CHUYÊN ĐỀ

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 Sĩ 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, 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.

§1. CÁC KHÁI NIỆM CƠ BẢN

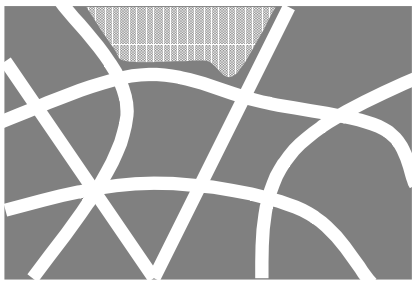
I. ĐỊNH NGHĨA ĐỒ THỊ (GRAPH)

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

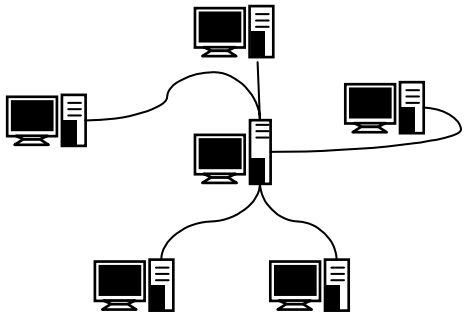
$$G = (V, E)$$

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

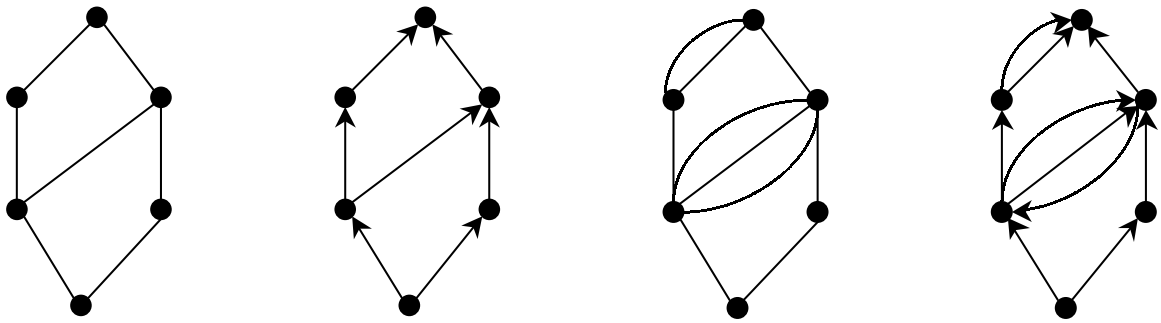
Hình 1: Ví dụ về mô hình đồ thị

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

Hình 2: Phân loại 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:

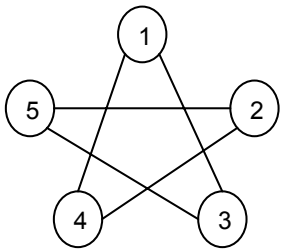
- $a_{ij} = 1$ nếu $(i, j) \in E$
- $a_{ij} = 0$ nếu $(i, j) \notin E$
- Quy ước $a_{ii} = 0$ với $\forall i$;

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

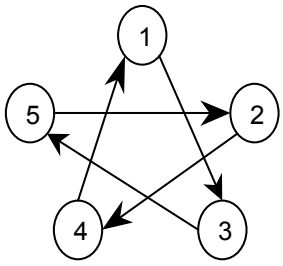
Ví dụ:

	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 = $deg^-(i)$

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: $a_{uv} \neq 0$.

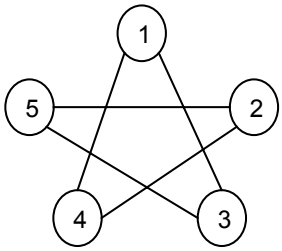
Nhược điểm của ma trận liền kề:

- 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

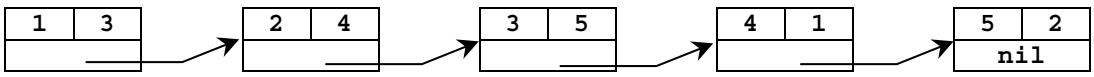
Trong trường hợp đồ thị có n đỉnh, m cạnh, ta có thể biểu diễn đồ thị dưới dạng danh sách cạnh, trong cách biểu diễn này, người ta liệt kê tất cả các cạnh của đồ thị trong một danh sách, mỗi phần tử của danh sách là một cặp (u, v) tương ứng với một cạnh của đồ thị. (Trong trường hợp đồ thị có hướng thì 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	2	3	4	5
(1, 3)	(2, 4)	(3, 5)	(4, 1)	(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)

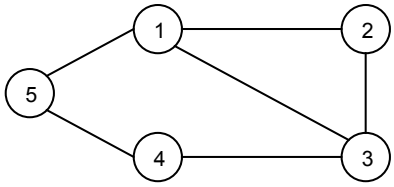
Nhược điểm của danh sách cạnh:

- 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:



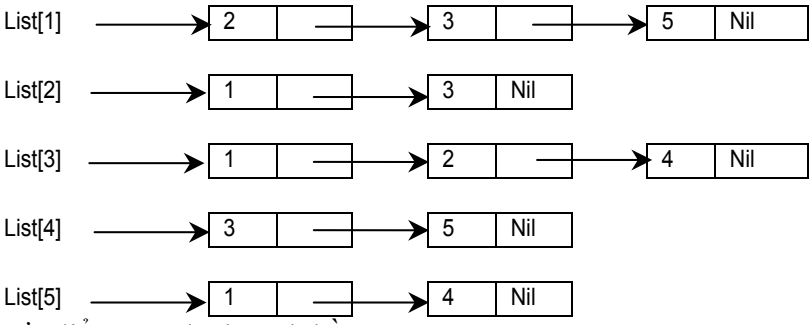
Cách 1: (Forward Star) Dùng một mảng các đỉnh, mảng đó chia làm n đoạn, đoạn thứ i trong mảng lưu danh sách các đỉnh kề với đỉnh i: Ví dụ với đồ thị sau, danh sách kề sẽ là một mảng A gồm 12 phần tử:

