. Điều này vô lý.

Theo định lý 2, ta đã **có thành phần liên thông mạnh chứa a nằm trong nhánh DFS gốc a**, theo chứng minh trên ta lại có: Mọi đỉnh trong **nhánh DFS gốc a nằm trong thành phần liên thông mạnh chứa a**. Kết hợp lại được: Nhánh DFS gốc a chính là thành phần liên thông mạnh chứa a.

3. Thuật toán Tarjan (R.E.Tarjan - 1972)

Chọn u là chốt mà từ đó quá trình tìm kiếm theo chiều sâu không thăm thêm bất kỳ một chốt nào khác, chọn lấy thành phần liên thông mạnh thứ nhất là nhánh DFS gốc u. Sau đó loại bỏ nhánh DFS gốc u ra khỏi cây DFS, lại tìm thấy một đỉnh chốt v khác mà nhánh DFS gốc v không chứa chốt nào khác, lại chọn lấy thành phần liên thông mạnh thứ hai là nhánh DFS gốc v. Tương tự như vậy cho thành phần liên thông mạnh thứ ba, thứ tư, v.v... Có thể hình dung thuật toán Tarjan "bẻ" cây DFS tại vị trí các chốt để được các nhánh rời rạc, mỗi nhánh là một thành phần liên thông mạnh.



Thuật toán Tarjan "bẻ" cây DFS

Trình bày dài dòng như vậy, nhưng điều quan trọng nhất bây giờ mới nói tới: **Làm thế nào kiểm tra một đỉnh v nào đó có phải là chốt hay không ?**

Hãy để ý nhánh DFS gốc ở đỉnh r nào đó.

Nhận xét 1: ***Nếu như từ các đỉnh thuộc nhánh gốc r này không có cung ngược hay cung chéo nào đi ra khỏi nhánh đó thì r là chốt***. Điều này dễ hiểu bởi như vậy có nghĩa là từ r, đi theo các cung của đồ thị thì chỉ đến được những đỉnh thuộc nhánh đó mà thôi. Vậy:

Thành phần liên thông mạnh chứa r ⊂ Tập các đỉnh có thể đến từ r = Nhánh DFS gốc r

nên r là chốt.

Nhận xét 2: ***Nếu từ một đỉnh v nào đó của nhánh DFS gốc r có một cung ngược tới một đỉnh w là tiền bối của r, thì r không là chốt***. Thật vậy: do có chu trình (w→r→v→w) nên w, r, v thuộc cùng một thành phần liên thông mạnh. Mà w được thăm trước r, điều này mâu thuẫn với cách xác định chốt (Xem lại định lý 2)

Nhận xét 3: Vấn đề phức tạp gặp phải ở đây là nếu từ một đỉnh v của nhánh DFS gốc r, có một cung chéo đi tới một nhánh khác. Ta sẽ thiết lập giải thuật liệt kê thành phần liên thông mạnh ngay trong thủ tục Visit(u), khi mà đỉnh u đã **duyệt xong**, tức là khi **các đỉnh khác của nhánh DFS gốc u đều đã thăm**. Nếu như u là chốt, ta thông báo nhánh DFS gốc u là thành phần liên thông mạnh chứa u và loại ngay các đỉnh thuộc thành phần đó khỏi đồ thị cũng như khỏi cây DFS. Có thể chứng minh được tính đúng đắn của phương pháp này, bởi nếu nhánh DFS gốc u chứa một chốt u' khác thì u' phải duyệt xong trước u và cả nhánh DFS gốc u' đã bị loại bỏ rồi. Hơn nữa còn có thể chứng minh được rằng, khi thuật toán tiến hành như trên thì nếu như **từ một đỉnh v của một nhánh DFS gốc r có một cung chéo đi tới một nhánh khác thì r không là chốt**.

Để chứng tỏ điều này, ta dựa vào tính chất của cây DFS: cung chéo sẽ nối từ một nhánh tới nhánh thăm trước đó, chứ không bao giờ có cung chéo đi tới nhánh thăm sau. Giả sử có cung chéo (v, w) đi từ v ∈ nhánh DFS gốc r tới w ∉ nhánh DFS gốc r, gọi z là chốt của thành phần liên thông chứa w. Theo tính chất trên, w phải thăm trước r, suy ra **z cũng phải thăm trước r**. Có hai khả năng xảy ra:

- Nếu z thuộc nhánh DFS đã duyệt trước r thì z sẽ được duyệt xong trước khi thăm r, tức là khi thăm r và cả sau này khi thăm v thì nhánh DFS gốc z đã bị huỷ, cung chéo (v, w) sẽ không được tính đến nữa.
- Nếu z là tiền bối của r thì ta có **z đến được r**, v nằm trong nhánh DFS gốc r nên **r đến được v**, **v đến được w** vì (v, w) là cung, **w lại đến được z** bởi z là chốt của thành phần liên thông mạnh chứa v. Ta thiết lập được chu trình (z→r→v→w→z), suy ra z và r thuộc cùng một thành phần liên thông mạnh, z là chốt nên r không thể là chốt nữa.

Từ ba nhận xét và cách cài đặt chương trình như trong nhận xét 3, Ta có: Đỉnh r là chốt **nếu và chỉ nếu** không tồn tại cung ngược hoặc cung chéo nối một đỉnh thuộc nhánh DFS gốc r với một đỉnh ngoài nhánh đó, hay nói cách khác: **r là chốt nếu và chỉ nếu không tồn tại cung nối từ một đỉnh thuộc nhánh DFS gốc r tới một đỉnh thăm trước r**.

Dưới đây là một cài đặt hết sức thông minh, chỉ cần sửa đổi một chút thủ tục Visit ở trên là ta có ngay phương pháp này. Nội dung của nó là đánh số thứ tự các đỉnh từ đỉnh được thăm đầu tiên đến đỉnh thăm sau cùng. Định nghĩa Numbering[u] là số thứ tự của đỉnh u theo cách đánh số đó. Ta tính thêm Low[u] là giá trị Numbering nhỏ nhất trong các đỉnh có thể đến được từ một đỉnh v nào đó của nhánh DFS gốc u bằng một cung (với giả thiết rằng u có một cung giả nối với chính u).

Cụ thể cách cực tiểu hoá Low[u] như sau:

Trong thủ tục Visit(u), trước hết ta đánh số thứ tự thăm cho đỉnh u và khởi gán

$$Low[u] := Numbering[u] \text{ (u có cung tới chính u)}$$

Xét tất cả những đỉnh v nối từ u:

- Nếu v đã thăm thì ta cực tiểu hoá Low[u] theo công thức:
$$Low[u]_{mới} := \min(Low[u]_{cũ}, Numbering[v]).$$
- Nếu v chưa thăm thì ta gọi đệ quy đi thăm v, sau đó cực tiểu hoá Low[u] theo công thức:
$$Low[u]_{mới} := \min(Low[u]_{cũ}, Low[v])$$

Dễ dàng chứng minh được tính đúng đắn của công thức tính.

Khi duyệt xong một đỉnh u (chuẩn bị thoát khỏi thủ tục Visit(u). Ta so sánh Low[u] và Numbering[u]. Nếu như Low[u] = Numbering[u] thì u là chốt, bởi không có cung nối từ một đỉnh thuộc nhánh DFS gốc u tới một đỉnh thăm trước u. Khi đó chỉ việc liệt kê các đỉnh thuộc thành phần liên thông mạnh chứa u là nhánh DFS gốc u.

Để công việc dễ dàng hơn nữa, ta định nghĩa một danh sách L được tổ chức dưới dạng ngăn xếp và dùng ngăn xếp này để lấy ra các đỉnh thuộc một nhánh nào đó. Khi thăm tới một đỉnh u, ta đẩy ngay đỉnh u đó vào ngăn xếp, thì sau đó khi duyệt xong đỉnh u, mọi đỉnh thuộc nhánh DFS gốc u sẽ được đẩy vào ngăn xếp L ngay sau u. Nếu u là chốt, ta chỉ việc lấy các đỉnh ra khỏi ngăn xếp L cho tới khi lấy tới đỉnh u là sẽ được nhánh DFS gốc u cũng chính là thành phần liên thông mạnh chứa u.

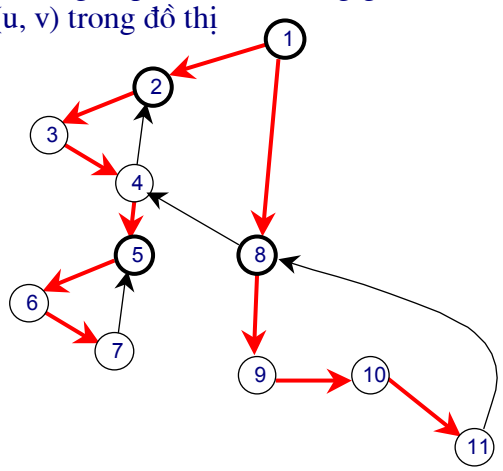
```
procedure Visit(u∈V)
begin
  Count := Count + 1; Numbering[u] := Count; {Trước hết đánh số u}
  Low[u] := Numbering[u];
  <Đưa u vào cây DFS>
  <Đẩy u vào ngăn xếp L>
  for (∀v: (u, v)∈E) do
    if <v đã thăm> then
      Low[u] := min(Low[u], Numbering[v])
    else
      begin
        Visit(v)
        Low[u] := min(Low[u], Low[v])
      end;
  if Numbering[u] = Low[u] then {Nếu u là chốt}
    begin
      <Thông báo thành phần liên thông mạnh với chốt u gồm có các đỉnh:>
      repeat
        <Lấy từ ngăn xếp L ra một đỉnh v>
        <Output v>
        <Xoá đỉnh v khỏi đồ thị>
      until v = u;
    end;
end;

begin
 <Thêm vào đồ thị một đỉnh x và các cung (x, v) với mọi v>
 <Khởi tạo một biến đếm Count := 0>
 <Khởi tạo một ngăn xếp L := ∅>
 <Khởi tạo cây tìm kiếm DFS := ∅>
 Visit(x)
end.
```

Mọi thứ đã sẵn sàng, dưới đây là toàn bộ chương trình. Trong chương trình này, ta sử dụng:

- Ma trận kề A để biểu diễn đồ thị.
- Mảng Visited kiểu Boolean được dùng để đánh dấu: Visited[u] = True ⇔ u đã thăm (dùng cho thủ tục Visit - tìm kiếm theo chiều sâu)
- Mảng Free kiểu Boolean, Free[u] = True nếu u chưa bị liệt kê vào thành phần liên thông nào, tức là u chưa bị loại khỏi đồ thị.
- Mảng Stack, thủ tục Push, hàm Pop để mô tả cấu trúc ngăn xếp.

Dữ liệu về đồ thị được nhập từ file văn bản GRAPH.INP:

- Dòng đầu: Ghi số đỉnh n và số cung m của đồ thị cách nhau một dấu cách
- m dòng tiếp theo, mỗi dòng ghi hai số nguyên u, v cách nhau một dấu cách thể hiện có cung (u, v) trong đồ thị



Đồ thị, file dữ liệu tương ứng và Output

GRAPH.INP	OUTPUT
11 14 1 2 1 8 2 3 3 4 4 2 4 5 5 6 6 7 7 5 8 4 8 9 9 10 10 11 11 8	Component 1: 7, 6, 5, Component 2: 4, 3, 2, Component 3: 11, 10, 9, 8, Component 4: 1,

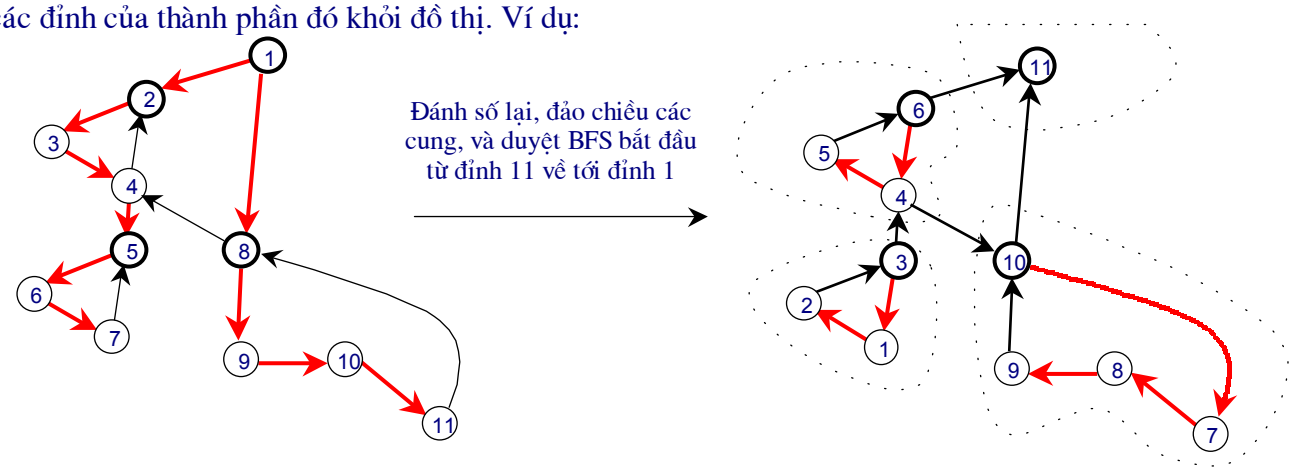
```
program Strong_connectivity; {Các thành phần liên thông mạnh}
const
 max = 100;
var
 A: array[1..max, 1..max] of Boolean;
 Visited, Free: array[1..max] of Boolean;
 Numbering, Low, Stack: array[1..max] of Byte;
 n, Count, ComponentCount, Last: Byte;

procedure Enter; {Nhập dữ liệu}
var
 f: Text;
 i, u, v: Byte;
 m: Word;
begin
 FillChar(A, SizeOf(A), False);
 Assign(f, 'GRAPH.INP'); Reset(f);
 Readln(f, n, m);
 for i := 1 to m do
  begin
   Readln(f, u, v);
   A[u, v] := True;
  end;
 Close(f);
end;

procedure Init; {Khởi tạo}
begin
 FillChar(Visited, SizeOf(Visited), False); {Các đỉnh đều chưa thăm}
 FillChar(Free, SizeOf(Free), True); {Các đỉnh đều còn trong đồ thị, chưa bị loại}
 Last := 0; {Ngăn xếp rỗng}
 Count := 0; {Biến đánh số theo thứ tự thăm}
 ComponentCount := 0; {Biến đánh số các thành phần liên thông}
end;

procedure Push(v: Byte); {Đẩy đỉnh v vào ngăn xếp Stack}
begin
 Inc(Last);
 Stack[Last] := v;
end;

function Pop: Byte; {Lấy 1 đỉnh khỏi ngăn xếp, trả về trong kết quả hàm}
begin
 Pop := Stack[Last];
 Dec(Last);
end;

function Min(a, b: Byte): Byte;
begin
 if a < b then Min := a else Min := b;
end;

procedure Visit(u: Byte); {Thuật toán tìm kiếm theo chiều sâu}
var
 v: Byte;
begin
 Inc(Count); Numbering[u] := Count; {Trước hết đánh số u theo thứ tự thăm}
 Low[u] := Numbering[u]; {u có cung tới u nên có thể khởi gán Low[u] thế này rồi sau cực tiểu hoá dần}
 Visited[u] := True; {Đánh dấu u đã thăm}
 Push(u); {Đưa u vào ngăn xếp}
 for v := 1 to n do
  if Free[v] and A[u, v] then {Chỉ xét những đỉnh v chưa bị loại bỏ khỏi đồ thị có (u, v) ∈ E}
```

```
    if Visited[v] then {Nếu v đã thăm}
     Low[u] := Min(Low[u], Numbering[v]) {Thì cực tiểu hoá Low[u] theo công thức này}
    else {Nếu v chưa thăm}
     begin
      Visit(v); {Trước hết thăm v}
      Low[u] := Min(Low[u], Low[v]); {Rồi cực tiểu hoá Low[u] theo công thức này}
     end;
  {Đến đây thì đỉnh u được duyệt xong, tức là các đỉnh thuộc nhánh DFS gốc u đều đã thăm}
  if Numbering[u] = Low[u] then {Nếu u là chốt}
   begin
    Inc(ComponentCount); {Bắt đầu liệt kê thành phần liên thông mạnh}
    Write('Component ', ComponentCount, ': ');
    repeat
     v := Pop; {Lấy dần các đỉnh khỏi ngăn xếp}
     Write(v, ', ');
     Free[v] := False; {Lấy ra đỉnh nào, liệt kê ra và loại ngay đỉnh đó khỏi đồ thị}
    until v = u; {Cho tới khi lấy tới u: toàn bộ các đỉnh ∈ nhánh DFS gốc u đã được liệt kê và đã được huỷ}
    Writeln;
   end;
end;

procedure Solve;
var
 u: Byte;
begin
  {Thay vì thêm một đỉnh giả x và các cung (x, v) với mọi đỉnh v rồi gọi Visit(x), ta có thể làm thế này cho nhanh, sau này đỡ phải huỷ bỏ thành phần liên thông gồm mỗimột đỉnh giả đó}
 for u := 1 to n do
  if not Visited[u] then Visit(u);
end;

begin
 Enter;
 Init;
 Solve;
end.
```


Bài tập:

1. Phương pháp cài đặt như trên có thể nói là rất hay và hiệu quả, đòi hỏi ta phải hiểu rõ bản chất thuật toán, nếu không thì rất dễ nhầm. Trên thực tế, còn có một phương pháp khác dễ hiểu hơn, tuy tính hiệu quả có kém hơn một chút. Hãy viết chương trình mô tả phương pháp sau:

Vẫn dùng thuật toán tìm kiếm theo chiều sâu với thủ tục Visit nói ở đầu mục, đánh số lại các đỉnh từ 1 tới n theo thứ tự **duyệt xong**, sau đó đảo chiều tất cả các cung của đồ thị. Xét lần lượt các đỉnh theo thứ tự từ đỉnh duyệt xong sau cùng tới đỉnh duyệt xong đầu tiên, với mỗi đỉnh đó, ta lại dùng thuật toán tìm kiếm trên đồ thị (BFS chẳng hạn) liệt kê những đỉnh nào đến được từ đỉnh đang xét, đó chính là một thành phần liên thông mạnh. Lưu ý là khi liệt kê xong thành phần nào, ta loại ngay các đỉnh của thành phần đó khỏi đồ thị. Ví dụ:



2. Thuật toán Warshall có thể áp dụng tìm bao đóng của đồ thị có hướng, vậy hãy kiểm tra tính liên thông mạnh của một đồ thị có hướng bằng hai cách: Dùng các thuật toán tìm kiếm trên đồ thị và thuật toán Warshall, sau đó so sánh ưu, nhược điểm của mỗi phương pháp

3. Mê cung hình chữ nhật kích thước m x n gồm các ô vuông đơn vị. Trên mỗi ô ký tự:
O: Nếu ô đó an toàn
X: Nếu ô đó có cạm bẫy
E: Nếu là ô có một nhà thám hiểm đang đứng.
Duy nhất chỉ có 1 ô ghi chữ E. Nhà thám hiểm có thể từ một ô đi sang một trong số các ô chung cạnh với ô đang đứng. Một cách đi thoát khỏi mê cung là một hành trình đi qua các ô an toàn ra một ô biên. Hãy chỉ giúp cho nhà thám hiểm một hành trình thoát ra khỏi mê cung

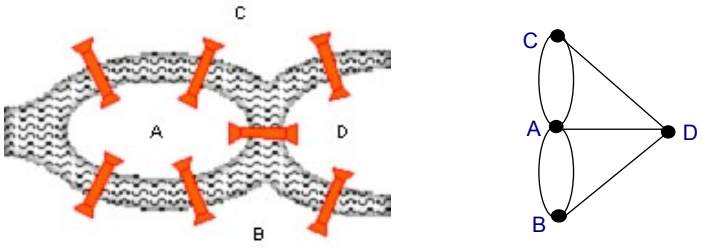
4. Lập chương trình kiểm tra xem một đỉnh v của đồ thị có nằm trên một chu trình nào không ?

5. Trên mặt phẳng với hệ toạ độ Decattes vuông góc cho n đường tròn, mỗi đường tròn xác định bởi bộ 3 số thực (X, Y, R) ở đây (X, Y) là toạ độ tâm và R là bán kính. Hai đường tròn gọi là thông nhau nếu chúng có điểm chung. Hãy chia các đường tròn thành một số tối thiểu các nhóm sao cho hai đường tròn bất kỳ trong một nhóm bất kỳ có thể đi được sang nhau sau một số hữu hạn các bước di chuyển giữa hai đường tròn thông nhau.

§5. CHU TRÌNH EULER, ĐƯỜNG ĐI EULER, ĐỒ THỊ EULER

I. BÀI TOÁN 7 CÁI CẦU

Thành phố Konigsberg thuộc Phổ (nay là Kaliningrad thuộc Cộng hoà Nga), được chia làm 4 vùng bằng các nhánh sông Pregel. Các vùng này gồm 2 vùng bên bờ sông (B, C), đảo Kneiphof (A) và một miền nằm giữa hai nhánh sông Pregel (D). Vào thế kỷ XVIII, người ta đã xây 7 chiếc cầu nối những vùng này với nhau. Người dân ở đây tự hỏi: Liệu có cách nào xuất phát tại một địa điểm trong thành phố, đi qua 7 chiếc cầu, mỗi chiếc đúng 1 lần rồi quay trở về nơi xuất phát không ?
Nhà toán học Thụy sĩ Leonhard Euler đã giải bài toán này và có thể coi đây là ứng dụng đầu tiên của Lý thuyết đồ thị, ông đã mô hình hoá sơ đồ 7 cái cầu bằng một đa đồ thị, bốn vùng được biểu diễn bằng 4 đỉnh, các cầu là các cạnh. Bài toán tìm đường qua 7 cầu mỗi cầu đúng một lần có thể tổng quát hoá bằng bài toán: **Có tồn tại chu trình đơn trong đa đồ thị chứa tất cả các cạnh ?.**



7 cái cầu ở Konigsberg

II. ĐỊNH NGHĨA

1. Chu trình đơn chứa tất cả các cạnh của đồ thị được gọi là chu trình Euler
2. Đường đi đơn chứa tất cả các cạnh của đồ thị được gọi là đường đi Euler
3. Một đồ thị có chu trình Euler được gọi là đồ thị Euler
4. Một đồ thị có đường đi Euler được gọi là đồ thị nửa Euler.

Rõ ràng một đồ thị Euler thì phải là nửa Euler nhưng điều ngược lại thì không phải luôn đúng

III. ĐỊNH LÝ

1. Một đồ thị **vô hướng liên thông** G = (V, E) có **chu trình Euler** khi và chỉ khi mọi đỉnh của nó đều có bậc chẵn: $\deg(v) \equiv 0 \pmod 2$ $(\forall v \in V)$
2. Một đồ thị vô hướng liên thông **có đường đi Euler nhưng không có chu trình Euler** khi và chỉ khi nó có đúng 2 đỉnh bậc lẻ
3. Một đồ thi **có hướng liên thông yếu** G = (V, E) có **chu trình Euler** thì mọi đỉnh của nó có bán bậc ra bằng bán bậc vào: $\deg^+(v) = \deg^-(v)$ $(\forall v \in V)$; Ngược lại, nếu G **liên thông yếu** và mọi đỉnh của nó có bán bậc ra bằng bán bậc vào thì G có **chu trình Euler**, hay G sẽ là **liên thông mạnh.**
4. Một đồ thị có hướng liên thông yếu G = (V, E) có **đường đi Euler nhưng không có chu trình Euler** nếu tồn tại đúng hai đỉnh u, $v \in V$ sao cho $\deg^+(u) - \deg^-(u) = \deg^-(v) - \deg^+(v) = 1$, còn tất cả những đỉnh khác u và v đều có bán bậc ra bằng bán bậc vào.