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. Ta gọi mảng lưu vị trí đó là mảng Head. Head[i] sẽ bằng chỉ số đứng liền trước đoạn thứ i. Quy ước Head[n + 1] sẽ bằng m. Với đồ thị bên thì mảng VT[1..6] sẽ là: (0, 3, 5, 8, 10, 12)

Như vậy đoạn từ vị trí Head[i] + 1 đến Head[i + 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 Integer; {Ma trận kề của đồ thị}
  n, i, j: Integer;
begin
  Write('Number of vertices'); ReadLn(n);
  FillChar(A, SizeOf(A), 0);
  repeat
    Write('Enter edge (i, j) (i = 0 to exit) ');
    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

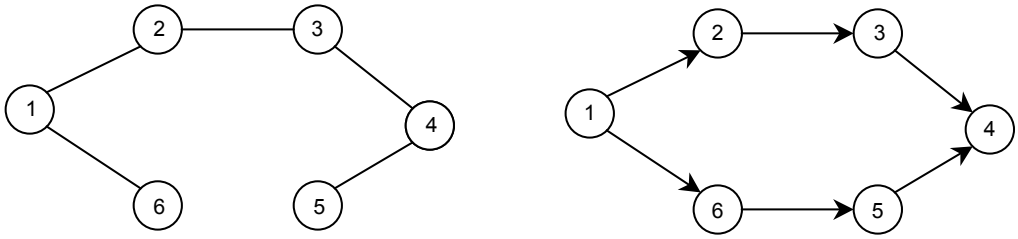
I. BÀI TOÁN

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:

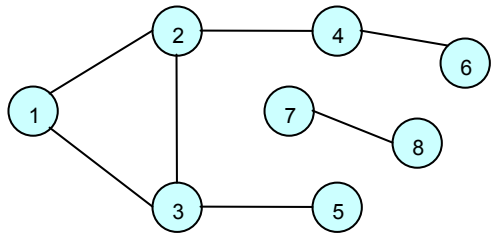


Trên cả hai đồ thị, (1, 2, 3, 4) là đường đi đơn độ dài 3 từ đỉnh 1 tới đỉnh 4. Bởi (1, 2) (2, 3) và (3, 4) đều là các cạnh (hay cung). (1, 6, 5, 4) không phải đường đi bởi (6, 5) không phải là cạnh (hay cung).

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 chúng.

Lưu ý:

1. 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.
2. Dữ liệu về đồ thị sẽ được nhập từ file văn bản GRAPH.INP. Trong đó:
   - Dòng 1 chứa số đỉnh n (≤ 100), số cạnh m của đồ thị, đỉnh xuất phát S, đỉnh kết thúc F cách nhau một 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.
3. Kết quả ghi ra file văn bản GRAPH.OUT
   - Dòng 1: Ghi danh sách các đỉnh có thể đến được từ S
   - Dòng 2: Đường đi từ S tới F được in ngược theo chiều từ F về S



GRAPH.INP	GRAPH.OUT
8 7 1 5 1 2 1 3 2 3 2 4 3 5 4 6 7 8	1, 2, 3, 5, 4, 6, 5<-3<-2<-1

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

1. Cài đặt đệ quy

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 thăm đỉnh u và tiếp tục quá trình duyệt DFS(v) với v là một đỉnh chưa thăm 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:

$$F \leftarrow p_1 = Trace[F] \leftarrow p_2 = Trace[p_1] \leftarrow ... \leftarrow S.$$

```
procedure DFS(u∈V);
begin
  < 1. Thông báo tới được u >;
  < 2. Đánh dấu u là đã thăm (có thể tới được từ S)>;
  < 3. Xét mọi đỉnh v kề với u mà chưa thăm, 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.
```