IV. THUẬT TOÁN FLEURY TÌM CHU TRÌNH EULER

1. Đối với đồ thị vô hướng liên thông, mọi đỉnh đều có bậc chẵn.

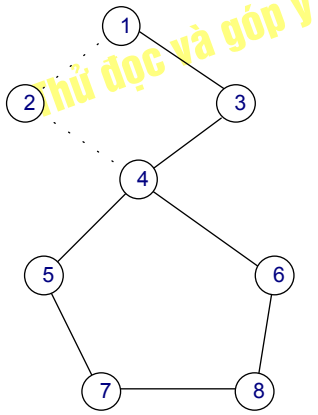
Xuất phát từ một đỉnh, ta chọn một cạnh liên thuộc với nó để đi tiếp sang đỉnh khác theo hai nguyên tắc sau:

- Xoá bỏ cạnh đã đi qua
- Chỉ đi qua cầu khi không còn cạnh nào khác để chọn

Và ta cứ chọn cạnh đi một cách thoải mái như vậy cho tới khi không đi tiếp được nữa, đường đi tìm được là chu trình Euler.

Ví dụ: Với đồ thị sau:

Nếu xuất phát từ đỉnh 1, có hai cách đi tiếp: hoặc sang 2 hoặc sang 3, giả sử ta sẽ sang 2 và xoá cạnh (1, 2) vừa đi qua. Từ 2 chỉ có cách duy nhất là sang 4, nên cho dù (2, 4) là cầu ta cũng phải đi sau đó xoá luôn cạnh (2, 4). Đến đây, các cạnh còn lại của đồ thị có thể vẽ như hình bên bằng nét liền, các cạnh đã bị xoá được vẽ bằng nét đứt.



Bây giờ đang đứng ở đỉnh 4 thì ta có 3 cách đi tiếp: sang 3, sang 5 hoặc sang 6. Vì (4, 3) là cầu nên ta sẽ không đi theo cạnh (4, 3) mà sẽ đi (4, 5) hoặc (4, 6). Nếu đi theo (4, 5) và cứ tiếp tục đi như vậy, ta sẽ được chu trình Euler là (1, 2, 4, 5, 7, 8, 6, 4, 3, 1). Còn đi theo (4, 6) sẽ tìm được chu trình Euler là: (1, 2, 4, 6, 8, 7, 5, 4, 3, 1).

2. Đối với đồ thị có hướng liên thông yếu, mọi đỉnh đều có bán bậc ra bằng bán bậc vào.

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

- Thứ nhất, dưới đây nếu ta nói cạnh (u, v) thì hiểu là cạnh nối đỉnh u và đỉnh v trên đồ thị vô hướng, hiểu là cung nối từ đỉnh u tới đỉnh v trên đồ thị có hướng.
- Thứ hai, ta gọi cạnh (u, v) là "một đi không trở lại" nếu như từ u ta đi tới v theo cạnh đó, sau đó xoá cạnh đó đi thì không có cách nào từ v quay lại u.

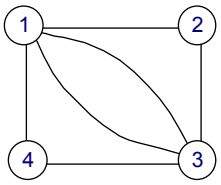
Vậy thì thuật toán Fleury tìm chu trình Euler có thể mô tả như sau:
Xuất phát từ một đỉnh, ta đi một cách tuỳ ý theo các cạnh tuân theo hai nguyên tắc: Xoá bỏ cạnh vừa đi qua và chỉ chọn cạnh "một đi không trở lại" nếu như không còn cạnh nào khác để chọn.

V. CÀI ĐẶT

Ta sẽ cài đặt thuật toán Fleury trên một đa đồ thị vô hướng, để đơn giản, ta coi đồ thị này đã có chu trình Euler, công việc của ta là tìm ra chu trình đó thôi. Bởi việc kiểm tra tính liên thông cũng như kiểm tra mọi đỉnh đều có bậc chẵn đến giờ có thể coi là chuyện nhỏ.
Và để tiết kiệm thời gian nhập liệu, chương trình quy định dữ liệu về đồ thị được vào từ file văn bản EULER.INP. Trong đó:

- Dòng 1: Ghi số đỉnh n của đồ thị
- Các dòng tiếp theo, mỗi dòng ghi 3 số nguyên dương cách nhau 1 dấu cách có dạng: u v k cho biết giữa đỉnh u và đỉnh v có k cạnh nối

Ví dụ:



EULER.INP	OUTPUT
4 1 2 1 2 3 1 3 4 1 4 1 1 1 3 2	1-->2-->3-->1-->3-->4-->1

Đa đồ thị Euler và file dữ liệu tương ứng và Output của chương trình

```
program Euler_Circuit; {Tìm chu trình Euler bằng thuật toán Fleury}
const
 max = 100;
var
 a: array[1..max, 1..max] of Word; {Ma trận kề: a[u, v] = a[v, u] = số cạnh nối u và v}
 n: Byte;

procedure Enter; {Nhập dữ liệu}
var
 f: Text;
 u, v: Byte;
 k: Word;
begin
 FillChar(a, SizeOf(a), 0); {Khởi tạo ma trận kề toàn 0: Đồ thị chưa có cạnh nào}
 Assign(f, 'EULER.INP'); Reset(f); {Mở file dữ liệu}
```

```
 Readln(f, n);                                  {Đọc dòng đầu tiên ra số đỉnh n}
 while not SeekEof(f) do          {Đọc từng dòng ra bộ ba (u, k, v) cho tới khi hết file}
  begin
   Readln(f, u, v, k);
   a[u, v] := k;
   a[v, u] := k;
  end;
 Close(f);
end;

{Thủ tục này kiểm tra nếu xoá một cạnh nối (x, y) thì y có còn quay lại được x hay không}
function CanGoBack(x, y: Byte): Boolean;
var
 Queue: array[1..max] of Byte; {Hàng đợi dùng cho Breadth First Search}
 First, Last: Byte; {First: Chỉ số đầu hàng đợi, Last: Chỉ số cuối hàng đợi}
 u, v: Byte;
 Free: array[1..max] of Boolean; {Mảng đánh dấu}
begin
 Dec(a[x, y]); Dec(a[y, x]);       {Thử xoá một cạnh (x, y) ⇔ Số cạnh nối (x, y) giảm 1}
 FillChar(Free, n, True);                {sau đó áp dụng BFS để xem từ y có quay lại x được không ?}
 Free[y] := False;
 First := 1; Last := 1;
 Queue[1] := y;
 repeat
  u := Queue[First]; Inc(First);
  for v := 1 to n do
   if Free[v] and (a[u, v] > 0) then
    begin
     Inc(Last);
     Queue[Last] := v;
     Free[v] := False;
     if Free[x] then Break;
    end;
 until First > Last;
 CanGoBack := not Free[x];
 Inc(a[x, y]); Inc(a[y, x]); {ở trên đã thử xoá cạnh thì giờ phải phục hồi}
end;

procedure FindEulerCircuit; {Thuật toán Fleury}
var
 Current, Next, v: Byte;
begin
 Current := 1;
 Write(1); {Bắt đầu từ đỉnh Current = 1}
 repeat
  Next := 0;
  for v := 1 to n do
   if a[Current, v] > 0 then
    begin
     Next := v;
     if CanGoBack(Current, Next) then Break;
    end;
  if Next <> 0 then
   begin
    Dec(a[Current, Next]);
    Dec(a[Next, Current]);          {Xoá bỏ cạnh vừa đi qua}
    Write('-->', Next);             {In kết quả đi tới Next}
    Current := Next;                {Lại tiếp tục với đỉnh đang đứng là Next}
   end;
 until Next = 0;     {Cho tới khi không đi tiếp được nữa}
 Writeln;
end;
```

Phân tích kỹ chỗ này để giải thích tại sao khi vòng lặp for v kết thúc, từ đỉnh đang đứng Current nó chọn đỉnh next để đi tiếp theo nguyên tắc chỉ chọn cầu khi không còn cách nào khác. Và nếu cả cầu cũng không còn thì Next = 0.

```
begin
 Enter;
 FindEulerCircuit;
end.
```


VI. THUẬT TOÁN TỐT HƠN

Trong trường hợp đồ thị Euler có **số cạnh đủ nhỏ**, ta có thể sử dụng phương pháp sau để tìm chu trình Euler trong đồ thị vô hướng: Bắt đầu từ một chu trình đơn C bất kỳ, chu trình này tìm được bằng cách xuất phát từ một đỉnh, đi tuỳ ý theo các cạnh cho tới khi quay về đỉnh xuất phát, lưu ý là đi qua cạnh nào xoá luôn cạnh đó. Nếu như chu trình C tìm được chứa tất cả các cạnh của đồ thị thì đó là chu trình Euler. Nếu không, xét các đỉnh dọc theo chu trình C, nếu còn có cạnh chưa xét liên thuộc với một đỉnh u nào đó thì lại từ u, ta đi tuỳ ý theo các cạnh cũng theo nguyên tắc trên cho tới khi quay trở về u, để được một chu trình đơn khác qua u. Loại bỏ vị trí u khỏi chu trình C và chèn vào C chu trình mới tìm được tại đúng vị trí của u vừa xoá, ta được một chu trình đơn C' mới lớn hơn chu trình C. Cứ làm như vậy cho tới khi được chu trình Euler. Việc chứng minh tính đúng đắn của thuật toán cũng là chứng minh định lý về điều kiện cần và đủ để một đồ thị vô hướng liên thông có chu trình Euler.

Mô hình thuật toán có thể viết như sau:

```
<Khởi tạo một ngăn xếp Stack ban đầu chỉ gồm mỗi đỉnh 1>
<Mô tả các phương thức Push (đẩy vào) và Pop(lấy ra) một đỉnh từ ngăn xếp Stack,
phương thức Get cho biết phần tử nằm ở đỉnh Stack. Khác với Pop, phương thức Get
chỉ cho biết phần tử ở đỉnh Stack chứ không lấy phần tử đó ra>
while Stack ≠∅ do
begin
 x := Get;
 if <Tồn tại đỉnh y mà (x, y)∈E> then {Từ x còn đi hướng khác được}
  begin
   Push(y);
   <Loại bỏ cạnh (x, y) khỏi đồ thị>
  end
 else {Từ x không đi tiếp được tới đâu nữa}
  begin
   x := Pop;
   <In ra đỉnh x trên đường đi Euler>
  end;
end;
```

Thuật toán trên có thể dùng để tìm chu trình Euler trong đồ thị có hướng liên thông yếu, mọi đỉnh có bán bậc ra bằng bán bậc vào. Tuy nhiên thứ tự các đỉnh in ra bị ngược so với các cung định hướng, ta có thể đảo ngược hướng các cung trước khi thực hiện thuật toán để được thứ tự đúng.

Thuật toán hoạt động với hiệu quả cao, dễ cài đặt, nhưng trường hợp xấu nhất thì Stack sẽ phải chứa toàn bộ danh sách đỉnh trên chu trình Euler chính vì vậy mà khi đa đồ thị có số cạnh quá lớn thì sẽ không đủ không gian nhớ mô tả Stack (Ta cứ thử với đồ thị chỉ gồm 2 đỉnh nhưng giữa hai đỉnh đó có tới 10^9 cạnh nối sẽ thấy ngay). Lý do thuật toán chỉ có thể áp dụng trong trường hợp số cạnh có giới hạn biết trước đủ nhỏ là như vậy. Thuật toán Fleury hoạt động chậm hơn, nhưng có thể cài đặt trên đồ thị với số cạnh lớn.

Như vậy tuỳ theo trường hợp cụ thể, ta có thể áp dụng thuật toán trên hay thuật toán Fleury để cho hiệu suất cao nhất. Đây là một ví dụ về một thuật toán rất tốt trên lý thuyết, nhưng khi cài đặt nhiều khi lại không tốt, phải lựa chọn thuật toán tồi hơn để làm.

Bài tập:

1. Chứng minh 4 định lý trong bài
2. Cài đặt thuật toán Fleury trên đa đồ thị có hướng.
3. Viết chương trình nhập vào hai số n, m và tạo ngẫu nhiên một đa đồ thị Euler có hướng gồm n đỉnh, m cung. Sau đó tự test bài 2 bằng cách ghi dữ liệu vào file EULER.INP rồi kiểm tra chu trình Euler tìm được có qua đúng m cạnh không?. Làm tương tự đối với đa đồ thị Euler vô hướng.

§6. CHU TRÌNH HAMILTON, ĐƯỜNG ĐI HAMILTON, ĐỒ THỊ HAMILTON

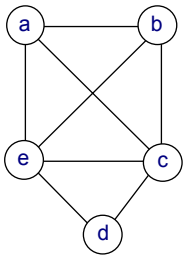
I. ĐỊNH NGHĨA

Cho đồ thị G = (V, E) có n đỉnh

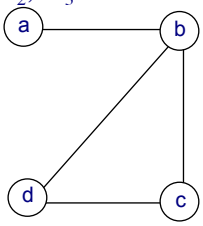
1. Chu trình $(x_1, x_2, ..., x_n, x_1)$ được gọi là chu trình Hamilton nếu $x_i \neq x_j$ với $1 \le i < j \le n$
2. Đường đi $(x_1, x_2, ..., x_n)$ được gọi là đường đi Hamilton nếu $x_i \neq x_j$ với $1 \le i < j \le n$

Có thể phát biểu một cách hình thức: Chu trình Hamilton là chu trình xuất phát từ 1 đỉnh, đi thăm tất cả những đỉnh còn lại mỗi đỉnh đúng 1 lần, cuối cùng quay trở lại đỉnh xuất phát. Đường đi Hamilton là đường đi qua tất cả các đỉnh của đồ thị, mỗi đỉnh đúng 1 lần. Khác với khái niệm chu trình Euler và đường đi Euler, một chu trình Hamilton không phải là đường đi Hamilton bởi có đỉnh xuất phát được thăm tới 2 lần.

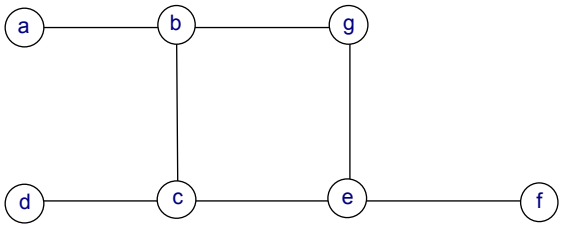
Ví dụ: Xét 3 đơn đồ thị G_1, G_2, G_3 sau:



G_1



G_2



G_3

Đồ thị G_1 có chu trình Hamilton (a, b, c, d, e, a). G_2 không có chu trình Hamilton vì deg(a) = 1 nhưng có đường đi Hamilton (a, b, c, d). G_3 không có cả chu trình Hamilton lẫn đường đi Hamilton

II. ĐỊNH LÝ

1. Đồ thị vô hướng G, trong đó tồn tại k đỉnh sao cho nếu xoá đi k đỉnh này cùng với những cạnh liên thuộc của chúng thì đồ thị nhận được sẽ có nhiều hơn k thành phần liên thông. Thì khẳng định là G không có chu trình Hamilton. Mệnh đề phản đảo của định lý này cho ta điều kiện cần để một đồ thị có chu trình Hamilton

2. Định lý Dirac (1952)): Đồ thị vô hướng G có n đỉnh ($n \ge 3$). Khi đó nếu mọi đỉnh v của G đều có $deg(v) \ge n/2$ thì G có chu trình Hamilton. Đây là một điều kiện đủ để một đồ thị có chu trình Hamilton

3. Đồ thị có hướng G liên thông mạnh và có n đỉnh. Nếu $deg^+(v) \ge n / 2$ và $deg^-(v) \ge n / 2$ với mọi đỉnh v thì G có chu trình Hamilton

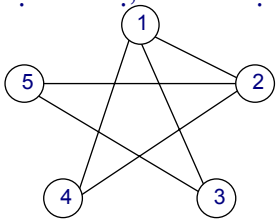
III. CÀI ĐẶT

Dưới đây ta sẽ cài đặt một chương trình liệt kê tất cả các chu trình Hamilton của một đơn đồ thị vô hướng bằng thuật toán quay lui. Lưu ý rằng cho tới nay, người ta vẫn **chưa tìm ra** một phương pháp nào thực sự hiệu quả hơn phương pháp quay lui để tìm dù chỉ một chu trình Hamilton cũng như đường đi Hamilton trong trường hợp đồ thị tổng quát.

Dữ liệu về đồ thị ta cho nhập từ file văn bản HAMILTON.INP. Trong đó:

- Dòng 1 ghi số đỉnh n và số cạnh m của đồ thị cách nhau 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng hai số nguyên dương u, v cách nhau 1 dấu cách, thể hiện u, v là hai đỉnh kề nhau trong đồ thị

Ví dụ: Đồ thị, file dữ liệu tương ứng và Output của chương trình



HAMILTON.INP	OUTPUT
5 6 1 3 2 4 3 5 4 1	1-->3-->5-->2-->4-->1 1-->4-->2-->5-->2-->1 2-->4-->1-->3-->5-->2 2-->5-->3-->1-->4-->2 ...

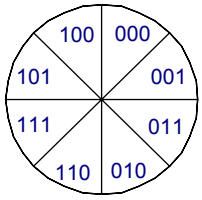
5 2 1 2	... 5-->3-->1-->4-->2-->5


```
program All_of_Hamilton_Circuits;
const
 max = 100;
var
 f: Text;
 A: array[1..max, 1..max] of Boolean; {Ma trận kề của đồ thị: A[u, v] = True ⇔(u, v) là cạnh}
 Free: array[1..max] of Boolean; {Mảng đánh dấu Free[v] = True nếu chưa đi qua đỉnh v}
 X: array[1..max] of Byte; {Chu trình Hamilton sẽ tìm là; X[1]→X[2]→ ...→X[n]→X[1] }
 n: Byte;

procedure Enter;
var
 DataFile: Text;
 i, u, v: Byte;
 m: Word;
begin
 FillChar(A, SizeOf(A), False); {Khởi tạo ma trận kề toàn False: đồ thị chưa có cạnh nào}
 Assign(DataFile, 'HAMILTON.INP'); Reset(DataFile);
 Readln(DataFile, n, m); {Đọc dòng đầu tiên của file ra số đỉnh và số cạnh}
 for i := 1 to m do
  begin
   Readln(DataFile, u, v);    {Đọc dòng thứ i trong số m dòng tiếp theo ra 2 số u, v}
   A[u, v] := True;           {Đặt phần tử tương ứng trong ma trận kề là True}
   A[v, u] := True;           {Đồ thị vô hướng nên A[v, u] phải bằng A[u, v]}
  end;
 Close(DataFile);
end;

procedure PrintResult; {In kết quả nếu tìm được chu trình Hamilton X[1]→X[2]→...→X[n]→X[1]}
var
 i: Byte;
begin
 for i := 1 to n do Write(X[i], '-->');
 Writeln(X[1]);
end;

procedure Try(i: Byte); {Thử các cách chọn đỉnh thứ i trong hành trình}
var
 j: Byte;
begin
 for j := 1 to n do      {Đỉnh thứ i (X[i]) có thể chọn trong những đỉnh}
  if Free[j] and A[x[i - 1], j] then  {kề với X[i - 1] và chưa bị đi qua }
   begin
    x[i] := j; {Thử một cách chọn X[i]}
    if i < n then {Nếu chưa thử chọn đến X[n]}
     begin
      Free[j] := False; {Đánh dấu đỉnh j là đã đi qua}
      Try(i + 1); {Để các bước thử kế tiếp không chọn phải đỉnh j nữa}
      Free[j] := True; {Sẽ thử phương án khác cho X[i] nên sẽ bỏ đánh dấu đỉnh vừa thử}
     end
    else {Nếu đã thử chọn đến X[n]}
     if A[j, X[1]] then PrintResult; {và nếu X[n] lại kề với X[1] thì ta có chu trình Hamilton}
   end;
end;

begin
 Enter;
 FillChar(Free, n, True); {Các đỉnh đều chưa bị đi qua}
 for x[1] := 1 to n do {Thử tất cả các phương án chọn đỉnh xuất phát}
```

```
  begin
   Free[x[1]] := False; {Đánh dấu đỉnh xuất phát}
   Try(2); {Thử các cách chọn đỉnh kế tiếp}
   Free[x[1]] := True; {Bỏ đánh dấu đỉnh xuất phát, để thử đỉnh khác làm đỉnh xuất phát}
  end;
end.
```


Bài tập:

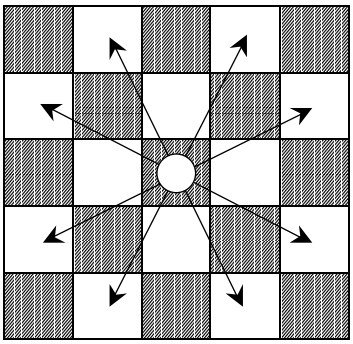
1. Lập chương trình nhập vào một đồ thị và chỉ ra đúng một chu trình Hamilton nếu có.
2. Lập chương trình nhập vào một đồ thị và chỉ ra đúng một đường đi Hamilton nếu có.
3. Trong đám cưới của Péc-xây và An-đrơ-nét có 2n hiệp sỹ. Mỗi hiệp sỹ có không quá n - 1 kẻ thù. Hãy giúp Ca-xi-ô-bê, mẹ của An-đrơ-nét xếp 2n hiệp sỹ ngồi quanh một bàn tròn sao cho không có hiệp sỹ nào phải ngồi cạnh kẻ thù của mình. Mỗi hiệp sỹ sẽ cho biết những kẻ thù của mình khi họ đến sân rồng.
4. Gray code: Một hình tròn được chia thành 2^n hình quạt đồng tâm. Hãy xếp tất cả các xâu nhị phân độ dài n vào các hình quạt, mỗi xâu vào một hình quạt sao cho bất cứ hai xâu nào ở hai hình quạt cạnh nhau đều chỉ khác nhau đúng 1 bít. Ví dụ với n = 3 ở hình vẽ bên



5. *__Thách đố:__ Bài toán mã đi tuần: Trên bàn cờ tổng quát kích thước n x n ô vuông (n chẵn và $6 \le n \le 20$). Trên một ô nào đó có đặt một quân mã. Quân mã đang ở ô (X_1, Y_1) có thể di chuyển sang ô (X_2, Y_2) nếu $|X_1-X_2|.|Y_1-Y_2| = 2$ (Xem hình vẽ). ***Hãy tìm một hành trình của quân mã từ ô xuất phát, đi qua tất cả các ô của bàn cờ, mỗi ô đúng 1 lần.***

Ví dụ:

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



Với n = 10; ô xuất phát (6, 5)									
18	71	**100**	43	20	69	86	45	22	25
97	42	19	70	99	44	21	24	87	46
72	17	98	95	68	85	88	63	26	23
41	96	73	84	81	94	67	90	47	50
16	83	80	93	74	89	64	49	62	27
79	40	35	82	**1**	76	91	66	51	48
36	15	78	75	92	65	2	61	28	53
39	12	37	34	77	60	57	52	3	6
14	33	10	59	56	31	8	5	54	29
11	38	13	32	9	58	55	30	7	4

Gợi ý: Nếu coi các ô của bàn cờ là các đỉnh của đồ thị và các cạnh là nối giữa hai đỉnh tương ứng với hai ô mã giao chân thì dễ thấy rằng hành trình của quân mã cần tìm sẽ là một đường đi Hamilton. Ta có thể xây dựng hành trình bằng thuật toán quay lui kết hợp với phương pháp duyệt ưu tiên Warnsdorff: Nếu gọi deg(x, y) là số ô kề với ô (x, y) và chưa đi qua (kề ở đây theo nghĩa đỉnh kề chứ không phải là ô kề cạnh) thì từ một ô ta sẽ **không thử xét lần lượt các hướng đi** có thể, mà ta sẽ **ưu tiên thử hướng đi tới ô có deg nhỏ nhất trước**. Trong **trường hợp có tồn tại đường đi**, phương pháp này hoạt động với tốc độ tuyệt vời: Với mọi n chẵn trong khoảng từ 6 tới 18, với mọi vị trí ô xuất phát, trung bình thời gian tính từ lúc bắt đầu tới lúc tìm ra một nghiệm < 1 giây. Tuy nhiên trong **trường hợp n lẻ, có lúc không tồn tại đường đi**, do phải duyệt hết mọi khả năng nên thời

gian thực thi lại hết sức tồi tệ. (Có xét ưu tiên như trên hay xét thứ tự như trước kia thì cũng vậy thôi. Không tin cứ thử với n lẻ: 5, 7, 9 ... và ô xuất phát (1, 2), sau đó ngồi xem máy tính toát mồ hôi).

§7. BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT

I. ĐỒ THỊ CÓ TRỌNG SỐ

Đồ thị mà mỗi cạnh của nó được gán cho tương ứng với một số (nguyên hoặc thực) được gọi là đồ thị có trọng số. Số gán cho mỗi cạnh của đồ thị được gọi là trọng số của cạnh. Tương tự như đồ thị không trọng số, có nhiều cách biểu diễn đồ thị có trọng số trong máy tính. Đối với đơn đồ thị thì cách dễ dùng nhất là sử dụng ma trận trọng số:

Giả sử đồ thị G = (V, E) có n đỉnh. Ta sẽ dựng ma trận vuông C kích thước n x n. Ở đây

- Nếu (u, v) ∈ E thì C[u, v] = trọng số của cạnh (u, v)
- Nếu (u, v) ∉ E thì tuỳ theo trường hợp cụ thể, C[u, v] được gán một giá trị nào đó để có thể nhận biết được (u, v) không phải là cạnh (Chẳng hạn có thể gán bằng +∞, hay bằng 0, bằng -∞ v.v...)
- Quy ước c[v, v] = 0 với mọi đỉnh v.

Đường đi, chu trình trong đồ thị có trọng số cũng được định nghĩa giống như trong trường hợp không trọng số, chỉ có khác là độ dài đường đi không phải tính bằng số cạnh đi qua, mà được tính bằng tổng trọng số của các cạnh đi qua.

II. BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT

Trong các ứng dụng thực tế, chẳng hạn trong mạng lưới giao thông đường bộ, đường thuỷ hoặc đường không. Người ta không chỉ quan tâm đến việc tìm đường đi giữa hai địa điểm mà còn phải lựa chọn một hành trình tiết kiệm nhất (theo tiêu chuẩn không gian, thời gian hay chi phí). Khi đó phát sinh yêu cầu tìm đường đi ngắn nhất giữa hai đỉnh của đồ thị. Bài toán đó phát biểu dưới dạng tổng quát như sau: Cho đồ thị có trọng số G = (V, E), hãy tìm một đường đi ngắn nhất từ đỉnh xuất phát S ∈ V đến đỉnh đích F ∈ V. Độ dài của đường đi này ta sẽ ký hiệu là d(S, F) và gọi là ***khoảng cách*** từ S đến F. Nếu như không tồn tại đường đi từ S tới F thì ta sẽ đặt khoảng cách đó = +∞.

- Nếu như đồ thị có chu trình âm (chu trình với độ dài âm) thì khoảng cách giữa một số cặp đỉnh nào đó có thể không xác định, bởi vì bằng cách đi vòng theo chu trình này một số lần đủ lớn, ta có thể chỉ ra đường đi giữa hai đỉnh nào đó trong chu trình này nhỏ hơn bất kỳ một số cho trước nào. Trong trường hợp như vậy, có thể đặt vấn đề tìm **đường đi cơ bản** (đường đi không có đỉnh lặp lại) ngắn nhất. Vấn đề đó là một vấn đề hết sức phức tạp mà ta sẽ không bàn tới ở đây.
- Nếu như đồ thị không có chu trình âm thì ta có thể chứng minh được rằng một trong những đường đi ngắn nhất là đường đi cơ bản. Và nếu như biết được khoảng cách từ S tới tất cả những đỉnh khác thì đường đi ngắn nhất từ S tới F có thể tìm được một cách dễ dàng qua thuật toán sau:

Gọi c[u, v] là trọng số của cạnh (u, v). Qui ước c[v, v] = 0 với mọi v ∈ V và c[u, v] = +∞ nếu như (u, v) ∉ E. Đặt d(S, v) là khoảng cách từ S tới v. Để tìm đường đi từ S tới F, ta có thể nhận thấy rằng luôn tồn tại đỉnh $F_1 \neq F$ sao cho:

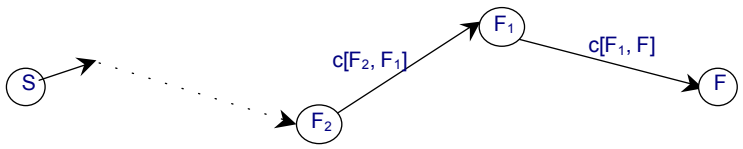
$$d(S, F) = d(S, F_1) + c[F_1, F]$$

(Độ dài đường đi ngắn nhất S->F = Độ dài đường đi ngắn nhất S->F_1 + Chi phí đi từ F_1 tới F)

Đỉnh F_1 đó là đỉnh liền trước F trong đường đi ngắn nhất từ S tới F. Nếu $F_1 \equiv S$ thì đường đi ngắn nhất là đường đi trực tiếp theo cung (S, F). Nếu không thì vấn đề trở thành tìm đường đi ngắn nhất từ S tới F_1. Và ta lại tìm được một đỉnh F_2 khác F và F_1 để:

$$d(S, F_1) = d(S, F_2) + c[F_2, F_1].$$

Cứ tiếp tục như vậy, sau một số hữu hạn bước, ta suy ra rằng dãy F, F_1, F_2, ... không chứa đỉnh lặp lại và kết thúc ở S. Lật ngược thứ tự dãy cho ta đường đi ngắn nhất từ S tới F.



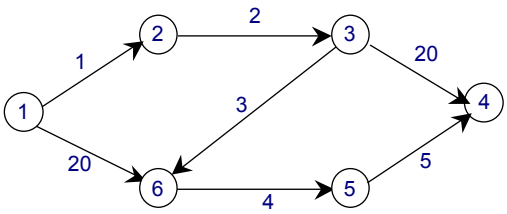
Tuy nhiên, trong đa số trường hợp, người ta không sử dụng phương pháp này mà sẽ kết hợp lưu vết đường đi ngay trong quá trình tìm kiếm.

Dưới đây ta sẽ xét một số thuật toán tìm đường đi ngắn nhất từ đỉnh S tới đỉnh F trên đơn đồ thị có hướng G = (V, E) có n đỉnh và m cung. Trong trường hợp đơn đồ thị vô hướng với trọng số không âm, bài toán tìm đường đi ngắn nhất có thể dẫn về bài toán trên đồ thị có hướng bằng cách thay mỗi cạnh của nó bằng hai cung có hướng ngược chiều nhau.

Dữ liệu về đồ thị được nhập từ file văn bản MINPATH.INP.

- Dòng 1: Ghi hai số đỉnh n và số cung m của đồ thị cách nhau 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng ba số u, v, c[u, v] cách nhau 1 dấu cách, thể hiện (u, v) là một cung ∈ E và trọng số của cung đó là c[u, v]

Riêng đỉnh xuất phát S và đỉnh đích F, vì đối với một đồ thị có thể có nhiều yêu cầu tìm đường đi ngắn nhất, nên ta sẽ cho nhập S và F từ bàn phím, bởi việc nhập đó cũng không mất nhiều thời gian và người sử dụng có thể đưa vào lần lượt từng yêu cầu tìm đường đi ngắn nhất cho tới khi hết yêu cầu. Ví dụ:



Đồ thị và file dữ liệu tương ứng

MINPATH.INP

```
6 7
1 2 1
1 6 20
2 3 2
3 4 20
3 6 3
5 4 5
6 5 4
```

Input/ Output của chương trình đối với đồ thị trên có thể như sau:

```
S, F = 1 4
Distance from 1 to 4: 15.000
4<--5<--6<--3<--2<--1
Do you want to continue ? Y/N: Y

S, F = 4 1
Not found any path from 4 to 1
Do you want to continue ? Y/N: Y

S, F = 3 4
Distance from 3 to 4: 12.000
4<--5<--6<--3
Do you want to continue ? Y/N: N
```

III. TRƯỜNG HỢP ĐỒ THỊ KHÔNG CÓ CHU TRÌNH ÂM - THUẬT TOÁN FORD-BELLMAN

Thuật toán Ford-Bellman có thể phát biểu rất đơn giản: ***Với đỉnh xuất phát S. Gọi d[v] là khoảng cách từ S tới v. Ban đầu d[S] được khởi gán bằng 0 còn các d[v] với v ≠ S được khởi gán bằng +∞. Sau đó ta tối ưu hoá dần các d[v] như sau: Xét mọi cặp đỉnh u, v của đồ thị, nếu có một cặp đỉnh u, v mà d[v] > d[u] + c[u, v] thì ta đặt lại d[v] := d[u] + c[u, v]***. Tức là nếu độ dài đường đi từ S tới v lại **lớn hơn** tổng độ dài đường đi từ S tới u cộng với chi phí đi từ u tới v thì ta sẽ huỷ bỏ đường đi từ S tới v đang có và coi đường đi từ S tới v chính là đường đi từ S tới u sau đó đi tiếp từ u tới v. Chú ý rằng ta đặt c[u, v] = +∞ nếu (u, v) không là cung. Thuật toán sẽ kết thúc khi không thể tối ưu thêm bất kỳ một nhãn d[v] nào nữa.

```
program Shortest_Path_by_Ford_Bellman;
uses crt;
const
 max = 100;
 maxReal = 1E9;
```

```
var
 c: array[1..max, 1..max] of Real;
 d: array[1..max] of Real;
 Trace: array[1..max] of Byte;
 n, S, F: Byte;

procedure LoadGraph;
var
 f: Text;
 i, m: Integer;
 u, v: Byte;
begin
 Assign(f, 'MINPATH.INP'); Reset(f);
 Readln(f, n, m);
 for u := 1 to n do
  for v := 1 to n do
   if u = v then c[u, v] := 0 else c[u, v] := maxReal;
 for i := 1 to m do Readln(f, u, v, c[u, v]);
 Close(f);
end;

procedure Init; {Nhập S, F và khởi gán giá trị mảng d}
var
 i: Byte;
begin
 Write('S, F = '); Readln(S, F);
 for i := 1 to n do d[i] := maxReal;
 d[S] := 0;
end;

procedure Ford_Bellman; {Thuật toán Ford_Bellman}
var
 Stop: Boolean;
 u, v: Byte;
begin
 repeat
  Stop := True;
  for u := 1 to n do
   for v := 1 to n do
    if d[v] > d[u] + c[u, v] then {Nếu tìm thấy một cặp (u, v) thoả mãn bất đẳng thức này}
     begin
      d[v] := d[u] + c[u, v]; {Thì tối ưu hoá đường đi từ S tới v}
      Trace[v] := u; {Ghi vết đường đi: Đỉnh liền trước v trong đường đi S -> v là u}
      Stop := False; {Báo hiệu phải lặp tiếp}
     end;
 until Stop; {Đặt cơ sở quy hoạch động, có thể chứng minh vòng lặp này sẽ lặp không quá n lần}
end;

procedure PrintResult;
begin
 if d[F] = maxReal then      {d[F] vẫn là +∞ thì không tồn tại đường đi}
  Writeln('Not found any path from ', S, ' to ', F)
 else          {Nếu không thì d[F] là độ dài đường đi ngắn nhất từ S->F. Truy vết tìm đường đi}
  begin
   Writeln('Distance from ', S, ' to ', F, ': ', d[F]:1:3);
   while F <> S do
    begin
     Write(F, '<--');
     F := Trace[F];
    end;
   Writeln(S);
  end;
```

```
end;

function Query_Answer: Char; {Cho giá trị 'Y' hay 'N' tuỳ theo người dùng có muốn tiếp hay không}
var
 ch: Char;
begin
 repeat
  Write('Do you want to continue ? Y/N: ');
  ch := Upcase(Readkey);
  Writeln(ch);
 until ch in ['Y', 'N'];
 Query_Answer := ch;
 Writeln;
end;

begin
 LoadGraph;
 repeat
  Init;
  Ford_Bellman;
  PrintResult;
 until Query_Answer = 'N';
end.
```


IV. TRƯỜNG HỢP TRỌNG SỐ TRÊN CÁC CUNG KHÔNG ÂM - THUẬT TOÁN DIJKSTRA

Trong trường hợp trọng số trên các cung không âm, thuật toán do Dijkstra đề xuất dưới đây hoạt động hiệu quả hơn nhiều so với thuật toán Ford-Bellman. Ta hãy xem trong trường hợp này, thuật toán Ford-Bellman thiếu hiệu quả ở chỗ nào:

Với đỉnh $v \in V$, Gọi d[v] là độ dài đường đi ngắn nhất từ S tới v. Thuật toán Ford-Bellman khởi tạo $d[S] = 0$ và các $d[v] = +\infty$ với $v \neq S$. Sau đó tối ưu hoá dần các nhãn d[v] bằng cách sửa nhãn theo công thức: $d[v] := \min(d[v], d[u] + c[u, v])$ với $\forall u, v \in V$. Như vậy nếu như ta dùng đỉnh u sửa nhãn đỉnh v, sau đó nếu ta lại tối ưu được d[u] thêm nữa thì ta cũng phải sửa lại nhãn d[v] dẫn tới việc d[v] có thể phải chỉnh đi chỉnh lại rất nhiều lần. Vậy nên chăng, tại mỗi bước **không phải ta xét mọi cặp đỉnh (u, v)** để dùng đỉnh u sửa nhãn đỉnh v mà sẽ **chọn đỉnh u là đỉnh mà không thể tối ưu nhãn d[u]** thêm được nữa.

Thuật toán Dijkstra (E.Dijkstra - 1959) có thể mô tả như sau:

Bước 1: Khởi tạo

Với đỉnh $v \in V$, gọi nhãn d[v] là độ dài đường đi ngắn nhất từ S tới v. Ta sẽ tính các d[v]. Ban đầu $d[S] = 0$ và $d[v] = +\infty$ với $v \neq S$. Nhãn của mỗi đỉnh có hai trạng thái tự do hay cố định, nhãn tự do có nghĩa là có thể còn tối ưu hơn được nữa và nhãn cố định tức là d[v] đã bằng độ dài đường đi ngắn nhất từ S tới v nên không thể tối ưu thêm. Để làm điều này ta có thể sử dụng một mảng đánh dấu: Free[v] = TRUE hay FALSE tuỳ theo d[v] tự do hay cố định. Ban đầu các nhãn đều tự do.

Vậy $d[S] := 0$; $d[v] := +\infty$ với $v \neq S$; và Free[v] := True với $\forall v \in V$.

Bước 2: Lặp

Bước lặp gồm có hai thao tác:

1. **Cố định nhãn: Chọn trong các đỉnh có nhãn tự do, lấy ra đỉnh u là đỉnh có d[u] nhỏ nhất, và cố định nhãn đỉnh u.**
2. **Sửa nhãn: Dùng đỉnh u, xét tất cả những đỉnh v và sửa lại các d[v] theo công thức:**

$$d[v] := \min(d[v], d[u] + c[u, v])$$

Bước lặp sẽ kết thúc khi mà đỉnh đích F được cố định nhãn (tìm được đường đi ngắn nhất từ S tới F); hoặc tại thao tác cố định nhãn, tất cả các đỉnh tự do đều có nhãn là $+\infty$ (không tồn tại đường đi).

Có thể đặt câu hỏi, ở thao tác 1, tại sao đỉnh u như vậy được cố định nhãn, giả sử d[u] còn có thể tối ưu thêm được nữa thì tất phải có một đỉnh t mang nhãn tự do sao cho $d[u] > d[t] + c[t, u]$. Do trọng

số c[t, u] không âm nên d[u] > d[t], trái với cách chọn d[u] là nhỏ nhất. Tất nhiên trong lần lặp đầu tiên thì S là đỉnh được cố định nhãn do d[S] = 0.

Bước 3: Kết hợp với việc lưu vết đường đi trên từng bước sửa nhãn, thông báo đường đi ngắn nhất tìm được hoặc cho biết không tồn tại đường đi (d[F] = +∞).


```
program Shortest_Path_by_Dijkstra;
uses crt;
const
 max = 100;
 maxReal = 1E9;
var
 c: array[1..max, 1..max] of Real;
 d: array[1..max] of Real;
 Trace: array[1..max] of Byte;
 Free: array[1..max] of Boolean; {Đánh dấu xem đỉnh có nhãn tự do hay cố định}
 n, S, F: Byte;

(*procedure LoadGraph; Không khác gì thuật toán Ford-Bellman ở trên*)

procedure Init;
var
 i: Byte;
begin
 Write('S, F = '); Readln(S, F);
 for i := 1 to n do d[i] := maxReal;
 d[S] := 0;
 FillChar(Free, n, True); {Khởi tạo các đỉnh đều có nhãn tự do}
end;

procedure Dijkstra; {Thuật toán Dijkstra}
var
 i, u, v: Byte;
 min: Real;
begin
 repeat
  {Cố định nhãn, chọn u có d[u] nhỏ nhất trong số các đỉnh có nhãn tự do < +∞}
  u := 0; min := maxReal;
  for i := 1 to n do
   if Free[i] and (d[i] < min) then
    begin
     min := d[i];
     u := i;
    end;
  if (u = 0) or (u = F) then Break; {Nếu không chọn được u hoặc u = F thì dừng ngay}
  Free[u] := False; {Cố định nhãn đỉnh u}
 {Sửa nhãn, dùng d[u] tối ưu lại các d[v]}
  for v := 1 to n do
   if Free[v] and (d[v] > d[u] + c[u, v]) then
    begin
     d[v] := d[u] + c[u, v];
     Trace[v] := u;
    end;
 until False;
end;

(*procedure PrintResult;Không khác gì trên*)

(*function Query_Answer: Char; Không khác gì trên*)

begin
 LoadGraph;
 repeat
```

```
  Init;
  Dijkstra;
  PrintResult;
 until Query_Answer = 'N';
end.
```


V. TRƯỜNG HỢP ĐỒ THỊ KHÔNG CÓ CHU TRÌNH - THỨ TỰ TÔPÔ

Ta có định lý sau: Giả sử G = (V, E) là đồ thị không có chu trình (có hướng - tất nhiên). Khi đó các đỉnh của nó có thể đánh số sao cho mỗi cung của nó chỉ nối từ đỉnh có chỉ số nhỏ hơn đến đỉnh có chỉ số lớn hơn.

Thuật toán đánh số lại các đỉnh của đồ thị có thể mô tả như sau:
Trước hết ta chọn một đỉnh không có cung đi vào và đánh chỉ số 1 cho đỉnh đó. Sau đó xoá bỏ đỉnh này cùng với tất cả những cung từ u đi ra, ta được một đồ thị mới cũng không có chu trình, và lại đánh chỉ số 2 cho một đỉnh v nào đó không có cung đi vào, rồi lại xoá đỉnh v cùng với các cung từ v đi ra ... Thuật toán sẽ kết thúc nếu như hoặc ta đã đánh chỉ số được hết các đỉnh, hoặc tất cả các đỉnh còn lại đều có cung đi vào. Trong trường hợp tất cả các đỉnh còn lại đều có cung đi vào thì sẽ tồn tại chu trình trong đồ thị và khẳng định thuật toán tìm đường đi ngắn nhất trong mục này không áp dụng được.
Nếu các đỉnh được đánh số sao cho mỗi cung phải nối từ một đỉnh tới một đỉnh khác mang chỉ số lớn hơn thì thuật toán tìm đường đi ngắn nhất có thể mô tả rất đơn giản:
Gọi d[v] là độ dài đường đi ngắn nhất từ S tới v. Khởi tạo d[S] = 0 và d[v] = +∞ với mọi v ≠ S. Ta sẽ tính các d[v] như sau:

```
for u := 1 to n - 1 do
 for v := u + 1 to n do
   d[v] := min(d[v], d[u] + c[u, v]);
```

(Giả thiết rằng c[u, v] = +∞ nếu như (u, v) không là cung).
Tức là dùng đỉnh u, tối ưu nhãn d[v] của những đỉnh v nối từ u, với u được xét lần lượt từ 1 tới n - 1. Có thể làm tốt hơn nữa bằng cách chỉ cần cho u chạy từ đỉnh xuất phát S tới đỉnh kết thúc F. Bởi hễ **u chạy tới đâu thì nhãn d[u] là không thể cực tiểu hoá thêm nữa.**

```
program Critical_Path; {Tìm đường đi ngắn nhất bằng thuật toán dùng đỉnh "trước" gán nhãn đỉnh "sau"}
uses crt;
const
 max = 100;
 maxReal = 1E9;
var
 c: array[1..max, 1..max] of Real;
 Index: array[1..max] of Byte; {Các đỉnh được đánh chỉ số lại thì Index[i] là chỉ số cũ của đỉnh i}
 d: array[1..max] of Real;
 Trace: array[1..max] of Byte;
 n, S, F, Count: Byte;

(*procedure LoadGraph; Như ở thuật toán Ford-Bellman *)

procedure Number; { Thuật toán đánh số các đỉnh}
var
 Deg: array[1..max] of Byte;
 u, v: Byte;
 Stop: Boolean;
```

```
begin
 FillChar(Deg, n, 0); {Trước hết tính các deg[u] = bán bậc vào của u = số đỉnh v nối đượctới u}
 for u := 1 to n do
  for v := 1 to n do
   if (v <> u) and (c[v, u] < maxReal) then Inc(Deg[u]);
 Count := 0;
 repeat
  Stop := True;
  for u := 1 to n do
   if Deg[u] = 0 then {Tìm đỉnh u có bán bậc vào bằng 0, nếu thấy}
    begin
     Inc(Count);
     Index[Count] := u; {Đưa u vào mảng Index và đánh chỉ số mới cho u là Count}
     for v := 1 to n do {Sau đó giảm bán bậc vào của những đỉnh v nối từ u ⇔ Xoá u và những cung ra}
      if (u <> v) and (c[u, v] < maxReal) then Dec(Deg[v]);
     Deg[u] := 255; {Đặt lại Deg[u] = +∞ để lần sau không tìm lại nữa}
     Stop := False;
    end;
 until Stop; {Cho tới khi không tìm được đỉnh nào có deg = 0, count là số đỉnh đánh số được}
end;

(*procedure Init; Như ở thuật toán Ford-Bellman*)

procedure FindPath;
var
 i, j, u, v: Byte;
begin
 for i := 1 to n - 1 do
  for j := i + 1 to n do
   begin
    u := Index[i]; v := Index[j];{Index[i] là chỉ số cũ của đỉnh i, để tối ưu nhãn thì ta phải đổi}
    if d[v] > d[u] + c[u, v] then{chỉ số mới i, j thành chỉ số cũ u, v. Để không bị lệch với ma trận c}
     begin
      d[v] := d[u] + c[u, v];
      Trace[v] := u;
     end
   end;
end;

(*procedure PrintResult; Giống như trong thuật toán Ford-Bellman*)

(*function Query_Answer: Char; Giống như trong thuật toán Ford-Bellman*)

begin
 LoadGraph;
 Number;
 if Count < n then
  Writeln('Error: Circuit Exist')
 else
  repeat
   Init;
   FindPath;
   PrintResult;
  until Query_Answer = 'N';
end.
```