PROG03_1.PAS * Thuật toán tìm kiếm theo chiều sâu

```
program Depth_First_Search_1;
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[v] = True ⇔ v chưa được thăm đến}
  Trace: array[1..max] of Integer;         {Trace[v] = đỉnh liền trước v trên đường đi từ S tới v}
  n, S, F: Integer;
```

```
procedure Enter;     {Nhập dữ liệu từ thiết bị nhập chuẩn (Input)}
var
  i, u, v, m: Integer;
begin
  FillChar(a, SizeOf(a), False);     {Khởi tạo đồ thị chưa có cạnh nào}
  ReadLn(n, m, S, F);                {Đọc dòng 1 ra 4 số n, m, S và F}
  for i := 1 to m do                 {Đọc m dòng tiếp ra danh sách cạnh}
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
end;

procedure DFS(u: Integer);           {Thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh u}
var
  v: Integer;
begin
  Write(u, ', ');                    {Thông báo tới được u}
  Free[u] := False;                  {Đánh dấu u đã thăm}
  for v := 1 to n do
    if Free[v] and a[u, v] then      {Với mỗi đỉnh v chưa thăm kề với u}
      begin
        Trace[v] := u;               {Lưu vết đường đi: Đỉnh liền trước v trong đường đi từ S tới v là u}
        DFS(v);                      {Tiếp tục tìm kiếm theo chiều sâu bắt đầu từ v}
      end;
end;

procedure Result;          {In đường đi từ S tới F}
begin
  WriteLn;                 {Vào dòng thứ hai của Output file}
  if Free[F] then          {Nếu F chưa đánh dấu thăm tức là không có đường}
    WriteLn('Path from ', S, ' to ', F, ' not found')
  else                     {Truy vết đường đi, bắt đầu từ F}
    begin
      while F <> S do
        begin
          Write(F, '<-');
          F := Trace[F];
        end;
      WriteLn(S);
    end;
end;

begin
  {Định nghĩa lại thiết bị nhập/xuất chuẩn thành Input/Output file}
  Assign(Input, 'GRAPH.INP'); Reset(Input);
  Assign(Output, 'GRAPH.OUT'); Rewrite(Output);
  Enter;
  FillChar(Free, n, True);
  DFS(S);
  Result;
  {Đóng Input/Output file, thực ra không cần vì BP tự động đóng thiết bị nhập xuất chuẩn trước khi kết thúc chương trình}
  Close(Input);
  Close(Output);
end.
```

Chú ý:

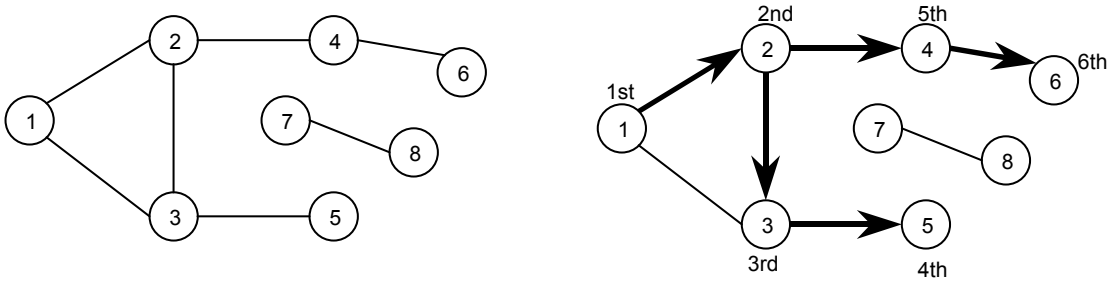
a) Vì có kỹ thuật đánh dấu, nên thủ tục DFS sẽ được gọi ≤ n lần (n là số đỉnh)

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] := -1 (Chỉ là để cho khác 0 thôi).

```
procedure DFS(u: Integer); {Cải tiến}
var
  v: Integer;
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 tìm kiếm DFS sau (Mũi tên u→v chỉ thao tác đệ quy: DFS(u) gọi DFS(v)).



Hình 3: Cây DFS

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.

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

Khi mô tả quá trình đệ quy bằng một ngăn xếp, ta luôn luôn để cho ngăn xếp lưu lại dây chuyền duyệt sâu từ nút gốc (đỉnh xuất phát S).

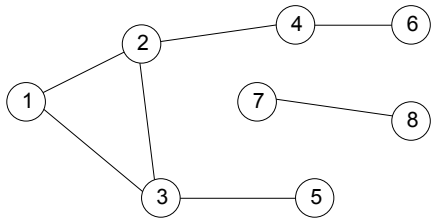
```
<Thăm S, đánh dấu S đã thăm>;
<Đẩy S vào ngăn xếp>;          {Dây chuyền đệ quy ban đầu chỉ có một đỉnh S}
repeat
  <Lấy u khỏi ngăn xếp>;       {Đang đứng ở đỉnh u}
  if <u có đỉnh kề chưa thăm> then
    begin
      <Chỉ chọn lấy 1 đỉnh v, là đỉnh đầu tiên kề u mà chưa được thăm>;
      <Thông báo thăm v>;
      <Đẩy u trở lại ngăn xếp>;          {Giữ lại địa chỉ quay lui}
      <Đẩy tiếp v vào ngăn xếp>;         {Dây chuyền duyệt sâu được "nối" thêm v nữa}
    end;
  {Còn nếu u không có đỉnh kề chưa thăm thì ngăn xếp sẽ ngắn lại, tương ứng với quá trình lùi về của dây chuyền DFS}
until <Ngăn xếp rỗng>;
```

PROG03_2.PAS * Thuật toán tìm kiếm theo chiều sâu không đệ quy

```
program Depth_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 Integer;
  Stack: array[1..max] of Integer;
  n, S, F, Last: Integer;

procedure Enter;     {Nhập dữ liệu (từ thiết bị nhập chuẩn)}
var
  i, u, v, m: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m, S, F);
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
end;

procedure Init;  {Khởi tạo}
begin
  FillChar(Free, n, True);    {Các đỉnh đều chưa đánh dấu}
  Last := 0;                  {Ngăn xếp rỗng}
end;

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

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

procedure DFS;
var
  u, v: Integer;
begin
  Write(S, ', '); Free[S] := False;    {Thăm S, đánh dấu S đã thăm}
  Push(S);                             {Khởi động dây chuyền duyệt sâu}
  repeat
    {Dây chuyền duyệt sâu đang là S→ ...→ u}
    u := Pop;                          {u là điểm cuối của dây chuyền duyệt sâu hiện tại}
    for v := 1 to n do
      if Free[v] and a[u, v] then      {Chọn v là đỉnh đầu tiên chưa thăm kề với u, nếu có:}
        begin
          Write(v, ', '); Free[v] := False;   {Thăm v, đánh dấu v đã thăm}
          Trace[v] := u;                      {Lưu vết đường đi}
          Push(u); Push(v);                   {Dây chuyền duyệt sâu bây giờ là S→ ...→ u→ v}
          Break;
        end;
  until Last = 0;  {Ngăn xếp rỗng}
end;
```