VI. ĐƯỜNG ĐI NGẮN NHẤT GIỮA MỌI CẶP ĐỈNH - THUẬT TOÁN FLOYD

Cho đồ thị có hướng, có trọng số G = (V, E) với n đỉnh và m cạnh. Bài toán đặt ra là hãy tính tất cả các d(u, v) là khoảng cách từ u tới v. Rõ ràng là ta có thể áp dụng thuật toán tìm đường đi ngắn nhất xuất phát từ một đỉnh với n khả năng chọn đỉnh xuất phát. Nhưng ta có cách làm gọn hơn nhiều,

cách làm này rất giống với thuật toán Warshall mà ta đã biết: Từ ma trận trọng số c, thuật toán Floyd tính lại các c[u, v] thành độ dài đường đi ngắn nhất từ u tới v:

Với mọi đỉnh k của đồ thị được xét theo thứ tự từ 1 tới n, xét mọi cặp đỉnh u, v. Cực tiểu hoá c[u, v] theo công thức:

c[u, v] := min(c[u, v], c[u, k] + c[k, v])

Tức là nếu như đường đi từ u tới v đang có lại dài hơn đường đi từ u tới k cộng với đường đi từ k tới v thì ta huỷ bỏ đường đi từ u tới v hiện thời và coi đường đi từ u tới v sẽ là nối của hai đường đi từ u tới k rồi từ k tới v. Chú ý rằng ta còn có việc lưu lại vết:

Lưu ý rằng cài đặt dưới đây chỉ chạy được với đồ thị có số đỉnh ≤ 90. Khéo hơn một chút, không cần dùng bảng vết, hoặc lưu mỗi dòng của ma trận trọng số dưới dạng mảng một chiều cấp phát động thì có thể chạy với đồ thị với số đỉnh lớn hơn.

```
program Shortest_Path_by_Floyd;
uses crt;
const
 max = 90;
 maxReal = 1E9;
var
 c: array[1..max, 1..max] of Real;
 Trace: array[1..max, 1..max] of Byte; {Trace[u, v] = đỉnh liền sau u trên đường từ u tới v}
 n, S, F: Byte;

(*procedure LoadGraph; Như trong thuật toán Ford-Bellman*)

procedure Init;
begin
 Write('S, F = '); Readln(S, F);
end;

procedure Floyd;
var
 k, u, v: Byte;
begin
 for u := 1 to n do          {Ban đầu khởi tạo đường đi ngắn nhất giữa ∀u, v là đường đi trực tiếp, }
  for v := 1 to n do Trace[u, v] := v; {tức là đỉnh liền sau u trong đường đi từ u tới v là v}
 for k := 1 to n do
  for u := 1 to n do
   for v := 1 to n do
    if c[u, v] > c[u, k] + c[k, v] then {Nếu đường đi từ u tới v phải vòng qua k}
     begin
      c[u, v] := c[u, k] + c[k, v]; {Tối ưu hoá c[u, v] theo c[u, k] và c[k, v]}
      Trace[u, v] := Trace[u, k]; {Đỉnh liền sau u trong đường u ->v là Trace[u, v] := Trace[u, k]}
     end;
end;

procedure PrintResult;
begin
 if c[S, F] = maxReal
  then Writeln('Not found any path from ', S, ' to ', F)
 else
  begin
   Writeln('Distance from ', S, ' to ', F, ': ', c[S, F]:1:3);
   repeat
    Write(S, '-->'); {In ra S}
    S := Trace[S, F]; {Truy tiếp đỉnh liền sau S trong đường đi ngắn nhất từ S tới F}
   until S = F;
   Writeln(F);
  end;
end;
```

```
(*function Query_Answer: Char; Như trong thuật toán Ford-Bellman*)

begin
 LoadGraph;
 Floyd;
 repeat
  Init;
  PrintResult;
 until Query_Answer = 'N';
end.
```


Khác biệt rõ ràng của thuật toán Floyd là khi người dùng nhập vào một cặp (S, F) mới, chương trình chỉ việc in kết quả chứ không phải thực hiện lại thuật toán Floyd nữa.

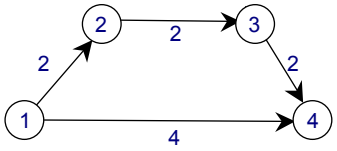
VII. NHẬN XÉT

Bài toán đường đi dài nhất trên đồ thị trong một số trường hợp có thể giải quyết bằng cách đổi dấu tất cả các cung rồi tìm đường đi ngắn nhất, nhưng hãy cẩn thận, có thể xảy ra trường hợp có chu trình âm.

Khác với một bài toán đại số hay hình học có nhiều cách giải thì chỉ cần nắm vững một cách cũng có thể coi là đạt yêu cầu, những thuật toán tìm đường đi ngắn nhất bộc lộ rất rõ ưu, nhược điểm trong từng trường hợp cụ thể (Ví dụ như số đỉnh của đồ thị quá lớn làm cho không thể biểu diễn bằng ma trận trọng số thì thuật toán Floyd sẽ gặp khó khăn, hay thuật toán Ford-Bellman làm việc khá chậm). Vì vậy yêu cầu trước tiên là phải hiểu bản chất và thành thạo trong việc cài đặt tất cả các thuật toán trên để có thể sử dụng chúng một cách uyển chuyển trong từng trường hợp cụ thể. Những bài tập sau đây cho ta thấy rõ điều đó.

Bài tập

1. Giải thích tại sao đối với đồ thị sau, cần tìm đường đi dài nhất từ đỉnh 1 tới đỉnh 4 lại không thể dùng thuật toán Dijkstra được, cứ thử áp dụng thuật toán Dijkstra theo từng bước xem sao:



2. Trên mặt phẳng cho n đường tròn ($n \le 2000$), đường tròn thứ i được cho bởi bộ ba số thực (X_i, Y_i, R_i), (X_i, Y_i) là toạ độ tâm và R_i là bán kính. Hai đường tròn có thể "bước" sang được nếu chúng có điểm chung, hai đường tròn có thể "nhảy" sang được nếu có thể chọn ra hai điểm trên hai đường tròn mà khoảng cách giữa hai điểm đó ≤ 1. Biết rằng một lần "bước" mất 1 công và một lần "nhảy" mất 1.5 công. Thứ nhất: hãy tìm phương án di chuyển giữa hai đường tròn S, F cho trước sao cho số công bỏ ra là ít nhất. Thứ hai: hãy tìm một dãy đường tròn lồng nhau dài nhất.

3. Cho một dãy n số nguyên A[1], A[2], ..., A[n] ($n \le 10000$; $1 \le A[i] \le 10000$). Hãy tìm một dãy con gồm nhiều nhất các phần tử của dãy đã cho mà tổng của hai phần tử liên tiếp là số nguyên tố.

4. Cho một mạng gồm n máy tính được đặt trên một nền phẳng với hệ toạ độ Decattes vuông góc Oxy. Máy tính thứ i được đặt ở toạ độ (X_i, Y_i). Cho m cáp mạng, cáp mạng thứ j được cho bởi cặp (p_j, q_j) tức là cáp mạng đó nối giữa hai máy tính p_j và q_j. Hai máy tính có thể chuyển thông tin cho nhau nếu như có cáp mạng nối chúng hoặc truyền qua một số máy trung gian. Vấn đề đặt ra là cho biết hai máy a, b. Hãy trả lời xem máy a có thể chuyển thông tin cho máy b không, nếu không thì cho biết cách nối thêm một số dây cáp mạng để có thể thực hiện đường truyền a->b và tốn ít chi phí nhất. Biết rằng chi phí đặt một dây cáp mạng nối hai máy bất kỳ tỉ lệ thuận với khoảng cách giữa chúng. ($n \le 300$, $m \le 1000$).

5. Một công trình lớn được chia làm n công đoạn đánh số 1, 2, ..., n. Công đoạn i phải thực hiện mất thời gian t[i]. Quan hệ giữa các công đoạn được cho bởi bảng a[i, j]: a[i, j] = TRUE ⇔ công đoạn j chỉ được bắt đầu khi mà công việc i đã xong. Hai công đoạn độc lập nhau có thể tiến hành song song, hãy bố trí lịch thực hiện các công đoạn sao cho thời gian hoàn thành cả công trình là sớm nhất, cho biết thời gian sớm nhất đó.

§8. BÀI TOÁN CÂY KHUNG NHỎ NHẤT

Cây là đồ thị **vô hướng**, **liên thông**, **không có chu trình đơn**. Đồ thị vô hướng không có chu trình đơn gọi là rừng (hợp của nhiều cây). Như vậy mỗi thành phần liên thông của rừng là một cây.
Khái niệm cây được sử dụng rộng rãi trong nhiều lĩnh vực khác nhau: Nghiên cứu cấu trúc các phân tử hữu cơ, xây dựng các thuật toán tổ chức thư mục, các thuật toán tìm kiếm, lưu trữ và nén dữ liệu...

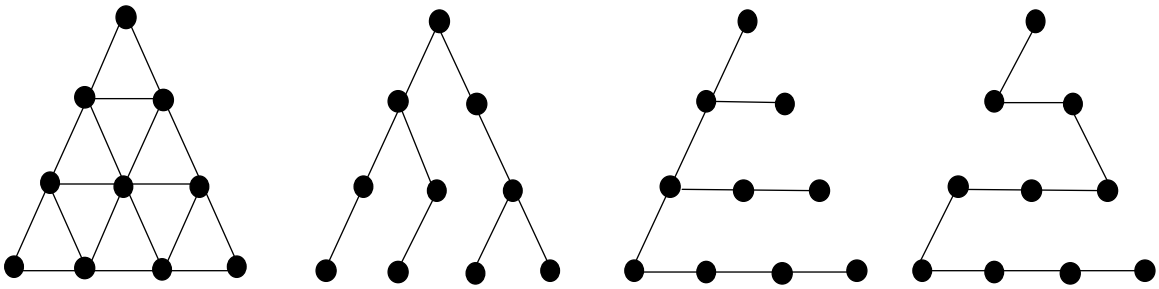
I. ĐỊNH LÝ

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

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

II. ĐỊNH NGHĨA

Giả sử G = (V, E) là đồ thị vô hướng liên thông. Cây T = (V, F) với F⊂E gọi là cây khung của đồ thị G. Tức là nếu như loại bỏ một số cạnh của G để được một cây thì cây đó gọi là cây khung (hay cây bao trùm của đồ thị).
Dễ thấy rằng với một đồ thị vô hướng liên thông có thể có nhiều cây khung.



Đồ thị G và một vài ví dụ về cây khung T1, T2, T3

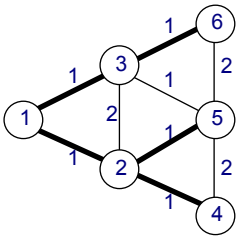
Định lý: Số cây khung của đồ thị đầy đủ K_n là n^{n-2}.

III. BÀI TOÁN CÂY KHUNG NHỎ NHẤT

Cho G = (V, E) là đồ thị vô hướng liên thông, với một cây khung T của G, ta gọi trọng số của cây T là tổng trọng số các cạnh trong T. Bài toán đặt ra là trong số các cây khung của G, chỉ ra cây khung có trọng số nhỏ nhất, cây khung như vậy được gọi là cây khung nhỏ nhất của đồ thị, và bài toán đó gọi là bài toán cây khung nhỏ nhất. Sau đây ta sẽ xét hai thuật toán thông dụng để giải bài toán cây khung nhỏ nhất.
Để tiết kiệm thời gian, dữ liệu về đồ thị sẽ nhập từ file văn bản MINTREE.INP:

- Dòng 1: Ghi hai số số đỉnh n và số cạnh m của đồ thị cách nhau 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng 3 số u, v, c[u, v] cách nhau 1 dấu cách thể hiện đồ thị có cạnh (u, v) và trọng số cạnh đó là c[u, v].

Ví dụ: Đơn đồ thị vô hướng có trọng số, file dữ liệu tương ứng và Output của chương trình:



MINTREE.INP	OUTPUT
6 9 1 2 1 1 3 1 2 4 1 2 5 1 3 5 1	Minimal spanning tree: (1, 2) (1, 3) (2, 4) (2, 5) (3, 6)

3 6 1 2 3 2 4 5 2 5 6 2	Weight = 5.000 Thử đọc và góp ý

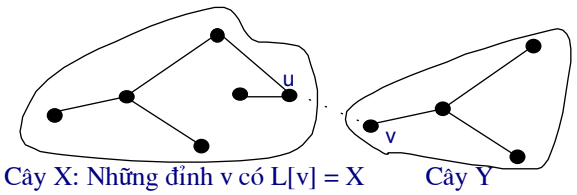
IV. THUẬT TOÁN KRUSKAL (JOSEPH KRUSKAL - 1956)

Thuật toán Kruskal phát biểu hình thức như sau: Với đồ thị vô hướng G = (V, E) có n đỉnh. Khởi tạo cây T ban đầu không có cạnh nào. Xét tất cả các cạnh của đồ thị **từ cạnh có trọng số nhỏ đến cạnh có trọng số lớn**, nếu việc thêm cạnh đó vào T **không tạo thành chu trình đơn** trong T thì **kết nạp thêm** cạnh đó vào T. Cứ làm như vậy cho tới khi:

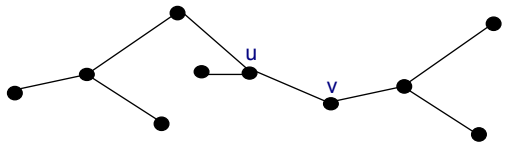
- Hoặc đã kết nạp được n - 1 cạnh vào trong T thì ta được T là cây khung nhỏ nhất
- Hoặc chưa kết nạp đủ n - 1 cạnh nhưng hễ cứ kết nạp thêm một cạnh bất kỳ trong số các cạnh còn lại thì sẽ tạo thành chu trình. Trong trường hợp này đồ thị G là không liên thông, việc tìm kiếm cây khung thất bại.

Như vậy có hai vấn đề quan trọng khi cài đặt thuật toán Kruskal:

1. Thứ nhất, làm thế nào để xét được các cạnh từ cạnh có trọng số nhỏ tới cạnh có trọng số lớn. Ta có thể thực hiện bằng cách sắp xếp danh sách cạnh theo thứ tự không giảm của trọng số, sau đó duyệt từ đầu tới cuối danh sách cạnh. Nên sử dụng các thuật toán sắp xếp như Sắp xếp chèn InsertionSort, Sắp xếp nhanh QuickSort, hay Sắp xếp kiểu vun đống HeapSort để đạt được tốc độ nhanh trong trường hợp số cạnh lớn.
2. Thứ hai, làm thế nào kiểm tra xem việc thêm một cạnh có tạo thành chu trình đơn trong T hay không. Để ý rằng các cạnh trong T ở các bước sẽ tạo thành một rừng (đồ thị không có chu trình đơn). Muốn thêm một cạnh (u, v) vào T mà không tạo thành chu trình đơn thì (u, v) phải nối hai cây khác nhau của rừng T, bởi nếu u, v thuộc cùng một cây thì sẽ tạo thành chu trình đơn trong cây đó (điều kiện tương đương thứ 6). Ta sử dụng kỹ thuật sau: Khởi tạo nhãn số hiệu cây: L[v] là số hiệu cây chứa đỉnh v. Ban đầu do T chưa có cạnh nào nên nó sẽ là một rừng gồm n cây: Đỉnh 1 thuộc cây 1, đỉnh 2 thuộc cây 2, ..., đỉnh n thuộc cây n. Tức là L[v] = v với mọi đỉnh v. Tại mỗi bước kết nạp cạnh, ta **chỉ được kết nạp cạnh (u, v) vào T nếu như L[u] ≠ L[v]**, và sau khi kết nạp cạnh đó rồi thì ta được một cây mới chứa hai cây L[u] và L[v]. Để ghi nhận cây mới này thì chỉ việc đặt những đỉnh đang thuộc cây L[v] bây giờ sẽ thuộc cây L[u], dĩ nhiên khi đó cây L[v] coi như bị huỷ:
   - a := L[v];
   - Xét mọi đỉnh k có L[k] = a đặt L[k] := L[u]



Cây X: Những đỉnh v có L[v] = X Cây Y



Cây X: sau khi nối hai cây X-Y

Cài đặt dưới đây sắp xếp danh sách cạnh bằng thuật toán sắp xếp đổi chỗ trực tiếp (BubbleSort) cho chương trình ngắn gọn và dễ hiểu, việc sử dụng các thuật toán QuickSort hay HeapSort trên danh sách móc nối cạnh coi như bài tập. Ngoài ra có thể tổ chức các cạnh như một cây nhị phân tìm kiếm thì việc duyệt cây nhị phân đó sẽ tự động duyệt các cạnh theo thứ tự không giảm.

```
program Minimal_Spanning_Tree_by_Kruskal;
const
 maxV = 100;
 maxE = (maxV + 1) * maxV div 2;
type
 TEdge = record {Cấu trúc dữ liệu chứa thông tin về 1 cạnh}
  u, v: Byte; {hai đỉnh}
  c: Real; {trọng số}
 end;
```

```
var
 e: array[1..maxE] of TEdge; {Danh sách cạnh}
 L: array[1..maxV] of Byte;
 n: Byte;
 m: Word;
 Connected: Boolean;

procedure LoadGraph;
var
 f: Text;
 i: Word;
begin
 Assign(f, 'MINTREE.INP'); Reset(f);
 Readln(f, n, m);
 for i := 1 to m do
  with e[i] do
   Readln(f, u, v, c);
 Close(f);
end;

procedure SortEdgeList;
var
 i, j: Word;
 t: TEdge;
begin {Thuật toán sắp xếp nổi bọt}
 for i := 1 to m - 1 do
  for j := i + 1 to m do
   if e[i].c > e[j].c then
    begin
     t := e[i]; e[i] := e[j]; e[j] := t;
    end;
end;

procedure Kruskal;
var
 i: Integer;
 Count, a, k: Byte;
begin
 for k := 1 to n do L[k] := k; {Khởi tạo rừng: cây k chỉ gồm mỗi đỉnh k}
 Count := 0;
 Connected := False; {Connected cho biết đồ thị ban đầu có liên thông không}
 for i := 1 to m do {Duyệt danh sách cạnh đã sắp xếp}
  if L[e[i].u] <> L[e[i].v] then {Nếu cạnh e[i] nối hai cây khác nhau}
   begin
    Inc(Count); {Đếm số cạnh đã kết nạp, sau đó hợp hai cây thành một cây}
    a := L[e[i].v]; {a là cây chứa đỉnh e[i].v}
    for k := 1 to n do
     if L[k] = a then L[k] := L[e[i].u]; {Nếu đỉnh k thuộc cây a thì giờ thuộc cây chứa e[i].u}
    if Count = n - 1 then {Nếu đã kết nạp đủ n - 1 cạnh thì thông báo đồ thị liên thông và dừng ngay}
     begin
      Connected := True; Break;
     end;
   end
  else e[i].u := 0; {Đây chẳng qua là đánh dấu cạnh e[i] rằng nếu thêm cạnh đó vào sẽ tạo chu trình đơn}
end;

procedure PrintResult;
var
 i: Word;
 Count: Byte;
 W: Real;
begin
 if not Connected then {Nếu đồ thị đã cho không liên thông thì thông báo thất bại}
```

```
  Writeln('Error: Graph is not connected')
 else
  begin
   Writeln('Minimal spanning tree: ');
   Count := 0;
   W := 0;
   for i := 1 to m do {Quét danh sách cạnh}
    with e[i] do
     begin
      if u <> 0 then {Nếu gặp cạnh e[i] không bị đánh dấu là sẽ tạo chu trình đơn}
       begin
        Writeln('(', u, ', ', v, ')'); {In ra cạnh e[i]}
        Inc(Count); {Đếm}
        W := W + c; {Tính trọng số}
       end;
      if Count = n - 1 then Break; {Nếu đếm đến n - 1 thì thôi}
     end;
   Writeln('Weight = ', W:1:3);
  end;
end;

begin
 LoadGraph;
 SortEdgeList;
 Kruskal;
 PrintResult;
end.
```

V. THUẬT TOÁN PRIM (ROBERT PRIM - 1957)

Thuật toán Kruskal hoạt động chậm trong trường hợp đồ thị dày (có nhiều cạnh). Trong trường hợp đó người ta thường sử dụng phương pháp lân cận gần nhất của Prim. Thuật toán đó có thể phát biểu hình thức như sau:

Đơn đồ thị vô hướng G = (V, E) có n đỉnh và m cạnh được cho bởi ma trận trong số C. Qui ước C[u, v] = +∞ nếu (u, v) không là cạnh. Xét cây T trong G và một đỉnh v, gọi **khoảng cách từ v tới T** là trọng số nhỏ nhất trong số các cạnh nối v với một đỉnh nào đó trong T:

$$d(v) = \min\{c[u, v] \mid u \in T\}$$

Ban đầu khởi tạo cây T chỉ gồm có mỗi đỉnh {1}. Sau đó cứ chọn trong số các đỉnh ngoài T ra một **đỉnh gần T nhất**, kết nạp đỉnh đó vào T đồng thời kết nạp luôn cả cạnh tạo ra khoảng cách gần nhất đó. Cứ làm như vậy cho tới khi:

- Hoặc đã kết nạp được tất cả n đỉnh thì ta có T là cây khung nhỏ nhất
- Hoặc chưa kết nạp được hết n đỉnh nhưng mọi đỉnh ngoài T đều có khoảng cách tới T là +∞. Khi đó đồ thị đã cho không liên thông, ta thông báo việc tìm cây khung thất bại.

Về mặt kỹ thuật cài đặt, ta có thể làm như sau:

Sử dụng mảng đánh dấu Free. Free[v] = TRUE nếu như đỉnh v chưa bị kết nạp vào T.

Gọi d[v] là khoảng cách từ v tới T. Ban đầu khởi tạo d[1] = 0 còn d[2] = d[3] = ... = d[n] = +∞. Tại mỗi bước chọn đỉnh đưa vào T, ta sẽ chọn đỉnh u nào ngoài T và có d[u] nhỏ nhất. Khi kết nạp u vào T rồi thì rõ ràng các nhãn d[v] sẽ thay đổi: $d[v]_{mới} := \min(d[v]_{cũ}, c[u, v])$. Vấn đề chỉ có vậy (chương trình rất giống thuật toán Dijkstra, chỉ khác ở công thức tối ưu nhãn)

```
program Minimal_Spanning_Tree_by_Prim;
const
 max = 100;
 maxReal = 1E9;
var
 c: array[1..max, 1..max] of Real;
 d: array[1..max] of Real;
 Free: array[1..max] of Boolean;
 Trace: array[1..max] of Byte; {Vết, Trace[v] là đỉnh nối với v trong cây khung nhỏ nhất}
```

```
 n: Byte;
 m: Word;
 Connected: Boolean;

procedure LoadGraph;
var
 f: Text;
 i: Word;
 u, v: Byte;
begin
 Assign(f, 'MINTREE.INP'); Reset(f);
 Readln(f, n, m);
 for u := 1 to n do
  for v := 1 to n do
   if u = v then c[u, v] := 0 else c[u, v] := maxReal; {Khởi tạo ma trận trọng số}
 for i := 1 to m do
  begin
   Readln(f, u, v, c[u, v]);
   c[v, u] := c[u, v]; {Đồ thị vô hướng nên c[v, u] = c[u, v]}
  end;
 Close(f);
end;

procedure Init;
var
 v: Byte;
begin
 d[1] := 0; {Đỉnh 1 có nhãn khoảng cách là 0}
 for v := 2 to n do d[v] := maxReal; {Các đỉnh khác có nhãn khoảng cách +∞}
 FillChar(Free, n, True); {Cây T ban đầu là rỗng}
end;

procedure Prim;
var
 k, i, u, v: Byte;
 min: Real;
begin
 Connected := True;
 for k := 1 to n do
  begin
   u := 0; min := maxReal; {Chọn đỉnh u chưa bị kết nạp có d[u] nhỏ nhất}
   for i := 1 to n do
    if Free[i] and (d[i] < min) then
     begin
      min := d[i];
      u := i;
     end;
   if u = 0 then {Nếu không chọn được u nào có d[u] < +∞ thì đồ thị không liên thông}
    begin
     Connected := False;
     Break;
    end;
   Free[u] := False; {Nếu chọn được thì đánh dấu u đã bị kết nạp, lặp lần 1 thì đĩ nhiên u = 1 bởi d[1] = 0}
   for v := 1 to n do
    if Free[v] and (d[v] > c[u, v]) then {Tính lại các nhãn khoảng cách d[v] với v chưa kết nạp}
     begin
      d[v] := c[u, v]; {Tối ưu nhãn d[v] theo công thức}
      Trace[v] := u; {Lưu vết, đỉnh nối với v cho khoảng cách ngắn nhất là u}
     end;
  end;
end;
```

```
procedure PrintResult;
var
 v: Byte;
 W: Real;
begin
 if not Connected then {Nếu đồ thị không liên thông thì thất bại}
  Writeln('Error: Graph is not connected')
 else
  begin
   Writeln('Minimal spanning tree: ');
   W := 0;
   for v := 2 to n do {Cây khung nhỏ nhất gồm những cạnh (v,Trace[v])}
    begin
     Writeln('(', Trace[v], ', ', v, ')');
     W := W + c[Trace[v], v];
    end;
   Writeln('Weight = ', W:1:3);
  end;
end;

begin
 LoadGraph;
 Init;
 Prim;
 PrintResult;
end.
```

Bài tập

1. Viết chương trình tạo đồ thị với số đỉnh ≤ 100, trọng số các cạnh là các số thực được sinh ngẫu nhiên. Ghi vào file dữ liệu MINTREE.INP đúng theo khuôn dạng quy định. So sánh kết quả làm việc của thuật toán Kruskal và thuật toán Prim về tính đúng đắn (Có lẽ chỉ cần so sánh trọng số của cây khung tìm được), và về tốc độ để thấy sự cần thiết phải cải tiến thủ tục sắp xếp của thuật toán Kruskal ở trên thành QuickSort hay HeapSort.
2. Trên một nền phẳng với hệ toạ độ Decattes vuông góc đặt n máy tính, máy tính thứ i được đặt ở toạ độ (X_i, Y_i). Cho phép nối thêm các dây cáp mạng nối giữa từng cặp máy tính. Chi phí nối một dây cáp mạng tỉ lệ thuận với khoảng cách giữa hai máy cần nối. Hãy tìm cách nối thêm các dây cáp mạng để cho các máy tính trong toàn mạng là liên thông và chi phí nối mạng là nhỏ nhất.
3. Tương tự như bài 2, nhưng ban đầu đã có sẵn một số cặp máy nối rồi, cần cho biết cách nối thêm ít chi phí nhất.
4. Hệ thống điện trong thành phố được cho bởi n trạm biến thế và các đường dây điện nối giữa các cặp trạm biến thế. Mỗi đường dây điện e có độ an toàn là p(e). ở đây $0 < p(e) \le 1$. Độ an toàn của cả lưới điện là tích độ an toàn trên các đường dây. Ví dụ như có một đường dây nguy hiểm: p(e) = 1% thì cho dù các đường dây khác là tuyệt đối an toàn (độ an toàn = 100%) thì độ an toàn của mạng cũng rất thấp (1%). Hãy tìm cách bỏ đi một số dây điện để cho các trạm biến thế vẫn liên thông và độ an toàn của mạng là lớn nhất có thể.

§9. BÀI TOÁN LUỒNG CỰC ĐẠI TRONG MẠNG

Ta gọi mạng là một đồ thị **có hướng** G = (V, E), trong đó có duy nhất một đỉnh A không có cung đi vào gọi là điểm phát, duy nhất một đỉnh B không có cung đi ra gọi là đỉnh thu và mỗi cung e = (u, v) ∈ E được gán với một số không âm c(e) = c(u, v) gọi là khả năng thông qua của cung đó. Để thuận tiện cho việc trình bày, ta qui ước rằng nếu không có cung (u, v) thì khả năng thông qua c(u, v) của nó được gán bằng 0.

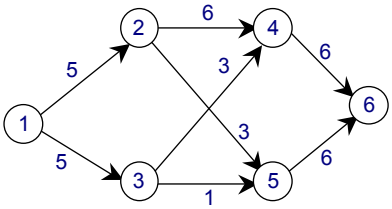
Nếu có mạng G = (V, E). Ta gọi luồng f trong mạng G là một phép gán cho mỗi cung e = (u, v) ∈ E một số thực không âm f(e) = f(u, v) gọi là luồng trên cung e, thoả mãn các điều kiện sau:

- Luồng trên mỗi cung không vượt quá khả năng thông qua của nó: $0 \le f(e) \le c(e)$
- Với mọi đỉnh v không trùng với đỉnh phát A và đỉnh thu B, tổng luồng trên các cung đi vào v bằng tổng luồng trên các cung đi ra khỏi v: $\sum_{u \in \Gamma^-(v)} f(u,v) = \sum_{w \in \Gamma^+(v)} f(v,w)$. Trong đó:

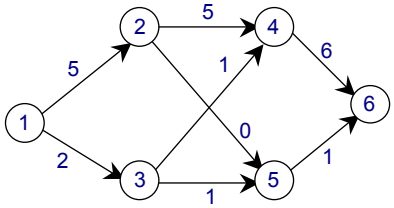
$\Gamma^-(v) = \{u \in V \mid (u, v) \in E\}$

$\Gamma^+(v) = \{w \in V \mid (v, w) \in E\}$

Giá trị của một luồng là tổng luồng trên các cung đi ra khỏi đỉnh phát = tổng luồng trên các cung đi vào đỉnh thu.



Mạng với các khả năng thông qua



và một luồng của nó với giá trị 7

I. BÀI TOÁN

Cho mạng G = (V, E). Hãy tìm luồng f* trong mạng với giá trị luồng lớn nhất. Luồng như vậy gọi là luồng cực đại trong mạng và bài toán này gọi là bài toán tìm luồng cực đại trên mạng.

II. Lát cắt, đường tăng luồng, định lý Ford- Fulkerson

1. Định nghĩa: Ta gọi lát cắt (X, Y) là một cách phân hoạch tập đỉnh V của mạng thành hai tập rời nhau X và Y, trong đó X chứa đỉnh phát và Y chứa đỉnh thu. Khả năng thông qua của lát cắt (X, Y) là tổng tất cả các khả năng thông qua của các cung (u, v) có u ∈ X và v ∈ Y. Lát cắt với khả năng thông qua nhỏ nhất gọi là lát cắt hẹp nhất.

2. Định lý Ford-Fulkerson: Giá trị luồng cực đại trên mạng đúng bằng khả năng thông qua của lát cắt hẹp nhất. Việc chứng minh định lý Ford- Fulkerson đã xây dựng được một thuật toán tìm luồng cực đại trên mạng:

Giả sử f là một luồng trong mạng G = (V, E). Từ mạng G = (V, E) ta xây dựng đồ thị có trọng số $G_f = (V, E_f)$ như sau:

Xét những cạnh e = (u, v) ∈ E (c(u, v) > 0):

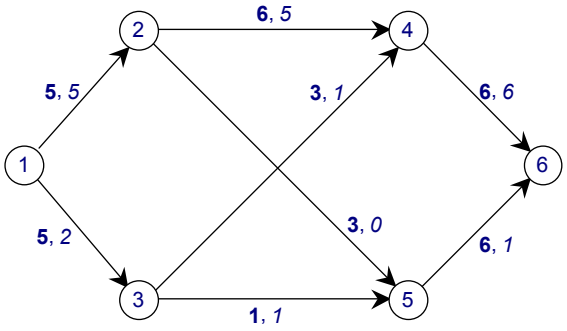
1. Nếu f(u, v) = 0 thì ta thêm cung (u, v) vào E_f với trọng số c(u, v)
2. Nếu f(u, v) = c(u, v) thì ta thêm cung (v, u) vào E_f với trọng số f(u, v)
3. Nếu 0 < f(u, v) < c(u, v) thì ta thêm cung (u, v) vào E_f với trọng số c(u, v) - f(u, v) và thêm cả cung (v, u) vào E_f với trọng số f(u, v)

Hay nói cách khác: Xét tất cả những cạnh e = (u, v) ∈ E,

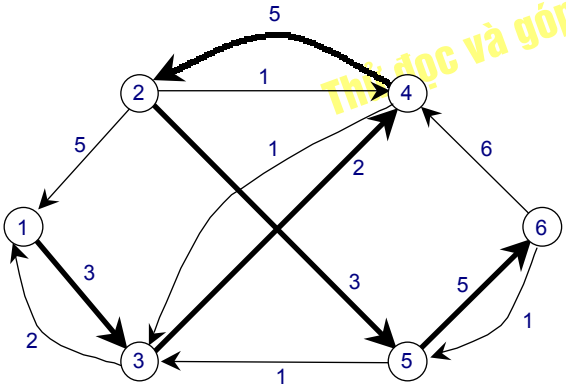
- Nếu f(u, v) < c(u, v) thì ta thêm cung (u, v) vào E_f với trọng số c(u, v) - f(u, v), cung đó gọi là **cung thuận**
- Xét tiếp nếu như f(u, v) > 0 thì ta thêm cung (v, u) vào E_f với trọng số f(u, v), cung đó gọi là **cung nghịch**.

Đồ thị G_f được gọi là **đồ thị tăng luồng.**

Ví dụ: Với mạng G sau: đỉnh phát 1, đỉnh thu 6.



Mạng G: cặp số ghi trên mỗi cung theo thứ tự là khả năng thông qua và luồng đang có



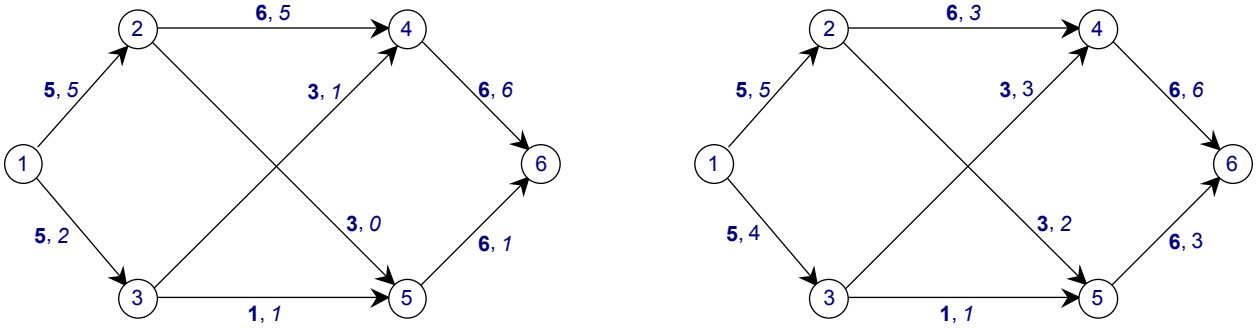
Đồ thị tăng luồng G_f tương ứng

Giả sử P là một đường đi cơ bản từ đỉnh phát A tới đỉnh thu B. Gọi Δ là giá trị nhỏ nhất của các trọng số của các cung trên đường đi P. Ta sẽ tăng giá trị của luồng f bằng cách đặt:

- **f(u, v)** := f(u, v) + Δ, nếu (u, v) là cung trong đường P và là cung thuận
- **f(v, u)** := f(v, u) - Δ, nếu (u, v) là cung trong đường P và là cung nghịch
- Còn luồng trên những cung khác giữ nguyên

Có thể kiểm tra luồng f mới xây dựng vẫn là luồng trong mạng và giá trị của luồng f mới được tăng thêm Δ so với giá trị luồng f cũ. Ta gọi thao tác biến đổi luồng như vậy là **tăng luồng dọc đường P**, đường đi cơ bản P từ A tới B được gọi là **đường tăng luồng.**

Ví dụ: với đồ thị tăng luồng G_f như trên, giả sử chọn đường đi (1, 3, 4, 2, 5, 6). Giá trị nhỏ nhất của trọng số trên các cung là 2, vậy thì ta sẽ tăng các giá trị f(1, 3), f(3, 4), f(2, 5), f(5, 6) lên 2, (do các cung đó là cung thuận) và giảm giá trị f(2, 4) đi 2 (do cung (4, 2) là cung nghịch). Được luồng mới mang giá trị 9.



Mạng G trước và sau khi tăng luồng

Đến đây ta có thể hình dung ra được thuật toán tìm luồng cực đại trên mạng: khởi tạo một luồng bất kỳ, sau đó cứ **tăng luồng dọc theo đường tăng luồng, cho tới khi không tìm được đường tăng luồng nữa**

Vậy các bước của thuật toán tìm luồng cực đại trên mạng có thể mô tả như sau:

Bước 1: Khởi tạo:

Một luồng bất kỳ trên mạng, chẳng hạn như luồng 0 (luồng trên các cung đều bằng 0), sau đó:

Bước 2: Lặp hai bước sau:

- Tìm đường tăng luồng P đối với luồng hiện có ≡ Tìm đường đi cơ bản từ A tới B trên đồ thị tăng luồng, nếu không tìm được đường tăng luồng thì bước lặp kết thúc.
- Tăng luồng dọc theo đường P

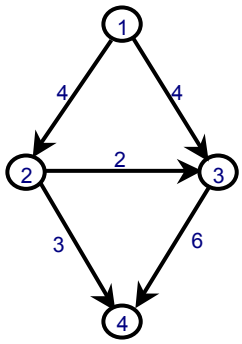
Bước 3: Thông báo giá trị luồng cực đại tìm được.

III. CÀI ĐẶT

Dữ liệu về mạng được nhập từ file văn bản MAXFLOW.INP. Trong đó:

- Dòng 1: Ghi số đỉnh n, số cạnh m của đồ thị, đỉnh phát A, đỉnh thu B theo đúng thứ tự cách nhau một dấu cách
- m dòng tiếp theo, mỗi dòng có dạng ba số u, v, c[u, v] cách nhau một dấu cách thể hiện có cung (u, v) trong mạng và khả năng thông qua của cung đó là c[u, v]

Chú ý rằng tại mỗi bước có nhiều phương án chọn đường tăng luồng, hai cách chọn khác nhau có thể cho hai luồng cực đại khác nhau, tuy nhiên về mặt giá trị thì tất cả các luồng xây dựng được theo cách trên sẽ có cùng giá trị cực đại.



MAXFLOW.INP	OUTPUT
4 5 1 4 1 2 4 1 3 4 2 3 2 2 4 3 3 4 6	f(1, 2) = 4.000 f(1, 3) = 4.000 f(2, 3) = 1.000 f(2, 4) = 3.000 f(3, 4) = 5.000 Max Flow: 8.000

Mạng G, file dữ liệu và Output tương ứng

Cài đặt chương trình tìm luồng cực đại dưới đây rất chân phương, từ ma trận những khả năng thông qua c và luồng f hiện có (khởi tạo f là luồng 0), nó xây dựng đồ thị tăng luồng Gf bằng cách xây dựng ma trận cf như sau:

- cf[u, v] = trọng số cung (u, v) trên đồ thị G_f nếu như (u, v) là cung thuận
- cf[u, v] = - trọng số cung (u, v) trên đồ thị G_f nếu như (u, v) là cung nghịch
- cf[u, v] = +∞ nếu như (u, v) không phải cung của G_f

cf gần giống như ma trận trọng số của G_f, chỉ có điều ta đổi dấu trọng số nếu như gặp cung nghịch. Câu hỏi đặt ra là nếu như mạng đã cho có những đường hai chiều (có cả cung (u, v) và cung (v, u) - điều này xảy ra rất nhiều trong mạng lưới giao thông) thì đồ thị tăng luồng rất có thể là đa đồ thị (giữa u, v có thể có nhiều cung từ u tới v). Ma trận cf cũng gặp nhược điểm như ma trận trọng số: **không thể biểu diễn được đa đồ thị**, tức là nếu như có nhiều cung nối từ u tới v trong đồ thị tăng luồng thì ta đành **chấp nhận bỏ bớt mà chỉ giữ lại một cung**. Rất may cho chúng ta là điều đó không làm sai lệch đi mục đích xây dựng đồ thị tăng luồng: chỉ là tìm một đường đi từ đỉnh phát A tới đỉnh thu B mà thôi, còn đường nào thì không quan trọng.

Sau đó chương trình tìm đường đi từ đỉnh phát A tới đỉnh thu B trên đồ thị tăng luồng bằng thuật toán tìm kiếm theo chiều rộng, nếu tìm được đường đi thì sẽ tăng luồng dọc theo đường tăng luồng...

```
program Max_Flow;
const
 max = 50;
 maxReal = 1E9;
var
 c, f, cf: array[1..max, 1..max] of Real; {c: khả năng thông, f: Luồng}
 Trace: array[1..max] of Byte;
 n, A, B: Byte;

procedure Enter; {Nhập mạng từ file}
var
 f: Text;
 m, i: Word;
 u, v: Byte;
begin
 FillChar(c, SizeOf(c), 0);
 Assign(f, 'MAXFLOW.INP'); Reset(f);
 Readln(f, n, m, A, B);
 for i := 1 to m do
  Readln(f, u, v, c[u, v]);
 Close(f);
```

```
end;

procedure CreateGf; {Tìm đồ thị tăng luồng, tức là xây dựng cf từ c và f}
var
 u, v: Byte;
begin
 for u := 1 to n do
  for v := 1 to n do cf[u, v] := maxReal;
 for u := 1 to n do
  for v := 1 to n do
   if c[u, v] > 0 then {Nếu u, v là cung trong mạng}
    begin
     if f[u, v] < c[u, v] then cf[u, v] := c[u, v] - f[u, v]; {Đặt cung thuận}
     if f[u, v] > 0 then cf[v, u] := -f[u, v]; {Đặt cung nghịch}
    end;
end;
{Thủ tục này tìm một đường đi từ A tới B bằng BFS, trả về TRUE nếu có đường, FALSE nếu không có đường}
function FindPath: Boolean;
var
 u, v: Byte;
 Queue: array[1..max] of Byte; {Hàng đợi dùng cho BFS}
 Free: array[1..max] of Boolean;
 First, Last: Byte;
begin
 FillChar(Free, SizeOf(Free), True);
 First := 1; Last := 1; Queue[1] := A; {Queue ← A}
 Free[A] := False; {đánh dấu A}
 repeat
  u := Queue[First]; Inc(First); {u ← Queue}
  for v := 1 to n do
   if Free[v] and (cf[u, v] <> maxReal) then {Xét v chưa đánh dấu kề với u}
    begin
     Trace[v] := u; {Lưu vết đường đi A → ... → u → v}
     if v = B then {v = B thì ta có đường đi từ A tới B, thoát thủ tục}
      begin
       FindPath := True; Exit;
      end;
     Free[v] := False; {đánh dấu v}
     Inc(Last);
     Queue[Last] := v; {Queue ← v}
    end;
 until First > Last; {Queue rỗng}
 FindPath := False; {ở trên không Exit được thì tức là không có đường}
end;

{Thủ tục tăng luồng dọc theo đường tăng luồng tìm được trong FindPath}
procedure IncFlow;
var
 IncValue: Real;
 u, v: Byte;
begin
 {Trước hết dò đường theo vết để tìm trọng số nhỏ nhất của các cung trên đường}
 IncValue := maxReal;
 v := B;
 while v <> A do
  begin
   u := Trace[v]; {Để ý rằng |cf[u, v]| là trọng số của cung (u, v) trên đồ thị tăng luồng}
   if Abs(cf[u, v]) < IncValue then IncValue := Abs(cf[u, v]);
   v:= u;
  end;
 {Dò lại đường lần thứ hai, lần này để tăng luồng}
 v := B;
```

```
 while v <> A do
  begin
   u := Trace[v];
   if cf[u, v] > 0 then f[u, v] := f[u, v] + IncValue {Nếu (u, v) là cung thuận trên Gf}
   else f[v, u] := f[v, u] - IncValue; {Nếu (u, v) là cung nghịch trên Gf}
   v := u;
  end;
end;

procedure PrintResult; {In luồng cực đại tìm được}
var
 u, v: Byte;
 m: Real;
begin
 m := 0;
 for u := 1 to n do
  for v := 1 to n do
   if c[u, v] > 0 then {Nếu có cung (u, v) trên mạng thì in ra giá trị luồng f gán cho cung đó}
    begin
     Writeln('f(', u, ', ', v, ') = ', f[u, v]:1:3);
     if u = A then m := m + f[A, v]; {Giá trị luồng cực đại = tổng luồng phát ra từ A}
    end;
 Writeln('Max Flow: ', m:1:3);
end;

begin
 Enter;
 FillChar(f, SizeOf(f), 0); {Khởi tạo luồng 0}
 repeat
  CreateGf; {Xây dựng đồ thị tăng luồng}
  if not FindPath then Break; {Tìm đường tăng luồng, nếu không thấy thì kết thúc bước lặp}
  IncFlow; {Tăng luồng dọc theo đường tăng luồng}
 until False;
 PrintResult;
end.
```


Bây giờ ta thử xem cách làm trên được ở chỗ nào và chưa hay ở chỗ nào ?
Trước hết thuật toán tìm đường bằng Breadth First Search là khá tốt, người ta đã chứng minh rằng nếu như đường tăng luồng được tìm bằng BFS sẽ làm giảm đáng kể số bước lặp tăng luồng so với DFS.
Nhưng có thể thấy rằng việc **xây dựng tường minh cả đồ thị G_f** thông qua việc xây dựng ma trận cf chỉ để làm mỗi một việc tìm đường là lãng phí, chỉ cần dựa vào ma trận khả năng thông qua c và luồng f hiện có là ta có thể biết được (u, v) có phải là cung trên đồ thị tăng luồng G_f hay không.
Thứ hai tại bước tăng luồng, ta phải dò lại hai lần đường đi, một lần để tìm trọng số nhỏ nhất của các cung trên đường, một lần để tăng luồng. Trong khi việc tìm trọng số nhỏ nhất của các cung trên đường có thể kết hợp làm ngay trong thủ tục tìm đường bằng cách sau:
Đặt Delta[v] là trọng số nhỏ nhất của các cung trên đường đi từ A tới v, khởi tạo Delta[A] = +∞.
Tại mỗi bước từ đỉnh u thăm đỉnh v trong BFS, thì Delta[v] có thể được tính bằng giá trị nhỏ nhất trong hai giá trị Delta[u] và trọng số cung (u, v) trên đồ thị tăng luồng. Khi tìm được đường đi từ A tới B thì Delta[B] cho ta trọng số nhỏ nhất của các cung trên đường tăng luồng.
Thứ ba, ngay trong bước tìm đường tăng luồng, ta có thể xác định ngay cung nào là cung thuận, cung nào là cung nghịch. Vì vậy khi từ đỉnh u thăm đỉnh v trong BFS, ta có thể vẫn lưu vết đường đi Trace[v] := u, nhưng sau đó sẽ đổi dấu Trace[v] nếu như (u, v) là cung nghịch.
Những cải tiến đó cho ta một cách cài đặt hiệu quả hơn, đó là:

IV. THUẬT TOÁN FORD- FULKERSON (L.R.FORD & D.R.FULKERSON - 1962)

Mỗi đỉnh v được gán nhãn (Trace[v], Delta[v]). Trong đó |Trace[v]| là đỉnh liền trước v trong đường đi từ A tới v, Trace[v] âm hay dương tuỳ theo (Trace[v], v) là cung nghịch hay cung thuận

trên đồ thị tăng luồng, Delta[v] là trọng số nhỏ nhất của các cung trên đường đi từ A tới v trên đồ thị tăng luồng.