```
procedure Result;      {In đường đi từ S tới F}
begin
  WriteLn;
  if Free[F] then
    WriteLn('Path from ', S, ' to ', F, ' not found')
  else
    begin
      while F <> S do
        begin
          Write(F, '<-');
          F := Trace[F];
        end;
      WriteLn(S);
    end;
end;

begin
  Assign(Input, 'GRAPH.INP'); Reset(Input);
  Assign(Output, 'GRAPH.OUT'); Rewrite(Output);
  Enter;
  Init;
  DFS;
  Result;
  Close(Input);
  Close(Output);
end.
```

Ví dụ: Với đồ thị dưới đây (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 được thăm với thứ tự từ 1st đến 6th trong cây tìm kiếm của thủ tục DFS dùng đệ quy.



Trước hết ta thăm đỉnh 1 và đẩy nó vào ngăn xếp.

Bước lặp	Ngăn xếp	u	**v**	Ngăn xếp sau mỗi bước	Giải thích
1	(1)	1	**2**	(1, 2)	Tiến sâu xuống thăm 2
2	(1, 2)	2	**3**	(1, 2, 3)	Tiến sâu xuống thăm 3
3	(1, 2, 3)	3	**5**	(1, 2, 3, 5)	Tiến sâu xuống thăm 5
4	(1, 2, 3, 5)	5	**Không có**	(1, 2, 3)	Lùi lại
5	(1, 2, 3)	3	**Không có**	(1, 2)	Lùi lại
6	(1, 2)	2	**4**	(1, 2, 4)	Tiến sâu xuống thăm 4
7	(1, 2, 4)	4	**6**	(1, 2, 4, 6)	Tiến sâu xuống thăm 6
8	(1, 2, 4, 6)	6	**Không có**	(1, 2, 4)	Lùi lại
9	(1, 2, 4)	4	**Không có**	(1, 2)	Lùi lại
10	(1, 2)	2	**Không có**	(1)	Lùi lại
11	(1)	1	**Không có**	∅	Lùi hết dây chuyền, Xong

Trên đây là phương pháp dựa vào tính chất của thủ tục đệ quy để tìm ra phương pháp mô phỏng nó. Tuy nhiên, trên mô hình đồ thị thì ta có thể có một cách viết khác tốt hơn cũng không đệ quy: Thử nhìn lại cách thăm đỉnh của DFS: Từ một đỉnh u, chọn lấy một đỉnh v kề nó mà chưa thăm rồi tiến sâu xuống thăm v. Còn nếu mọi đỉnh kề u đều đã thăm thì lùi lại một bước và lặp lại quá trình tương

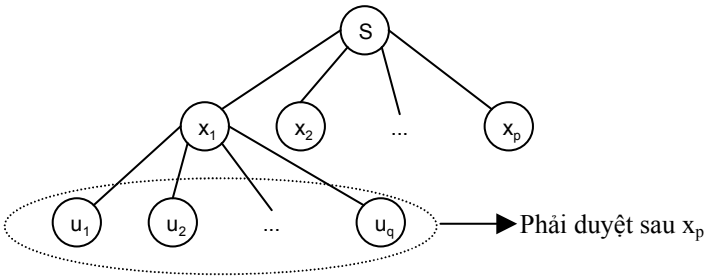
tự, việc lùi lại này có thể thực hiện dễ dàng mà không cần dùng Stack nào cả, bởi với mỗi đỉnh u đã có một nhãn Trace[u] (là đỉnh mà đã từ đó mà ta tới thăm u), khi quay lui từ u sẽ lùi về đó.
Vậy nếu ta đang đứng ở đỉnh u, thì đỉnh kế tiếp phải thăm tới sẽ được tìm như trong hàm FindNext dưới đây:

```
function FindNext(u∈V): ∈V;      {Tìm đỉnh sẽ thăm sau đỉnh u, trả về 0 nếu mọi đỉnh tới được từ S đều đã thăm}
begin
  repeat
    for (∀v ∈ Kề(u)) do
      if <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. Lưu ý là Trace[S] được gán bằng n + 1}
  until u = n + 1;
  FindNext := 0; {ở trên không Exit được tức là mọi đỉnh tới được từ S đã duyệt xong}
end;

begin          {Thuật toán duyệt theo chiều sâu}
  Trace[S] := n + 1;
  u := S;
  repeat
    <Thông báo thăm u, đánh dấu u đã thăm>;
    u := FindNext(u);
  until u = 0;
end;
```

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

1. Cài đặt bằng hàng đợi

Cơ sở của phương pháp cài đặt này 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 chiều rộng (đỉnh nào gần S hơn sẽ được duyệt trước). Ví dụ: Bắt đầu ta 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).



Hình 4: Cây BFS

Giả sử ta có một danh sách chứa những đỉnh đang "chờ" thăm. Tại mỗi bước, ta thăm một đỉnh đầu danh sách và cho những đỉnh chưa "xếp hàng" kề với nó xếp hàng thêm vào cuối danh sách. Chính vì nguyên tắc đó nên danh sách chứa những đỉnh đang chờ sẽ được tổ chức dưới dạng hàng đợi (Queue)

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. 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 hàng đợi (v sẽ chờ được duyệt tại những bước sau)

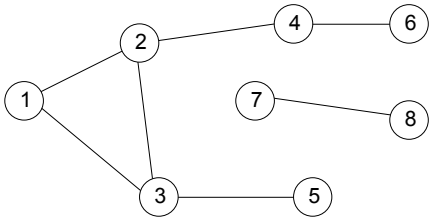
Bước 3: Truy vết tìm đường đi.

PROG03_3.PAS * Thuật toán tìm kiếm theo chiều rộng dùng hàng đợi