Bước lặp sẽ tìm đường đi từ A tới B trên đồ thị tăng luồng đồng thời tính luôn các nhãn (Trace[v], Delta[v]). Quy ước rằng nếu Trace[B] = 0 tức là không tồn tại đường đi từ A tới B, sau đó tăng luồng dọc theo đường tăng luồng nếu tìm thấy.


```
program Max_Flow_by_Ford_Fulkerson;
const
 max = 50;
 maxReal = 1E9;
 var c, f: array[1..max, 1..max] of Real;
 Trace: array[1..max] of ShortInt;
 Delta: array[1..max] of Real;
 n, A, B: Byte;

(*procedure Enter; Như trên *)

function Min(X, Y: Real): Real;begin if X < Y then Min := X else Min := Y;end;function FindPath: Boolean;
var
 u, v: Byte;
 Queue: array[1..max] of Byte;
 First, Last: Byte;
begin
 FillChar(Trace, SizeOf(Trace), 0); {Trace[v] = 0 đồng nghĩa với v chưa đánh dấu}
 First := 1; Last := 1; Queue[1] := A;
 Trace[A] := n + 1; {Chỉ là để nó khác 0 mà thôi, số dương nào chẳng được}
 Delta[A] := maxReal; {Khởi tạo nhãn}
 repeat
  u := Queue[First]; Inc(First);
  for v := 1 to n do
   if Trace[v] = 0 then {Xét những đỉnh v chưa đánh dấu}
    begin
     if (c[u, v] > 0) and (f[u, v] < c[u, v]) then {Nếu (u, v) là cung thuận trên Gβ}
      begin
       Trace[v] := u; {Lưu vết, Trace[v] mang dấu dương}
       Delta[v] := min(Delta[u], c[u, v] - f[u, v]); {c[u, v] - f[u, v] là trọng số cung (u, v)}
      end
     else
      if (c[v, u] > 0) and (f[v, u] > 0) then {Nếu (u, v) là cung nghịch trên Gβ}
       begin
        Trace[v] := -u; {Lưu vết, Trace[v] mang dấu âm}
        Delta[v] := min(Delta[u], f[v, u]); {f[v, u] là trọng số cung nghịch (u, v) trên Gβ}
       end;
     if Trace[B] <> 0 then {Nếu đã đánh dấu được B thì việc tìm đường thành công, thoát ngay}
      begin
       FindPath := True; Exit;
      end;
     if Trace[v] <> 0 then {Trace[v] khác 0 tức là từ u có thể thăm v, thì đưa v vào Queue}
      begin
       Inc(Last);
       Queue[Last] := v;
      end;
    end;
 until First > Last; {Hàng đợi Queue rỗng}
 FindPath := False; {ở trên không Exit được tức là không có đường}
end;

procedure IncFlow; {Tăng luồng dọc đường tăng luồng}
var
 IncValue: Real;
```

```
 u, v: ShortInt;
begin
 IncValue := Delta[B]; {Nhãn Delta[B] chính là trọng số nhỏ nhất trên các cung của đường tăng luồng}
 v := B; {Truy vết đường đi, tăng luồng dọc theo đường đi}
 repeat
  u := Trace[v]; {Xét cung ( |u|, v) trên đường tăng luồng}
  if u > 0 then f[u, v] := f[u, v] + IncValue {( |u|, v) là cung thuận thì tăng f[ |u|, v]}
  else
   begin
    u := -u;
    f[v, u] := f[v, u] - IncValue; {( |u|, v) là cung nghịch thì giảm f[v, |u|]}
   end;
  v := u;
 until v = A;
end;

(*procedure PrintResult; Như trên*)

begin
 Enter;
 FillChar(f, SizeOf(f), 0);
 repeat
  if not FindPath then Break;
  IncFlow;
 until False;
 PrintResult;
end.
```


Định lý về luồng cực đại trong mạng và lát cắt hẹp nhất: Luồng cực đại trong mạng bằng khả năng thông qua của lát cắt hẹp nhất. Khi đã tìm được luồng cực đại thì theo thuật toán trên sẽ không có đường đi từ A tới B trên đồ thị tăng luồng. Nếu đặt tập X gồm những đỉnh đến được từ đỉnh phát A trên đồ thị tăng luồng (tất nhiên A∈X) và tập Y gồm những đỉnh còn lại (tất nhiên B∈Y) thì (X, Y) là lát cắt hẹp nhất đó.

Định lý về tính nguyên: Nếu tất cả các khả năng thông qua là số nguyên thì thuật toán trên luôn tìm được luồng cực đại với luồng trên cung là các số nguyên. Điều này có thể chứng minh rất dễ bởi ban đầu khởi tạo luồng 0 thì tức các luồng trên cung là nguyên. Mỗi lần tăng luồng lên một lượng bằng trọng số nhỏ nhất trên các cung của đường tăng luồng cũng là số nguyên nên cuối cùng luồng cực đại tất sẽ phải có luồng trên các cung là nguyên.

Bài tập:

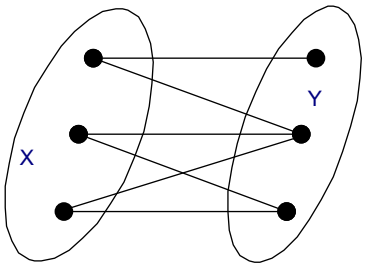
1. Cho một mạng gồm n đỉnh với p điểm phát A_1, A_2, ..., A_p và q điểm thu B_1, B_2, ..., B_q. Mỗi cung của mạng được gán khả năng thông qua là số nguyên. Các đỉnh phát chỉ có cung đi ra và các đỉnh thu chỉ có cung đi vào. Một luồng trên mạng này là một phép gán cho mỗi cung một số thực gọi là luồng trên cung đó không vượt quá khả năng thông qua và thoả mãn với mỗi đỉnh không phải đỉnh phát hay đỉnh thu thì tổng luồng đi vào bằng tổng luồng đi ra. Giá trị luồng bằng tổng luồng đi ra từ các đỉnh phát = tổng luồng đi vào các đỉnh thu. Hãy tìm luồng cực đại trên mạng.

2. Cho một mạng với đỉnh phát A và đỉnh thu B. Mỗi cung (u, v) được gán khả năng thông qua c(u, v). Mỗi đỉnh v được gán khả năng thông qua d[v]. Một luồng trên mạng được định nghĩa như trước và thêm điều kiện: tổng luồng đi vào đỉnh v không được vượt quá khả năng thông qua d[v] của đỉnh đó. Hãy tìm luồng cực đại trên mạng.

§10. BÀI TOÁN TÌM CẶP GHÉP CỰC ĐẠI TRÊN ĐỒ THỊ HAI PHÍA

I. ĐỒ THỊ PHÂN ĐÔI (BIPARTITE GRAPH)

Các tên gọi đồ thị phân đôi, đồ thị lưỡng phân, đồ thị hai phía, đồ thị đối sánh hai phần v.v... là để chỉ chung một dạng đơn đồ thị vô hướng G = (V, E) mà tập đỉnh của nó có thể chia làm hai tập con X, Y rời nhau sao cho bất kỳ cạnh nào của đồ thị cũng nối một đỉnh của X với một đỉnh thuộc Y. Khi đó người ta còn ký hiệu G là (X∪Y, E) và gọi một tập (chẳng hạn tập X) là **tập các đỉnh trái** và tập còn lại là **tập các đỉnh phải** của đồ thị phân đôi G. Các đỉnh thuộc X còn gọi là các X_đỉnh, các đỉnh thuộc Y gọi là các Y_đỉnh.



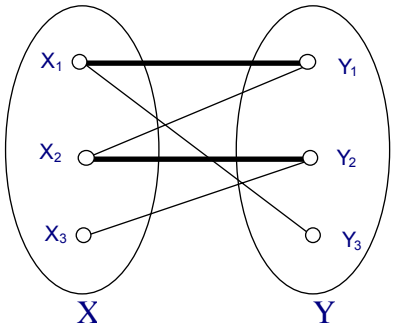
Để kiểm tra một đồ thị liên thông có phải là đồ thị phân đôi hay không, ta có thể áp dụng thuật toán sau:
Với một đỉnh v bất kỳ:
X := {v}; Y := ∅;
repeat
 Y := Y ∪ Kề(X);
 X := X ∪ Kề(Y);
until (X∩Y ≠ ∅) or (X và Y là tối đại - không bổ sung được nữa);
if X∩Y ≠ ∅ then < Không phải đồ thị hai phía >
else <Đây là đồ thị hai phía, X là tập các đỉnh trái: các đỉnh đến được từ v qua một số chẵn cạnh, Y là tập các đỉnh phải: các đỉnh đến được từ v qua một số lẻ cạnh>
Đồ thị phân đôi gặp rất nhiều mô hình trong thực tế. Chẳng hạn quan hệ hôn nhân giữa tập những người đàn ông và tập những người đàn bà, việc sinh viên chọn trường, thầy giáo chọn tiết dạy trong thời khoá biểu v.v... nhưng có lẽ quan hệ hôn nhân là trực quan nhất.

II. BÀI TOÁN GHÉP ĐÔI KHÔNG TRỌNG VÀ CÁC KHÁI NIỆM

Cho một đồ thị phân đôi G = (X∪Y, E) ở đây X là tập các đỉnh trái và Y là tập các đỉnh phải của G
Một bộ ghép (matching) của G là một tập hợp các cạnh của G đôi một không có đỉnh chung.
Bài toán ghép đôi (matching problem) là tìm một bộ ghép lớn nhất (nghĩa là có số cạnh lớn nhất) của G
Xét một bộ ghép M của G.

- Các đỉnh trong M gọi là các đỉnh đã ghép (matched vertices), các đỉnh khác là chưa ghép.
- Các cạnh trong M gọi là các cạnh đã ghép, các cạnh khác là chưa ghép
- Một đường pha (alternating path) là một đường đi đơn trong G bắt đầu bằng một X_đỉnh chưa ghép, đi theo một cạnh **chưa ghép**, rồi đến một cạnh **đã ghép**, rồi lại đến một cạnh **chưa ghép** ... cứ xen kẽ nhau như vậy.
- Một đường mở (augmenting path) **là một đường pha. Bắt đầu từ một X_đỉnh chưa ghép kết thúc bằng một Y_đỉnh chưa ghép**(*).

Ví dụ: với đồ thị phân đôi như hình bên, và bộ ghép M = {(X_1, Y_1), (X_2, Y_2)}
X_3 và Y_3 là những đỉnh chưa ghép, các đỉnh khác là đã ghép
Đường (X_3, Y_2, X_2, Y_1) là đường pha
Đường (X_3, Y_2, X_2, Y_1, X_1, Y_3) là đường mở.



III. THUẬT TOÁN ĐƯỜNG MỞ

Thuật toán đường mở để tìm một bộ ghép lớn nhất phát biểu như sau:

(*) Thực ra phải phát biểu một cách chặt chẽ: Nếu định hướng lại các cạnh của đồ thị thành cung, những cạnh chưa ghép được định hướng từ X sang Y, những cạnh đã ghép định hướng từ Y sang X. Trên đồ thị định hướng đó: Một đường đi xuất phát từ một X_đỉnh gọi là đường pha, một đường đi từ một X_đỉnh chưa ghép tới một Y_đỉnh chưa ghép gọi là đường mở.

- Bắt đầu từ một bộ ghép bất kỳ M (thông thường bộ ghép được khởi gán bằng bộ ghép rỗng hay được tìm bằng các thuật toán tham lam)
- Sau đó đi tìm một đường mở, nếu tìm được thì mở rộng bộ ghép M như sau: Trên đường mở, loại bỏ những cạnh đã ghép khỏi M và thêm vào M những cạnh chưa ghép. Nếu không tìm được đường mở thì bộ ghép hiện thời là lớn nhất.

Như ví dụ trên, với bộ ghép hai cạnh M = {(X_1, Y_1), (X_2, Y_2)} và đường mở tìm được gồm các cạnh
1. (X_3, Y_2) ∉ M; 2.(Y_2, X_2) ∈ M
3. (X_2, Y_1) ∉ M; 4.(Y_1, X_1) ∈ M
5. (X_1, Y_3) ∉ M
Vậy thì ta sẽ loại đi các cạnh (Y_2, X_2) và (Y_1, X_1) trong bộ ghép cũ và thêm vào đó các cạnh (X_3, Y_2), (X_2, Y_1), (X_1, Y_3) được bộ ghép 3 cạnh.

IV. CÀI ĐẶT

1. Biểu diễn đồ thị phân đôi

Giả sử đồ thị phân đôi G = (X∪Y, E) có các X_đỉnh ký hiệu là X[1], X[2], ..., X[m] và các Y_đỉnh ký hiệu là Y[1], Y[2], ..., Y[n]. Ta sẽ biểu diễn đồ thị phân đôi này bằng ma trận A cỡ mxn. Trong đó:
A[i, j] = TRUE nếu như có cạnh nối đỉnh X[i] với đỉnh Y[j].
A[i, j] = FALSE nếu như không có cạnh nối đỉnh X[i] với đỉnh Y[j].

2. Biểu diễn bộ ghép

Để biểu diễn bộ ghép, ta sử dụng hai mảng: matchX[1..m] và matchY[1..n].
matchX[i] là đỉnh thuộc tập Y ghép với đỉnh X[i]
matchY[j] là đỉnh thuộc tập X ghép với đỉnh Y[j].
Tức là nếu như cạnh (X[i], Y[j]) thuộc bộ ghép thì matchX[i] = j và matchY[j] = i.
Quy ước rằng:
Nếu như X[i] chưa ghép với đỉnh nào của tập Y thì matchX[i] = 0
Nếu như Y[j] chưa ghép với đỉnh nào của tập X thì matchY[j] = 0.
Để thêm một cạnh (X[i], Y[j]) vào bộ ghép thì ta chỉ việc đặt matchX[i] := j và matchY[j] := i;
Để loại một cạnh (X[i], Y[j]) khỏi bộ ghép thì ta chỉ việc đặt matchX[i] := 0 và matchY[j] := 0;

3. Tìm đường mở như thế nào.

Vì đường mở bắt đầu từ một X_đỉnh chưa ghép, đi theo một cạnh chưa ghép sang tập Y, rồi theo một đã ghép để về tập X, rồi lại một cạnh chưa ghép sang tập Y ... **cuối cùng là cạnh chưa ghép** tới một Y_đỉnh chưa ghép. Nên có thể thấy ngay rằng độ dài đường mở là lẻ và trên đường mở số cạnh ∈ M ít hơn số cạnh ∉ M là 1 cạnh. Và cũng dễ thấy rằng giải thuật tìm đường mở nên sử dụng thuật toán loang là hợp lý nhất.

Khởi tạo tập Left := {tập những X_đỉnh chưa ghép}
- Từ tập Left tính tập Right: Tập những đỉnh j của Y kề với một đỉnh i nào đó của tập Left qua một cạnh chưa ghép, lưu vết đường đi Trace[j] = i và coi như đánh dấu j lại để không bị đi lặp lưu ý rằng nếu như gặp đỉnh j chưa ghép thì dừng vòng lặp và kết luận tìm được đường mở kết thúc ở đỉnh chưa ghép j.
- Từ tập Right vừa tính, cập nhật lại tập Left là những đỉnh kề với một đỉnh nào đó của tập Right qua một cạnh đã ghép (Left := ∅; for ∀j∈Right do Left := Left ∪ {matchY[j]}).

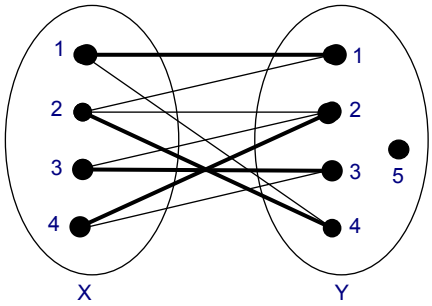
Rồi lại tiếp tục quay lại dùng Left tính Right ... cho tới khi hoặc đã tìm ra đường mở, hoặc không loang tiếp được nữa: Left = Right = ∅

4. Nhập đồ thị từ file văn bản B_GRAPH.INP

- Dòng 1: Ghi hai số m, n theo thứ tự là số X_đỉnh và số Y_đỉnh cách nhau 1 dấu cách

- Các dòng tiếp theo, mỗi dòng ghi hai số i, j cách nhau 1 dấu cách thể hiện có cạnh nối hai đỉnh (X[i], Y[j]) .

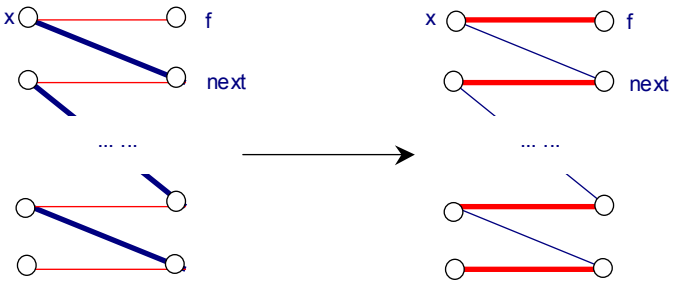
Ví dụ: Đồ thị phân đôi, file dữ liệu tương ứng và Output của chương trình:



B_GRAPH.INP	OUTPUT
4 5 1 1 1 4 2 1 2 2 2 4 3 2 3 3 4 2 4 3	Match: 1) X[1]--Y[1] 2) X[2]--Y[4] 3) X[3]--Y[3] 4) X[4]--Y[2]

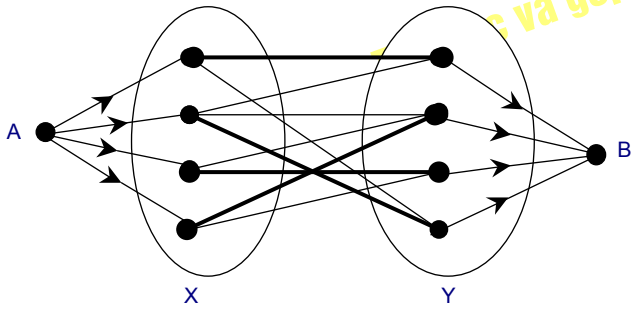
```
program MatchingProblem; {Augmenting Path Algorithm}
const
 max = 100;
var
 m, n: Byte;
 a: array[1..max, 1..max] of Boolean;
 matchX, matchY: array[1..max] of Byte;
 Trace: array[1..max] of Byte; {với y ∈ Y, Trace[y] là đỉnh ∈ X liền trước đỉnh y trên đường mở}

procedure Enter;
var
 f: Text;
 i, j: Byte;
begin
 FillChar(a, SizeOf(a), False);
 Assign(f, 'B_GRAPH.INP'); Reset(f);
 Readln(f, m, n);
 while not SeekEof(f) do
  begin
   Readln(f, i, j);
   a[i, j] := True;
  end;
end;

procedure Init;
begin
 FillChar(matchX, m, 0);
 FillChar(matchY, n, 0);
end;

{Thuật toán tìm đường mở, nếu thấy thì trả về đỉnh kết thúc của đường mở và mảng vết Trace, nếu không thấy trả về 0}
function FindAugmentingPath: Byte;
var
 Left, Right: set of Byte;
 x, y: Byte;
begin
 FillChar(Trace, SizeOf(Trace), 0);
 Left := [];
 for x := 1 to m do
  if matchX[x] = 0 then Left := Left + [x]; {Khởi tạo Left là tập những X_đỉnh chưa ghép}
 repeat
  {Dùng Left loang ra Right: tập những Y_đỉnh kề với một đỉnh nào đó của Left qua một cạnh chưa ghép}
  Right := [];
  for x := 1 to m do
   if x in Left then {Xét ∀x ∈ Left}
```

```
    for y := 1 to n do {Xét ∀ y chưa đánh dấu và kề với x qua cạnh (x, y) chưa ghép}
     if (Trace[y] = 0) and (matchX[x] <> y) and a[x, y] then
      begin
       Trace[y] := X; {Lưu vết đường đi cũng là đánh dấu (≠0)}
       if matchY[y] = 0 then {Nếu loang tới một đỉnh chưa ghép y ∈ Y thì dừng ngay với đỉnh kết thúc y}
        begin
         FindAugmentingPath := y;
         Exit;
        end;
       Right := Right + [y]; {Đưa y vào tập Right}
      end;
  {Bây giờ dùng Right cập nhật lại Left: Tập cácX_ đỉnh kề với một đỉnh nào đó của Right qua một cạnh đã ghép}
  Left := [];
  for y := 1 to n do
   if y in Right then Left := Left + [matchY[y]];
 until Left = [];
 FindAugmentingPath := 0; {ở trên không Exit được tức là không có đường mở}
end;

procedure Enlarge(f: Byte); {Nới rộng bộ ghép bởi đường mở kết thúc ở f}
var
 x, next: Byte;
begin
 repeat
  x := Trace[f];
  next := matchX[x];
  matchX[x] := f;
  matchY[f] := x;
  f := next;
 until f = 0;
end;
```




```
procedure Process; {Thuật toán đường mở}
var
 k: Byte;
begin
 repeat
  k := FindAugmentingPath; {Tìm đường mở}
  if k <> 0 then Enlarge(k); {Nếu tìm thấy thì nới rộng bộ ghép theo đường mở}
 until k = 0; {Cho tới khi không tìm thấy đường mở}
end;

procedure PrintResult;
var
 i, Count: Byte;
begin
 Writeln('Match: ');
 Count := 0;
 for i := 1 to m do
  if matchX[i] <> 0 then
   begin
    Inc(Count);
    Writeln(Count, ') X[', i, ']--Y[', matchX[i], ']');
   end;
end;

begin
 Enter;
 Init;
 Process;
 PrintResult;
end.
```

Khảo sát tính đúng đắn của thuật toán cho ta một kết quả khá thú vị:



Nếu ta thêm một đỉnh A và cho thêm m cung từ A tới tất cả những đỉnh của tập X, thêm một đỉnh B và nối thêm n cung từ tất cả các đỉnh của Y tới B. Ta được một mạng với đỉnh phát A và đỉnh thu B. Nếu đặt khả năng thông qua của các cung đều là 1 sau đó tìm luồng cực đại trên mạng bằng thuật toán Ford-Fulkerson thì theo định lý về tính nguyên, luồng tìm được trên các cung đều phải là số nguyên (tức là bằng 1 hoặc 0). Khi đó dễ thấy rằng những cung có luồng 1 từ tập X tới tập Y sẽ cho ta một bộ ghép lớn nhất. Để chứng minh thuật toán đường mở tìm được bộ ghép lớn nhất sau hữu hạn bước, ta sẽ chứng minh rằng số bộ ghép tìm được bằng thuật toán đường mở sẽ bằng giá trị luồng cực đại nói trên, điều đó cũng rất dễ bởi vì nếu để ý kỹ một chút thì đường mở chẳng qua là đường tăng luồng trên đồ thị tăng luồng mà thôi, ngay cái tên augmenting path đã cho ta biết điều này. Vì vậy thuật toán đường mở ở trường hợp này là một **cách cài đặt hiệu quả trên một dạng đồ thị đặc biệt**, nó làm cho chương trình sáng sủa hơn nhiều so với phương pháp tìm bộ ghép dựa trên bài toán luồng và thuật toán Ford-Fulkerson thuần túy.