```
program Breadth_First_Search_1;
const
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  Free: array[1..max] of Boolean;    {Free[v] ⇔ v chưa được xếp vào hàng đợi để chờ thăm}
  Trace: array[1..max] of Integer;
  Queue: array[1..max] of Integer;
  n, S, F, First, Last: Integer;

procedure Enter;    {Nhập dữ liệu}
var
  i, u, v, m: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m, S, F);
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
end;

procedure Init;     {Khởi tạo}
begin
  FillChar(Free, n, True);    {Các đỉnh đều chưa đánh dấu}
  Free[S] := False;           {Ngoại trừ đỉnh S}
  Queue[1] := S;              {Hàng đợi chỉ gồm có một đỉnh S}
  Last := 1;
  First := 1;
end;

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

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

procedure BFS;    {Thuật toán tìm kiếm theo chiều rộng}
var
```

```
  u, v: Integer;
begin
  repeat
    u := Pop;                {Lấy một đỉnh u khỏi hàng đợi}
    Write(u, ', ');          {Thông báo thăm u}
    for v := 1 to n do
      if Free[v] and a[u, v] then    {Xét những đỉnh v chưa đánh dấu kề u}
        begin
          Push(v);                   {Đưa v vào hàng đợi để chờ thăm}
          Free[v] := False;          {Đánh dấu v}
          Trace[v] := u;             {Lưu vết đường đi: đỉnh liền trước v trong đường đi từ S là u}
        end;
  until First > Last;        {Cho tới khi hàng đợi rỗng}
end;

procedure Result;            {In đường đi từ S tới F}
begin
  WriteLn;
  if Free[F] then
    WriteLn('Path from ', S, ' to ', F, ' not found')
  else
    begin
      while F <> S do
        begin
          Write(F, '<-');
          F := Trace[F];
        end;
      WriteLn(S);
    end;
end;

begin
  Assign(Input, 'GRAPH.INP'); Reset(Input);
  Assign(Output, 'GRAPH.OUT'); Rewrite(Output);
  Enter;
  Init;
  BFS;
  Result;
  Close(Input);
  Close(Output);
end.
```

Ví dụ: Xét đồ thị dưới đây, Đỉnh xuất phát S = 1.

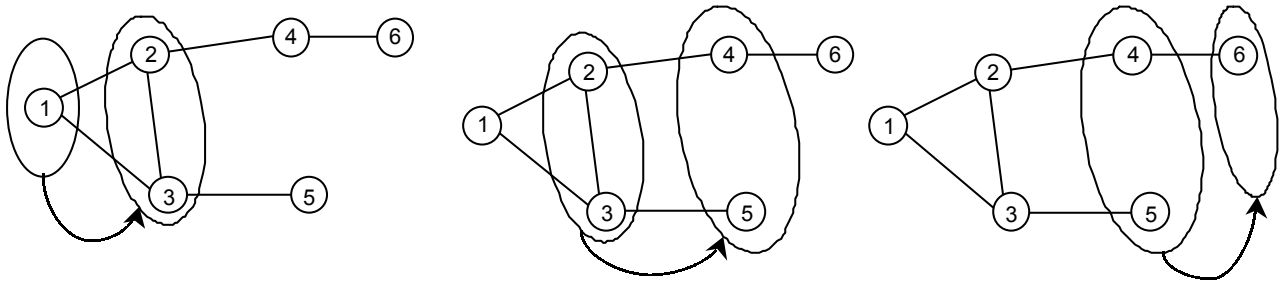


Hàng đợi	**Đỉnh u (lấy ra từ hàng đợi)**	Hàng đợi (sau khi lấy u ra)	Các đỉnh v kề u mà chưa lên lịch	Hàng đợi sau khi đẩy những đỉnh v vào
(1)	**1**	∅	2, 3	(2, 3)
(2, 3)	**2**	(3)	4	(3, 4)
(3, 4)	**3**	(4)	5	(4, 5)
(4, 5)	**4**	(5)	6	(5, 6)
(5, 6)	**5**	(6)	Không có	(6)
(6)	**6**	∅	Không có	∅

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:



Hình 5: Thuật toán loang

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 và lặp lại (có thể luân phiên vai trò hai tập này)

Bước 3: Truy vết tìm đường đi.

PROG03_4.PAS * Thuật toán tìm kiếm theo chiều rộng dùng phương pháp loang

```
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 Integer;
  Old, New: set of Byte;
  n, S, F: Byte;

procedure Enter;    {Nhập dữ liệu}
var
  i, u, v, m: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m, S, F);
```

```
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
end;

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 Old, 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}
end;

procedure Result;
begin
  WriteLn;
  if Free[F] then
    WriteLn('Path from ', S, ' to ', F, ' not found')
  else
    begin
      while F <> S do
        begin
          Write(F, '<-');
          F := Trace[F];
        end;
      WriteLn(S);
    end;
end;

begin
  Assign(Input, 'GRAPH.INP'); Reset(Input);
  Assign(Output, 'GRAPH.OUT'); Rewrite(Output);
  Enter;
  Init;
  BFS;
  Result;
  Close(Input);
  Close(Output);
end.
```

IV. ĐỘ PHỨC TẠP TÍNH TOÁN CỦA BFS VÀ DFS

Quá trình tìm kiếm trên đồ thị bắt đầu từ một đỉnh có thể thăm tất cả các đỉnh còn lại, khi đó cách biểu diễn đồ thị có ảnh hưởng lớn tới chi phí về thời gian thực hiện giải thuật:

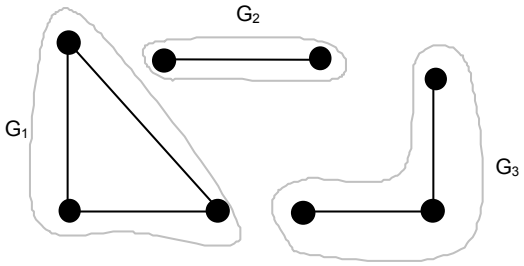
- Trong trường hợp ta biểu diễn đồ thị bằng danh sách kề, cả hai thuật toán BFS và DFS đều có độ phức tạp tính toán là O(n + m) = O(max(n, m)). Đây là cách cài đặt tốt nhất.
- Nếu ta biểu diễn đồ thị bằng ma trận kề như ở trên thì độ phức tạp tính toán trong trường hợp này là $O(n + n^2) = O(n^2)$.
- Nếu ta biểu diễn đồ thị bằng danh sách cạnh, thao tác duyệt những đỉnh kề với đỉnh u sẽ dẫn tới việc phải duyệt qua toàn bộ danh sách cạnh, đây là cài đặt tồi nhất, nó có độ phức tạp tính toán là O(n.m).

§4. TÍNH LIÊN THÔNG CỦA ĐỒ THỊ

I. ĐỊNH NGHĨA

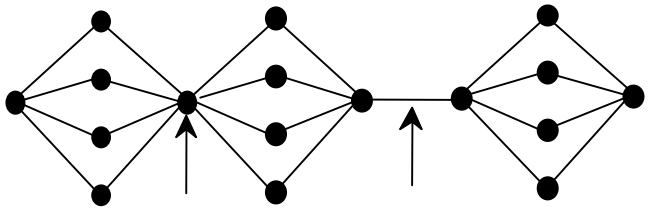
1. Đối với đồ thị vô hướng G = (V, E)

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* 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).



Hình 6: 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.**

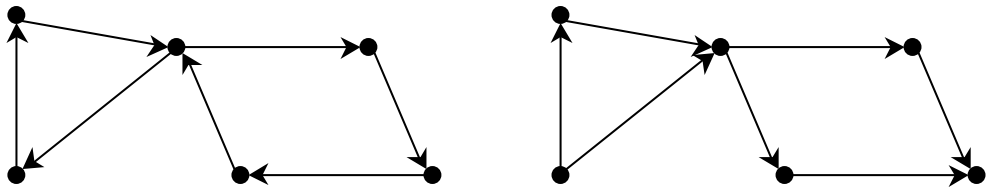


Hình 7: Khớp và cầu

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



Hình 8: Liên thông mạnh và Liên thông yếu

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

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;
  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.
```

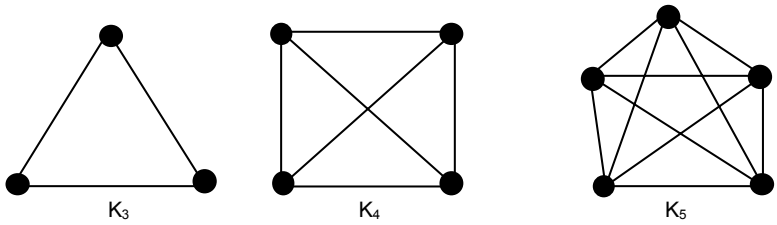
Với thuật toán liệt kê các thành phần liên thông như thế này, thì độ phức tạp tính toán của nó đúng bằng độ phức tạp tính toán của thuật toán tìm kiếm trên đồ thị trong thủ tục Duyệt.

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.



Hình 9: Đồ thị đầy đủ

2. Bao đóng đồ thị:

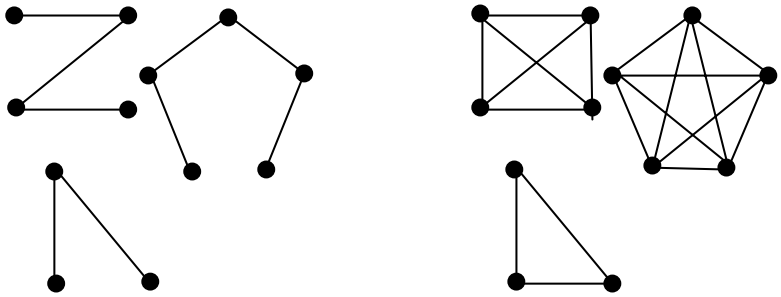
Với đồ thị G = (V, E), người ta xây dựng đồ thị G' = (V, E') cũng gồm những đỉnh của G còn các cạnh xây dựng như sau: (ở đây quy ước giữa u và u luôn có đường đi)

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.



Hình 10: 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.