Bài tập

1. Có n thợ và n công việc ($n \le 100$), mỗi thợ thực hiện được ít nhất một việc. Như vậy một thợ có thể làm được nhiều việc, và một việc có thể có nhiều thợ làm được. Hãy phân công n thợ thực hiện n việc đó sao cho mỗi thợ phải làm đúng 1 việc hoặc thông báo rằng không có cách phân công nào thoả mãn điều trên.

2. Có n thợ và m công việc ($n, m \le 100$). Mỗi thợ cho biết mình có thể làm được những việc nào, hãy phân công các thợ làm các công việc đó sao cho mỗi thợ phải làm ít nhất 2 việc và số việc thực hiện được là nhiều nhất.

§11. BÀI TOÁN TÌM BỘ GHÉP CỰC ĐẠI VỚI TRỌNG SỐ CỰC TIỂU TRÊN ĐỒ THỊ HAI PHÍA - THUẬT TOÁN HUNGARI

I. BÀI TOÁN PHÂN CÔNG

- Đây là một dạng bài toán trong thực tế thường hay gặp. Phát biểu như sau: Có m người (đánh số 1, 2, ..., m) và n công việc (đánh số 1, 2, ..., n), mỗi người có khả năng thực hiện một số công việc nào đó. Để giao cho người i thực hiện công việc j cần một chi phí là c(i, j) ≥ 0. Cần phân cho mỗi thợ một việc và mỗi việc chỉ do một thợ thực hiện sao cho số công việc có thể thực hiện được là nhiều nhất và nếu có ≥ 2 phương án đều thực hiện được nhiều công việc nhất thì chỉ ra phương án chi phí ít nhất.
- Dựng đồ thị hai phía G = (X∪Y, E) với X là tập m người, Y là tập n việc và (u, v) ∈ E với trọng số c(u, v) nếu như người u làm được công việc v. Bài toán đưa về tìm bộ ghép nhiều cạnh nhất của G có trọng số nhỏ nhất.
- Gọi k = max(m, n). Bổ sung vào tập X và Y một số đỉnh giả để $|X| = |Y| = k$.
- Gọi M là một số dương đủ lớn, chẳng hạn đặt: $M = 1 + \sum_{i=1}^{m}\sum_{j=1}^{n} c(i,j)$. Với mỗi cặp đỉnh (u, v): u ∈ X và v ∈ Y. Nếu (u, v) ∉ E thì ta bổ sung cạnh (u, v) vào E với trọng số là M.
- Khi đó ta được G là một **đồ thị phân đôi đầy đủ** (Đồ thị phân đôi mà giữa một đỉnh bất kỳ của X và một đỉnh bất kỳ của Y đều có cạnh nối). Và nếu như ta **tìm được bộ ghép đầy đủ k cạnh mang trọng số nhỏ nhất** thì ta chỉ cần **loại bỏ khỏi bộ ghép đó những cạnh mang trọng số M vừa thêm vào** thì sẽ được kế hoạch phân công 1 người ↔ 1 việc cần tìm. Điều này dễ hiểu bởi bộ ghép đầy đủ mang trọng số nhỏ nhất tức là phải ít cạnh trọng số M nhất, tức là số phép phân công là nhiều nhất, và tất nhiên trong số các phương án ghép ít cạnh trọng số M nhất thì đây là phương án trọng số nhỏ nhất, tức là tổng chi phí trên các phép phân công là ít nhất.

II. PHÂN TÍCH

- Vào: Đồ thị hai phía đầy đủ G = (X∪Y, E); $|X| = |Y| = k$. Được cho bởi ma trận vuông C cỡ kxk, c_{ij} = trọng số cạnh nối đỉnh X_i với Y_j. Giả thiết $c_{ij} \geq 0$. với mọi i, j.
- Ra: Bộ ghép đầy đủ trọng số nhỏ nhất.

Hai định lý sau đây tuy rất đơn giản nhưng là những định lý quan trọng tạo cơ sở cho thuật toán sẽ trình bày:

Định lý 1: Loại bỏ khỏi G những cạnh trọng số > 0. Nếu những cạnh trọng số 0 còn lại tạo ra bộ ghép k cạnh trong G thì đây là bộ ghép cần tìm.

Chứng minh: Theo giả thiết, các cạnh của G mang trọng số không âm nên bất kỳ bộ ghép nào trong G cũng có trọng số không âm, mà bộ ghép ở trên mang trọng số 0, nên tất nhiên đó là bộ ghép đầy đủ trọng số nhỏ nhất.

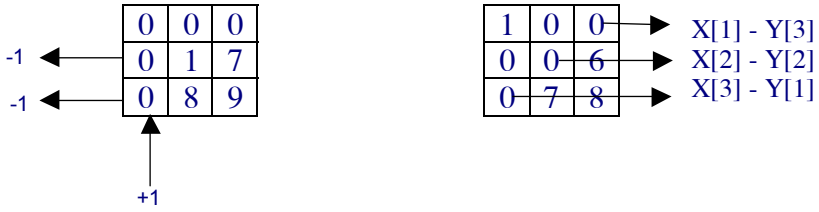
Định lý 2: Với đỉnh X_i, nếu ta cộng thêm một số Δ(dương hay âm) vào tất cả những cạnh liên thuộc với X_i (tương đương với việc cộng thêm Δ vào tất cả các phần tử thuộc hàng i của ma trận C) thì không ảnh hưởng tới bộ ghép đầy đủ trọng số nhỏ nhất.

Chứng minh: Với một bộ ghép đầy đủ bất kỳ thì có một và chỉ một cạnh ghép với X_i. Nên việc cộng thêm Δ vào tất cả các cạnh liên thuộc với X_i sẽ làm cho trọng số bộ ghép đó lên Δ. Vì vậy nếu như ban đầu, M là bộ ghép đầy đủ trọng số nhỏ nhất thì sau thao tác trên, M vẫn là bộ ghép đầy đủ trọng số nhỏ nhất.

Hệ quả: Với đỉnh Y_j, nếu ta cộng thêm một số Δ (dương hay âm) vào tất cả những cạnh liên thuộc với Y_j (tương đương với việc cộng thêm Δ vào tất cả các phần tử thuộc cột j của ma trận C) thì không ảnh hưởng tới bộ ghép đầy đủ trọng số nhỏ nhất.

Từ đây có thể nhận ra tư tưởng của thuật toán: Từ đồ thị G, ta tìm chiến lược cộng / trừ một cách hợp lý trọng số của các cạnh liên thuộc với một đỉnh nào đó để được một đồ thị mới vẫn có các cạnh trọng số không âm, mà các cạnh trọng số 0 của đồ thị mới đó chứa một bộ ghép đầy đủ k cạnh.

Ví dụ: Biến đổi ma trận trọng số của đồ thị hai phía 3 đỉnh trái, 3 đỉnh phải:



III. THUẬT TOÁN

1. Các khái niệm:

Để cho gọn, ta gọi những cạnh trọng số 0 của G là những 0_cạnh.

Xét một bộ ghép M chỉ gồm những 0_cạnh.

- Những đỉnh ∈ M gọi là những đỉnh đã ghép, những đỉnh còn lại gọi là những đỉnh chưa ghép.
- Những 0_cạnh ∈ M gọi là những 0_cạnh đã ghép, những 0_cạnh còn lại là những 0_cạnh chưa ghép.
- Đường pha (Alternating Path) là một đường đi bắt đầu từ một đỉnh $x \in X$ chưa ghép, đi theo một 0_cạnh chưa ghép sang tập Y, rồi theo một 0_cạnh đã ghép về tập X, rồi lại theo một 0_cạnh chưa ghép sang tập Y v.v... Cứ xen kẽ nhau như vậy. Việc xác định những đỉnh nào có thể đến được từ x bằng một đường pha có thể sử dụng các thuật toán tìm kiếm trên đồ thị (BFS hoặc DFS). Những đỉnh và những cạnh được duyệt qua tạo thành một cây pha gốc x
- Một đường mở (Augmenting Path) là một đường pha, bắt đầu từ một X_đỉnh chưa ghép, kết thúc bằng một 0_cạnh chưa ghép tới một Y_đỉnh chưa ghép. Lưu ý rằng: a) Đường đi trực tiếp từ một X_đỉnh chưa ghép tới một Y_đỉnh chưa ghép qua một 0_cạnh chưa ghép cũng là một đường mở. b) Dọc trên đường mở, số 0_cạnh chưa ghép nhiều hơn số 0_cạnh đã ghép đúng 1 cạnh.

2. Thuật toán Hungari

Bước 1: Khởi tạo:

- Một bộ ghép M := ∅

Bước 2: Với mọi đỉnh $x^* \in X$, ta tìm cách ghép x^* như sau.

Bắt đầu từ đỉnh x^* chưa ghép, thử tìm đường mở bắt đầu ở x^* bằng thuật toán tìm kiếm trên đồ thị (BFS hoặc DFS - thông thường nên dùng BFS để tìm đường qua ít cạnh nhất) có hai khả năng xảy ra:

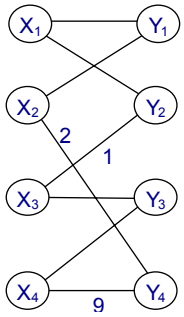
- Hoặc tìm được đường mở thì dọc theo đường mở, ta loại bỏ những cạnh đã ghép khỏi M và thêm vào M những cạnh chưa ghép, ta được một **bộ ghép mới nhiều hơn bộ ghép cũ 1 cạnh** và **đỉnh x^* trở thành đã ghép**.
- Hoặc không tìm được đường mở thì do ta sử dụng thuật toán tìm kiếm trên đồ thị nên có thể xác định được hai tập:
  - VisitedX = {Tập những X_đỉnh có thể đến được từ x^* bằng một đường pha}
  - VisitedY = {Tập những Y_đỉnh có thể đến được từ x^* bằng một đường pha}
  - Gọi Δ là trọng số nhỏ nhất của các cạnh nối giữa một đỉnh thuộc VisitedX với một đỉnh không thuộc VisitedY. Dễ thấy Δ > 0 bởi nếu Δ = 0 thì tồn tại một 0_cạnh (x, y) với $x \in$ VisitedX và $y \notin$ VisitedY. Vì x^* đến được x bằng một đường pha và (x, y) là một 0_cạnh nên x^* cũng đến được y bằng một đường pha, dẫn tới y ∈ VisitedY, điều này vô lý.
  - Biến đổi đồ thị G như sau: Với ∀x ∈ VisitedX, trừ Δ vào trọng số những cạnh liên thuộc với x, Với ∀ y ∈ VisitedY, cộng Δ vào trọng số những cạnh liên thuộc với y.

- Lặp lại thủ tục tìm kiếm trên đồ thị thử tìm đường mở xuất phát ở x^* cho tới khi tìm ra đường mở.

Bước 3: Sau bước 2 thì mọi X_đỉnh đều được ghép, in kết quả về bộ ghép tìm được.

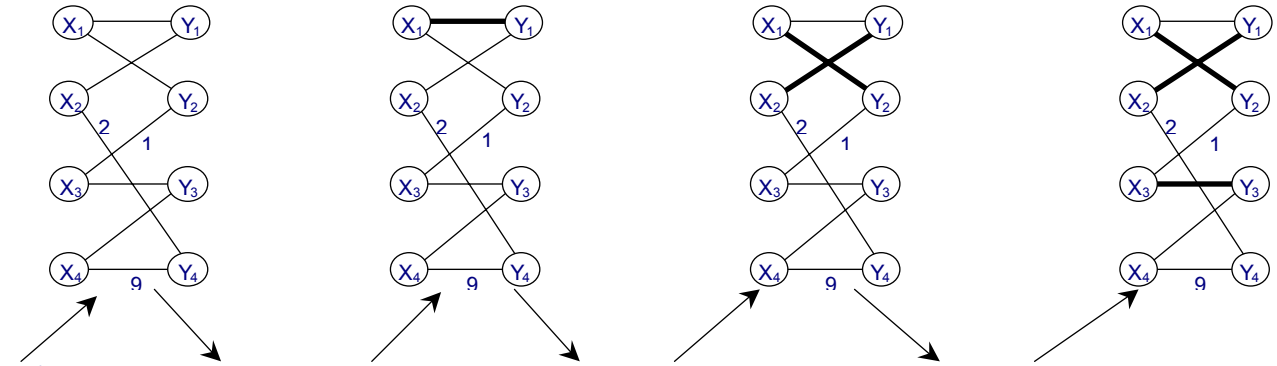
Mô hình cài đặt của thuật toán có thể viết như sau:

```
<Khởi tạo: M := ∅ ...>
for (x*∈X) do
 begin
  repeat
   <Tìm đường mở xuất phát ở x*>
   if <Không tìm thấy đường mở> then <Biến đổi đồ thị G: Chọn Δ := ...>
  until <Tìm thấy đường mở>;
  <Dọc theo đường mở, loại bỏ những cạnh đã ghép khỏi M
   và thêm vào M những cạnh chưa ghép>
 end;
<Kết quả>
```



Ví dụ minh hoạ:

Để không bị rối hình, ta hiểu những cạnh không ghi trọng số là những 0_cạnh, những cạnh không vẽ mang trọng số rất lớn trong trường hợp này không cần thiết phải tính đến. Những cạnh nét đậm là những cạnh đã ghép, những cạnh nét thanh là những cạnh chưa ghép.

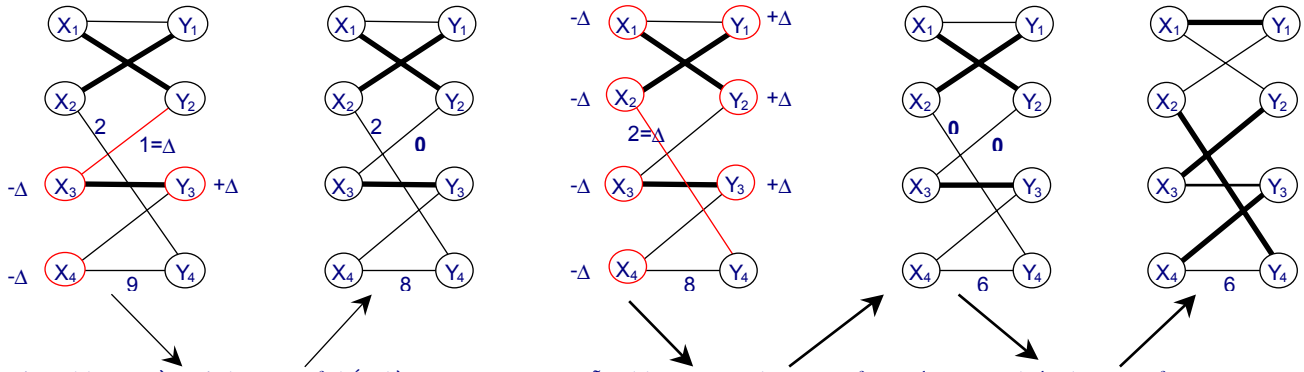


Khởi tạo: Bộ ghép M = ∅

i=1; Tìm được đường mở $X_1 \Rightarrow Y_1$: Tăng cặp

i=2; Tìm được đường mở $X_2 \Rightarrow Y_1 \rightarrow X1 \Rightarrow$ Y2: Tăng cặp

i=3; Tìm được đường mở $X_3 \Rightarrow Y_3$: Tăng cặp



i=4, không tồn tại đường mở bắt đầu từ X_4. Xác định các đỉnh đến được từ X_4 bằng một đường pha: VisitedX={X_3, X_4}; VisitedY = {Y_3}. Trọng số nhỏ nhất của cạnh nối một đỉnh của VisitedX với một đỉnh không thuộc VisitedY là 1. Trừ 1 vào trọng số các cạnh liên thuộc với X_3 và X_4 sau đó cộng 1 vào trọng số các cạnh liên thuộc với Y_3

Vẫn không có đường mở xuất phát ở X_4 Xác định:
VisitedX={X_1, X_2, X_3, X_4}
VisitedY={Y_1, Y_2, Y_3}
Δ=2
Biến đổi tiếp

Tìm thấy đường mở $X_4 \Rightarrow Y_3 \rightarrow X_3 \Rightarrow Y_2 \rightarrow X_1 \Rightarrow Y_1 \rightarrow X_2 \Rightarrow Y_4$. Tăng cặp ⇒ Xong

Để ý rằng nếu như không tìm thấy đường mở xuất phát ở x^* thì quá trình tìm kiếm trên đồ thị sẽ cho ta một cây pha gốc x^*. Giá trị xoay Δ thực chất là trọng số nhỏ nhất của cạnh nối một X_đỉnh trong cây pha với một Y_đỉnh ngoài cây pha (cạnh ngoài). Việc trừ Δ vào những cạnh liên thuộc với X_đỉnh trong cây pha và cộng Δ vào những cạnh liên thuộc với Y_đỉnh trong cây pha sẽ làm cho cạnh ngoài nói trên trở thành 0_cạnh, các cạnh khác vẫn có trọng số ≥ 0. Nhưng quan trọng hơn là ***tất cả những cạnh trong cây pha vẫn cứ là 0_cạnh.*** *Điều đó đảm bảo cho quá trình tìm kiếm trên đồ thị lần sau sẽ xây dựng được cây pha mới lớn hơn cây pha cũ (Thể hiện ở chỗ: tập VisitedY sẽ rộng hơn trước ít nhất 1 phần tử). Vì tập các Y_ đỉnh đã ghép là hữu hạn nên sau không quá k bước, sẽ có một Y_đỉnh chưa ghép ∈ VisitedY, tức là tìm ra đường mở*

Trên thực tế, để chương trình hoạt động nhanh hơn, trong bước khởi tạo, người ta có thể thêm một thao tác:

Với mỗi đỉnh x ∈ X, xác định trọng số nhỏ nhất của các cạnh liên thuộc với x, sau đó trừ tất cả trọng số các cạnh liên thuộc với x đi trọng số nhỏ nhất đó. Làm tương tự như vậy với các Y_đỉnh. Điều này tương đương với việc trừ tất cả các phần tử trên mỗi hàng của ma trận C đi giá trị nhỏ nhất trên hàng đó, rồi lại trừ tất cả các phần tử trên mỗi cột của ma trận C đi phần tử nhỏ nhất trên cột đó. Khi đó số 0_cạnh của đồ thị là khá nhiều, có thể chứa ngay bộ ghép đầy đủ hoặc chỉ cần qua ít bước biến đổi là sẽ chứa bộ ghép đầy đủ k cạnh.

Để tưởng nhớ hai nhà toán học Konig và Egervary, những người đã đặt cơ sở lý thuyết đầu tiên cho phương pháp, người ta đã lấy tên của đất nước sinh ra hai nhà toán học này để đặt tên cho thuật toán. Mặc dù sau này có một số cải tiến nhưng tên gọi Thuật toán Hungari (Hungarian Algorithm) vẫn được dùng phổ biến.

IV. CÀI ĐẶT

1. Phương pháp đối ngẫu Kuhn-Munkres (Không làm biến đổi ma trận C ban đầu)

Phương pháp Kuhn-Munkres đi tìm hai dãy số Fx[1..k] và Fy[1..k] thoả mãn:

- c[i, j] - Fx[i] - Fy[j] ≥ 0
- Tập các cạnh (X[i], Y[j]) thoả mãn c[i, j] - Fx[i] - Fy[j] = 0 chứa trọn một bộ ghép đầy đủ k cạnh, đây chính là bộ ghép cần tìm.

Chứng minh:

Nếu tìm được hai dãy số thoả mãn trên thì ta chỉ việc thực hiện hai thao tác:

Với mỗi đỉnh X[i], trừ tất cả trọng số của những cạnh liên thuộc với X[i] đi Fx[i]

Với mỗi đỉnh Y[j], trừ tất cả trọng số của những cạnh liên thuộc với Y[j] đi Fy[j]

(Hai thao tác này tương đương với việc trừ tất cả trọng số của các cạnh (X[i], Y[j]) đi một lượng Fx[i] + Fy[j]: c[i, j] := c[i, j] - Fx[i] - Fy[j])

Thì dễ thấy đồ thị mới tạo thành sẽ gồm có các cạnh trọng số không âm và những 0_cạnh của đồ thị chứa trọn một bộ ghép đầy đủ.

Như ví dụ trên:

	1	2	3	4	
1	**0**	0	M	M	Fx[1] = 2
2	0	M	M	**2**	Fx[2] = 2
3	M	**1**	0	M	Fx[3] = 3
4	M	M	**0**	9	Fx[4] = 3
	Fy[1] = -2	Fy[2] = -2	Fy[3] = -3	Fy[4] = 0	

(Có nhiều phương án khác: Fx = (0, 0, 1, 1); Fy = (0, 0, -1, 2) cũng đúng)

Vậy phương pháp Kuhn-Munkres đưa việc biến đổi đồ thị G (biến đổi ma trận C) về việc biến đổi hay dãy số Fx và Fy. Việc trừ Δ vào trọng số tất cả những cạnh liên thuộc với X[i] tương đương với việc tăng Fx[i] lên Δ. Việc cộng Δ vào trọng số tất cả những cạnh liên thuộc với Y[j] tương đương với giảm Fy[j] đi Δ. Khi cần biết trọng số cạnh (X[i], Y[j]) là bao nhiêu sau các bước biến đổi, thay vì viết c[i, j], ta viết c[i, j] - Fx[i] - Fy[j].

Ví dụ: Thủ tục tìm đường mở trong thuật toán Hungari đòi hỏi phải xác định được cạnh nào là 0_cạnh, khi cài đặt bằng phương pháp Kuhn-Munkres, việc xác định cạnh nào là 0_cạnh có thể kiểm tra bằng đẳng thức: c[i, j] - Fx[i] - Fy[j] = 0 hay **c[i, j] = Fx[i] + Fy[j].**

Câu hỏi: Tìm hiểu **bản chất** của hai dãy Fx[1..k] và Fy[1..k]. Nếu ta tìm được hai dãy Fx và Fy thoả mãn hai điều kiện nêu nêu ở đầu mục thì tổng các phần tử ở hai dãy sẽ là gì ?

Sơ đồ cài đặt phương pháp Kuhn-Munkres có thể viết như sau:

Bước 1: Khởi tạo:

M := ∅;

Việc khởi tạo các Fx, Fy có thể có nhiều cách chẳng hạn Fx[i] := 0; Fy[j] := 0 với ∀i, j.

Hoặc: Fx[i] := $\min_{1\le j\le k}(c[i, j])$ với ∀i. Sau đó đặt Fy[j] := $\min_{1\le i\le k}(c[i, j] - Fx[i])$ với ∀j.

(Miễn sao c[i, j] - Fx[i] - Fy[j] ≥ 0)

Bước 2: Với mọi đỉnh $x^* \in X$, ta tìm cách ghép x^* như sau:

Bắt đầu từ đỉnh x^*, thử tìm đường mở bắt đầu ở x^* bằng thuật toán tìm kiếm trên đồ thị (BFS hoặc DFS). Lưu ý rằng 0_cạnh là cạnh thoả mãn c[i, j] = Fx[i] + Fy[j]. Có hai khả năng xảy ra:

- Hoặc tìm được đường mở thì dọc theo đường mở, ta loại bỏ những cạnh đã ghép khỏi M và thêm vào M những cạnh chưa ghép.
- Hoặc không tìm được đường mở thì xác định được hai tập:
  - ❖ VisitedX = {Tập những X_đỉnh có thể đến được từ x^* bằng một đường pha}
  - ❖ VisitedY = {Tập những Y_đỉnh có thể đến được từ x^* bằng một đường pha}
  - ❖ Đặt Δ := min{c[i, j] - Fx[i] - Fy[j] | ∀X[i] ∈ VisitedX; ∀Y[j] ∉ VisitedY}
  - ❖ Với ∀X[i] ∈ VisitedX: Fx[i] := Fx[i] + Δ;
  - ❖ Với ∀Y[j] ∈ VisitedY: Fy[j] := Fy[j] - Δ;
  - ❖ Lặp lại thủ tục tìm đường mở xuất phát tại x^* cho tới khi tìm ra đường mở.

Đáng lưu ý ở phương pháp Kuhn-Munkres là nó không làm thay đổi ma trận C ban đầu. Điều đó thực sự hữu ích trong trường hợp trọng số của cạnh (X[i], Y[j]) không được cho một cách tường minh bằng giá trị C[i, j] mà lại cho bằng hàm c(i, j): trong trường hợp này, việc trừ hàng/cộng cột trực tiếp trên ma trận chi phí C là không thể thực hiện được.

2. Dưới đây ta sẽ cài đặt chương trình giải bài toán phân công bằng thuật toán Hungari với phương pháp đối ngẫu Kuhn-Munkres:

a) Biểu diễn bộ ghép

Để biểu diễn bộ ghép, ta sử dụng hai mảng: matchX[1..k] và matchY[1..k].

- matchX[i] là đỉnh thuộc tập Y ghép với đỉnh X[i]
- matchY[j] là đỉnh thuộc tập X ghép với đỉnh Y[j].

Tức là nếu như cạnh (X[i], Y[j]) thuộc bộ ghép thì matchX[i] = j và matchY[j] = i.

Quy ước rằng:

- Nếu như X[i] chưa ghép với đỉnh nào của tập Y thì matchX[i] = 0
- Nếu như Y[j] chưa ghép với đỉnh nào của tập X thì matchY[j] = 0.
- Để thêm một cạnh (X[i], Y[j]) vào bộ ghép thì chỉ việc đặt matchX[i] := j và matchY[j] := i;
- Để loại một cạnh (X[i], Y[j]) khỏi bộ ghép thì chỉ việc đặt matchX[i] := 0 và matchY[j] := 0;

b) Tìm đường mở như thế nào.

Đường mở bắt đầu từ một đỉnh x^* chưa ghép, đi theo một 0_cạnh chưa ghép sang tập Y, rồi theo một 0_cạnh đã ghép để về tập X, rồi lại một 0_cạnh chưa ghép sang tập Y ... **cuối cùng là 0_cạnh chưa ghép** tới một Y_đỉnh chưa ghép. Ta sẽ tìm đường mở và xây dựng hai tập VisitedX và VisitedY bằng thuật toán Loang:

Khởi tạo tập Left := {x^*}. Đánh dấu x^* đã thăm.

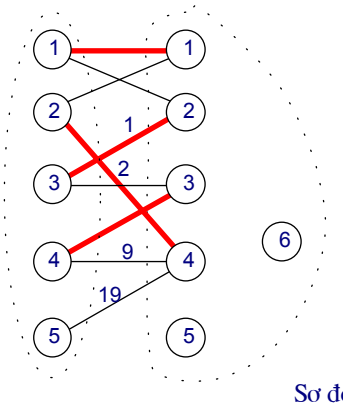
- ❖ Từ tập Left tính tập Right: Tập những đỉnh y∈Y chưa thăm kề với một đỉnh x∈Left qua một 0_cạnh chưa ghép, đồng thời đánh dấu các đỉnh y là đã thăm. Nếu như y chưa ghép thì dừng vòng lặp và kết luận tìm được đường mở kết thúc ở đỉnh chưa ghép y.

❖ Từ tập Right vừa tính, cập nhật lại tập Left là những đỉnh kề với một đỉnh nào đó của tập Right qua một cạnh đã ghép (Left := ∅; for ∀y∈Right do Left := Left ∪ {matchY[y]}).

Rồi lại tiếp tục quay lại dùng Left tính Right ... cho tới khi hoặc đã tìm ra đường mở, hoặc không loang tiếp được nữa: Left = Right = ∅, khi đó những đỉnh đánh dấu đã thăm cho ta hai tập VisitedX và VisitedY.

3. Nhập dữ liệu từ file văn bản ASSIGN.INP

- Dòng 1: Ghi hai số m, n theo thứ tự là số thợ và số việc cách nhau 1 dấu cách
- Các dòng tiếp theo, mỗi dòng ghi ba số i, j, c[i, j] cách nhau 1 dấu cách thể hiện thợ i làm được việc j và chi phí để làm là c[i, j].



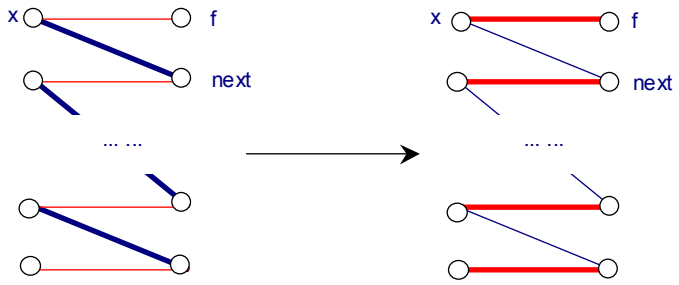
ASSIGN.INP	OUTPUT
5 6 1 1 0 1 2 0 2 1 0 2 4 2 3 2 1 3 3 0 4 3 0 4 4 9 5 4 19	Optimal Assignment: 1) X[1] <--> Y[1] 0 2) X[2] <--> Y[4] 2 3) X[3] <--> Y[2] 1 4) X[4] <--> Y[3] 0 Total Spending: 3

Sơ đồ khả năng làm việc và file dữ liệu, output tương ứng

```
program AssignmentProblemSolve; {Giải bài toán phân công}
const
 max = 100;
 maxLong = 1000000;
var
 c: array[1..max, 1..max] of LongInt; {c[i,j] = trọng số cạnh nối X[i] với Y[j]}
 Fx, Fy: array[1..max] of LongInt; {Các giá trị đối ngẫu}
 matchX, matchY: array[1..max] of Byte;
 VisitedX, VisitedY: array[1..max] of Boolean; {Đánh dấu TRUE nếu một đỉnh thuộc cây pha}
 Trace: array[1..max] of Byte; {Vết dùng để truy đường mở}
 m, n, k: Byte;

procedure Enter;
var
 f: Text;
 i, j: Byte;
begin
 Assign(f, 'ASSIGN.INP'); Reset(f);
 Readln(f, m, n);
 if m > n then k := m else k := n; {k = max(m, n), coi như có k thợ, k việc}
 for i := 1 to k do
  for j := 1 to k do c[i, j] := maxLong; {Những (i, j) nào không có trong file thì c[i, j]:=maxLong}
 while not SeekEof(f) do Readln(f, i, j, c[i, j]);
 Close(f);
end;

procedure Init;
var
 i, j: Byte;
begin
 {Khởi tạo một bộ ghép rỗng, chưa có cạnh nào}
 FillChar(matchX, k, 0);
 FillChar(matchY, k, 0);
 {Ta hoàn toàn có thể đặt tất cả các Fx cũng như Fy bằng 0, nhưng để nhanh hơn thì nên}
 for i := 1 to k do {Trừ trọng số các cạnh liên thuộc với X[ i] cho trọng số cạnh nhỏ nhất, tức là}
  begin {Đặt Fx[i] := trọng số cạnh nhỏ nhất liên thuộc với X[i]}
   Fx[i] := maxLong;
```

```
    for j := 1 to k do
     if c[i, j] < Fx[i] then Fx[i] := c[i, j];
   end;
  for j := 1 to k do {Rồi lại trừ tất cả trọng số những cạnh liên thuộc với Y[j] cho trọng số cạnh nhỏ nhất}
   begin { Lưu ý rằng trọng số cạnh (X[i], Y[j]) bây giờ là c[i, j] - Fx[i] chứ không còn là c[i, j] nữa}
    Fy[j] := maxLong;
    for i := 1 to k do
     if c[i, j] - Fx[i] < Fy[j] then Fy[j] := c[i, j] - Fx[i];
   end;
end;

{Tìm đường mở bắt đầu ở StartX∈X, nếu tìm thấy trả về đỉnh kết thúc đường mở, nếu không thấy trả về 0}
function FindAugmentingPath(StartX: Byte): Byte;
var
 Left, Right: set of Byte;
 x, y: Byte;
begin
 Left := [StartX];
 FillChar(VisitedX, k, False); FillChar(VisitedY, k, False); {Các đỉnh đều chưa thăm}
 VisitedX[StartX] := True; {Ngoại trừ đỉnh StartX hiển nhiên đến được từ X nên được thăm}
 repeat
  {Từ Left tính Right: Tập các Y_đỉnh chưa thăm kề với một đỉnh nào đó của Left qua một 0_cạnh chưa ghép}
  Right := [];
  for x := 1 to k do
   if x in Left then
    for y := 1 to k do
     if not VisitedY[y] and (matchY[y] <> x) and (c[x, y] = Fx[x] + Fy[y]) then
      begin
       Trace[y] := x; {Lưu vết đường}
       if matchY[y] = 0 then {y chưa ghép thì dừng ngay với đường mở từ StartX tới y}
        begin
         FindAugmentingPath := y;
         Exit;
        end;
       Right := Right + [y];
       VisitedY[y] := True;
      end;
  {Từ tập Right, tính lại Left là tập các X_đỉnh kề với một đỉnh nào đó của tập Right qua một cạnh đã ghép }
  Left := [];
  for y := 1 to k do
   if y in Right then
    begin
     x := matchY[y];
     Left := Left + [x];
     VisitedX[x] := True;
    end;
 until Right = []; {Không loang tiếp được nữa}
 FindAugmentingPath := 0; {ở trên không Exit được tức là không có đường}
end;

procedure SubX_AddY; {Biến đổi đồ thị G}
var
 x, y: Byte;
 Delta: LongInt;
begin
 Delta := maxLong; {Tính Δ là trọng số nhỏ nhất trong các cạnh nối VisitedX với Y\VisitedY }
 for x := 1 to k do
  if VisitedX[x] then
   for y := 1 to k do
    if not VisitedY[y] and (c[x, y] - Fx[x] - Fy[y] < Delta) then
          Delta := c[x, y] - Fx[x] - Fy[y];
 for x := 1 to k do {Trừ trọng số của tất cả các cạnh liên thuộc với x∈ VisitedX đi Δ}
```

```
  if VisitedX[x] then Fx[x] := Fx[x] + Delta;
 for y := 1 to k do {Cộng trọng số của tất cả các cạnh liên thuộc với y∈ Visited lên Δ}
  if VisitedY[y] then Fy[y] := Fy[y] - Delta;
end;

procedure Enlarge(f: Byte); {Nới rộng bộ ghép bởi đường mở kết thúc ở f∈Y}
var
 x, next: Byte;
begin
 repeat
  x := Trace[f];
  next := matchX[x];
  matchX[x] := f;
  matchY[f] := x;
  f := Next;
 until f = 0;
end;
```




```
procedure Solve; {Thuật toán Hungari}
var
 x, y: Byte;
begin
 for x := 1 to k do {Xét ∀x∈X}
  begin
   repeat
    y := FindAugmentingPath(x); {Tìm đường mở xuất phát tại x}
    if y = 0 then SubX_AddY; {Nếu không thấy thì biến đổi đồ thị}
   until y <> 0; {Cho tới khi thấy đường mở xuất phát ở x}
   Enlarge(y);   {Nới rộng bộ ghép dọc theo đường mở tìm được, x trở thành đã ghép}
  end;
end;

procedure Result;
var
 x, y: Byte;
 Count, W: LongInt;
begin
 Writeln('Assignment Solution:');
 W := 0; Count := 0;
 for x := 1 to k do
  begin
   y := matchX[x];
   {Những cạnh ghép có trọng số maxLong tương ứng với 1 thợ không được giao việc và 1 việc không được phân công}
   if c[x, y] < maxLong then {Nên chỉ cần in những phép phân công thật sự}
    begin
     Inc(Count);
     Writeln(Count:5, ') X[', x:3, '] <--> Y[', y:3, ']');
     W := W + c[x, y];
    end;
  end;
 Writeln('Total Spending: ', W);
end;

begin
 Enter;
 Init;
 Solve;
 Result;
end.
```

Nhận xét:

1. Nếu cài đặt như trên thì cho dù đồ thị có cạnh mang trọng số âm, chương trình vẫn tìm được bộ ghép cực đại với trọng số cực tiểu. Lý do: Ban đầu, ta trừ tất cả các phần tử trên mỗi hàng

của ma trận C đi giá trị nhỏ nhất trên hàng đó, rồi lại trừ tất cả các phần tử trên mỗi cột của ma trận C đi giá trị nhỏ nhất trên cột đó (Phép trừ ở đây làm gián tiếp qua các Fx, Fy chứ không phải trừ trực tiếp trên ma trận C). Nên sau bước này, tất cả các cạnh của đồ thị sẽ có trọng số không âm bởi phần tử nhỏ nhất trên mỗi cột của C chắc chắn là 0.

2. Sau khi kết thúc thuật toán, tổng tất cả các phần tử ở hai dãy Fx, Fy bằng trọng số cực tiểu của bộ ghép đầy đủ tìm được.

V. BÀI TOÁN TÌM BỘ GHÉP CỰC ĐẠI VỚI TRỌNG SỐ CỰC ĐẠI TRÊN ĐỒ THỊ HAI PHÍA

Ở trên đã trình bày bài toán tìm bộ ghép cực đại với trọng số cực tiểu được đưa về bài toán tìm bộ ghép đầy đủ trọng số cực tiểu bằng thuật toán Hungari. Bài toán tìm bộ ghép cực đại với trọng số cực đại cũng có thể giải nhờ phương pháp Hungari bằng cách đổi dấu tất cả các phần tử ma trận chi phí (Nhờ nhận xét 1).

Khi cài đặt, ta có thể sửa lại đôi chút trong chương trình trên để giải bài toán tìm bộ ghép cực đại với trọng số cực đại. Cụ thể như sau:

Bước 1: Khởi tạo:

- M := ∅;
- Khởi tạo hai dãy Fx và Fy thoả mãn: ∀i, j: **Fx[i] + Fy[j] ≥ c[i, j]**; Chẳng hạn ta có thể đặt Fx[i] := Phần tử lớn nhất trên dòng i của ma trận C và đặt các Fy[j] := 0.

Bước 2: Với mọi đỉnh $x^* \in X$, ta tìm cách ghép x^* như sau:

Với cách hiểu 0_cạnh là cạnh thoả mãn **c[i, j] = Fx[i] + Fy[j]**. Bắt đầu từ đỉnh x^*, thử tìm đường mở bắt đầu ở x^*. Có hai khả năng xảy ra:

- Hoặc tìm được đường mở thì dọc theo đường mở, ta loại bỏ những cạnh đã ghép khỏi M và thêm vào M những cạnh chưa ghép.
- Hoặc không tìm được đường mở thì xác định được hai tập:
  - VisitedX = {Tập những X_đỉnh có thể đến được từ x^* bằng một đường pha}
  - VisitedY = {Tập những Y_đỉnh có thể đến được từ x^* bằng một đường pha}
  - Đặt $\Delta := \min\{\mathbf{Fx[i] + Fy[j] - c[i, j]} \mid \forall X[i] \in \text{VisitedX}; \forall Y[j] \notin \text{VisitedY}\}$
  - Với $\forall X[i] \in$ VisitedX: Fx[i] := Fx[i] - Δ;
  - Với $\forall Y[j] \in$ VisitedY: Fy[j] := Fy[j] + Δ;
  - Lặp lại thủ tục tìm đường mở xuất phát tại x^* cho tới khi tìm ra đường mở.

Bước 3: Sau bước 2 thì mọi X_đỉnh đều đã ghép, ta được một bộ ghép đầy đủ k cạnh với trọng số lớn nhất.

Dễ dàng chứng minh được tính đúng đắn của phương pháp, bởi nếu ta đặt:

c'[i, j] = - c[i, j]; F'x[i] := - Fx[i]; F'y[j] = - Fy[j].

Thì bài toán trở thành tìm cặp ghép đầy đủ trọng số cực tiểu trên đồ thị hai phía với ma trận trọng số c'[1..k, 1..k]. Bài toán này được giải quyết bằng cách tính hai dãy đối ngẫu F'x và F'y. Từ đó bằng những biến đổi đại số cơ bản, ta có thể kiểm chứng được tính tương đương giữa các bước của phương pháp nêu trên với các bước của phương pháp Kuhn-Munkres ở mục trước.

Graph Theory Glossary

Chris Caldwell

This glossary is written to suplement the Interactive Tutorials in Graph Theory written using the Web Tutor. Here we define the terms that we introduce in our tutorials--you may need to go to the library to find the definitions of more advanced terms. Please let me know of any corrections or suggestion!

[A B C D E F G H I J K L M N O P Q R S T U V W X Y Z]

---

adjacent

Two vertices are adjacent if they are connected by an edge.

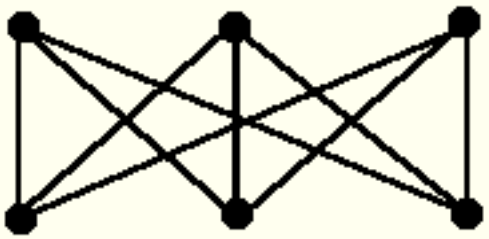
arc

A synonym for edge. See graph.

articulation point

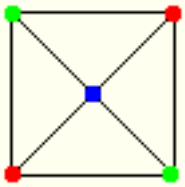
See cut vertices.

bipartite



A graph is **bipartite** if its vertices can be partitioned into two disjoint subsets U and V such that each edge connects a vertex from U to one from V. A bipartite graph is a **complete bipartite** graph if every vertex in U is connected to every vertex in V. If U has *n* elements and V has *m*, then we denote the resulting complete bipartite graph by K*n,m*. The illustration shows $K_{3,3}$. See also complete graph and cut vertices.

chromatic number



The chromatic number of a graph is the least number of colors it takes to color its vertices so that adjacent vertices have different colors. For example, this graph has chromatic number three.

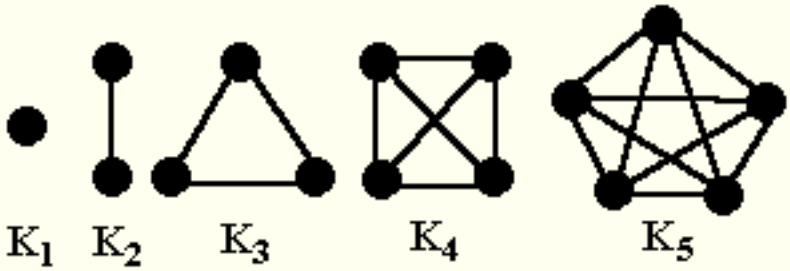
When applied to a map this is the least number of colors so necessary that countries that share non-trivial borders (borders consisting of more than single points) have different colors. See the Four Color Theorem.

circuit

A circuit is a path which ends at the vertex it begins (so a loop is an circuit of length one).

complete graph

A complete graph with *n* vertices (denoted K*n*) is a graph with *n* vertices in which each vertex is connected to each of the others (with one edge between each pair of vertices). Here are the first five complete graphs:

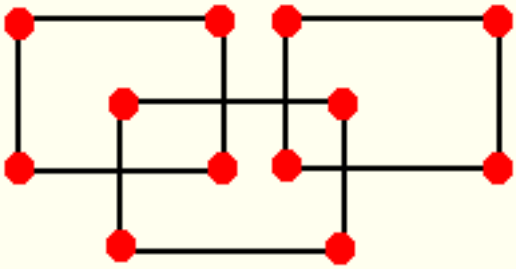


component

See connected.

connected

A graph is connected if there is a path connecting every pair of vertices. A graph that is not connected can be divided into **connected components** (disjoint connected subgraphs). For example, this graph is made of three connected components.

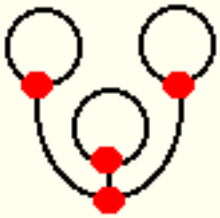


cut vertex

A cut vertex is a vertex that if removed (along with all edges incident with it) produces a graph with more connected components than the original graph. See connected.

degree

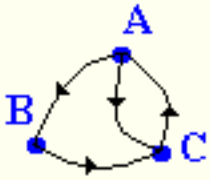
The degree (or valence) of a vertex is the number of edge *ends* at that vertex. For example, in this graph all of the vertices have degree three.



In a digraph (directed graph) the degree is usually divided into the in-degree and the out-degree (whose sum is the degree of the vertex in the underlying undirected graph).

digraph

A digraph (or a **directed graph**) is a graph in which the edges are directed. (Formally: a digraph is a (usually finite) set of vertices V and set of *ordered* pairs (*a*,*b*) (where *a*, *b* are in V) called edges. The vertex *a* is the **initial vertex** of the edge and *b* the **terminal vertex**.



directed graph

See digraph.

edge

See graph.

Four Color Theorem

Every planar graph can be colored using no more than four colors.

graph

Informally, a graph is a finite set of dots called **vertices** (or **nodes**) connected by links called **edges** (or **arcs**). More formally: a **simple graph** is a (usually finite) set of vertices V and set of unordered pairs of distinct elements of V called edges.

Not all graphs are simple. Sometimes a pair of vertices are connected by multiple edge yielding a multigraph. At times vertices are even connected to themselves by a edge called a loop, yeilding a pseudograph. Finally, edges can also be given a direction yielding a directed graph (or digraph).

in-degree

The in-degree of a vertex *v* is the number of edges with *v* as their terminal vertex. See also digraph and degree.

initial vertex

See digraph.

isolated

A vertex of degree zero (with no edges connected) is isolated..

Kuratowski's Theorem

A graph is nonplanar if and only if it contains a subgraph homeomorphic to K3,3 or K5.

length

For the length of a path see path.

loop

A loop is an edge that connects a vertex to itself. (See the illustration for degree which has a graph with three loops.) See pseudograph for a formal definition of loop.

multigraph

Informally, a multigraph is a graph with multiple edges between the same vertices. Formally: a **multigraph** is a set V of **vertices** along, a set E of **edges**, and a function *f* from E to {{*u*,*v*}|*u*,*v* in V; u,v distinct}. (The function *f* shows which vertices are connected by which edge.) The edges *r* and *s* are called **parallel** or **multiple** edges if *f*(*r*)=*f*(*s*). See also graph and pseudograph.

multiple edge

See multigraph.

node

A synonym for vertex. See graph.

out-degree

The out-degree of a vertex *v* is the number of edges with *v* as their initial vertex. See also digraph and degree.

parallel edge

See multigraph.

path

A path is a sequence of consecutive edges in a graph and the length of the path is the number of edges traversed. (This illustration shows a path of length four.)

pendant

A vertex of degree one (with only one edge connected) is a pendant edge..

planar

A graph is planar if it can be drawn on a plane so that the edges intersect only at the vertices. (For example, of the five first complete graphs all but the fifth, K5, is planar.)

pseudograph

Informally, a pseudograph is a graph with multiple edges (or loops) between the same vertices (or the same vertex). Formally: a **pseudograph** is a set V of **vertices** along, a set E of **edges**, and a function *f* from E to {{*u*,*v*}|*u*,*v* in V}. (The function *f* shows which vertices are connected by which edge.) An edge is a **loop** if *f*(*e*) = {*u*} for some vertex *u* in V. See also graph and multigraph.

terminal vertex

See digraph.

undirected edge

Edges in graphs are undirected (as opposed to those in digraphs).

valence

See degree.

vertex

See graph.

If you came to this page from a Web Tutor tutorial, use the back button on your browser to return.

[UT Martin | Back Door | Quick Guide]

Chris Caldwell *caldwell@utm.edu*