Với n là số đỉnh của đồ thị, ta có thể viết thuật toán Warshall như sau:

```
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];
```

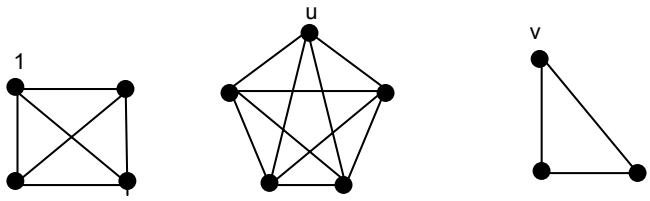
Việc chứng minh tính đúng đắn của thuật toán đòi hỏi phải lật lại các lý thuyết về bao đóng bắc cầu và quan hệ liên thông, ta sẽ không trình bày ở đây. Có nhận xét rằng tuy thuật toán Warshall rất dễ cài đặt nhưng độ phức tạp tính toán của thuật toán này khá lớn ($O(n^3)$).

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

Việc cài đặt thuật toán sẽ qua những bước sau:

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...

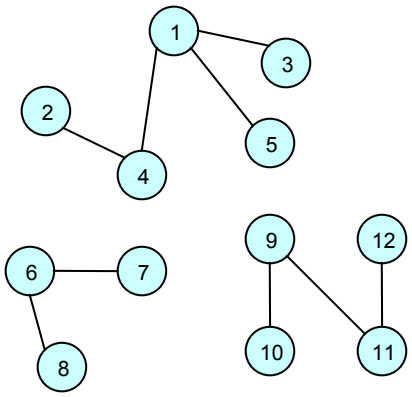


Input: file văn bản GRAPH.INP

- Dòng 1: Chứa số đỉnh n (≤ 100) và số cạnh m của đồ thị cách nhau ít nhất một dấu cách
- m dòng tiếp theo, mỗi dòng chứa một cặp số u và v cách nhau ít nhất một dấu cách tượng trưng cho một cạnh (u, v)

Output: file văn bản GRAPH.OUT

- Liệt kê các thành phần liên thông



GRAPH.INP	GRAPH.OUT
12 9 1 3 1 4 1 5 2 4 6 7 6 8 9 10 9 11 11 12	Connected Component 1: 1, 2, 3, 4, 5, Connected Component 2: 6, 7, 8, Connected Component 3: 9, 10, 11, 12,

PROG04_1.PAS * Thuật toán Warshall liệt kê các thành phần liên thông

```
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;         {Free[v] = True ⇔ v chưa được liệt kê vào thành phần liên thông nào}
  k, u, v, n: Integer;
  Count: Integer;

procedure Enter;                          {Nhập đồ thị}
var
  i, u, v, m: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m);
  for v := 1 to n do a[v, v] := True;     {Dĩ nhiên từ v có đường đi đến chính v}
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
```

```
end;

begin
  Assign(Input, 'GRAPH.INP'); Reset(Input);
  Assign(Output, 'GRAPH.OUT'); Rewrite(Output);
  Enter;
  {Thuật toán Warshall}
  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];
  Count := 0;
  FillChar(Free, n, True);    {Mọi đỉnh đều chưa được liệt kê vào thành phần liên thông nào}
  for u := 1 to n do
    if Free[u] then      {Với một đỉnh u chưa được liệt kê vào thành phần liên thông nào}
      begin
        Inc(Count);
        WriteLn('Connected Component ', Count, ': ');
        for v := 1 to n do
          if a[u, v] then          {Xét những đỉnh kề u (trên bao đóng)}
            begin
              Write(v, ', ');      {Liệt kê đỉnh đó vào thành phần liên thông chứa u}
              Free[v] := False;    {Liệt kê đỉnh nào đánh dấu đỉnh đó}
            end;
        WriteLn;
      end;
  Close(Input);
  Close(Output);
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

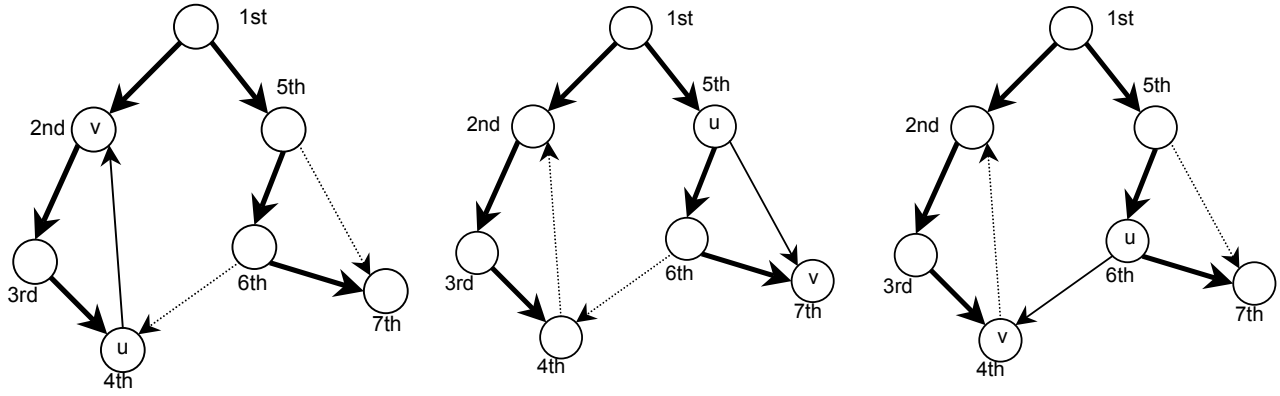
Thêm vào đồ thị một đỉnh x và nối x với tất cả các đỉnh còn lại của đồ thị bằng các cung định hướng. Khi đó quá trình tìm kiếm theo chiều sâu bắt đầu từ x có thể coi như một quá trình xây dựng cây tìm kiếm theo chiều sâu (cây DFS) gốc x.

```
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 DFS đã duyệt trước u
(u, v là cung chéo)

Hình 11: Ba dạng cung ngoài cây DFS

Ta nhận thấy một đặc điểm của thuật toán tìm kiếm theo chiều sâu, thuật toán không chỉ duyệt qua các đỉnh, nó còn duyệt qua tất cả những cung nữa. Ngoài những cung nằm trên cây 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 $x_1, x_2, ..., x_k$ đều thuộc C nên chúng sẽ phải thăm sau đỉnh r. Khi thủ tục Visit(r) được gọi thì tất cả các đỉnh $x_1, x_2..., x_k=v$ 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 $x_1, x_2, ..., x_k = v$ sẽ thuộc nhánh gốc r của cây DFS. Bởi chọn v là đỉnh bất kỳ trong C nên ta có điều phải chứng minh.

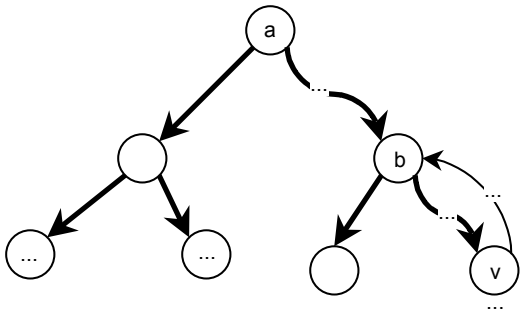
Đỉnh r trong chứng minh định lý - **đỉnh thăm trước tất cả các đỉnh khác trong C** - gọi là **chốt** của thành phần C. Mỗi thành phần liên thông mạnh có duy nhất một chốt. Xét về vị trí trong cây tìm kiếm DFS, chốt của một thành phần liên thông là **đỉnh nằm cao nhất so với các đỉnh khác thuộc thành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

Định lý 3:

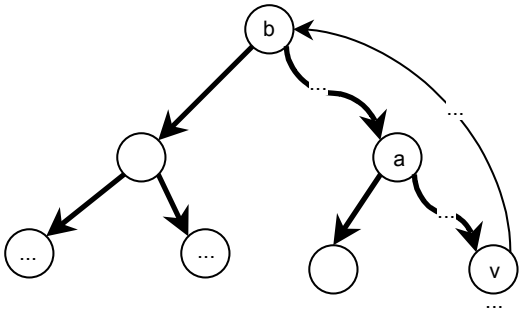
Luôn tìm được đỉnh chốt a thoả mãn: Quá trình tìm kiếm theo chiều sâu bắt đầu từ a không thăm được bất kỳ một chốt nào khác. (Tức là nhánh DFS gốc a không chứa một chốt nào ngoài a) chẳng hạn ta chọn a là chốt được thăm sau cùng trong một dây chuyền đệ quy hoặc chọn a là chốt thăm sau tất cả các chốt 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 phải 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



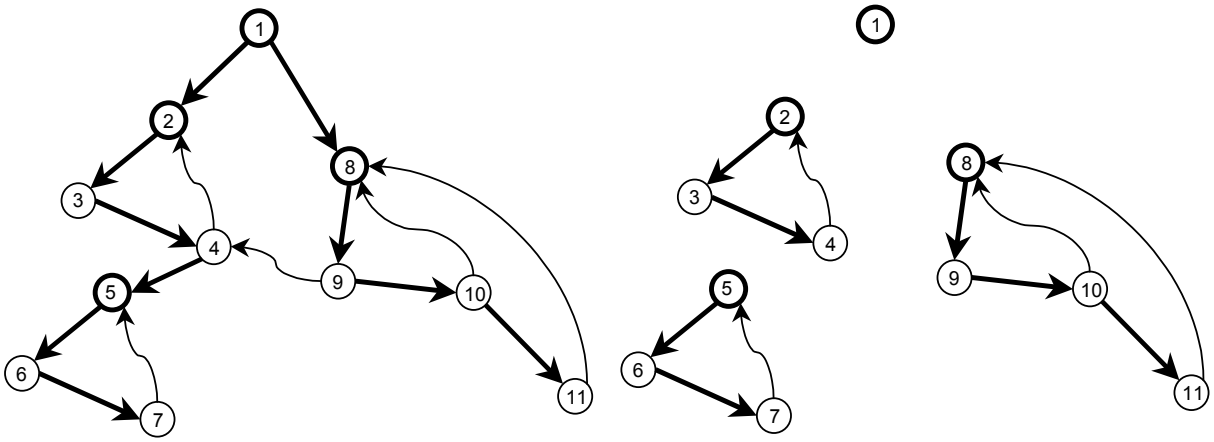
Khả năng 1: b → a → v

- 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.



Hình 12: 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 $\subset$ 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** và quá trình thăm đệ quy lùi lại về Visit(u). 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, v') đi từ v $\in$ nhánh DFS gốc r tới v' $\notin$ nhánh DFS gốc r, gọi r' là chốt của thành phần liên thông chứa v'. Theo tính chất trên, v' phải thăm trước r, suy ra **r' cũng phải thăm trước r**. Có hai khả năng xảy ra:

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

Bởi thuật toán Tarjan chỉ là sửa đổi một chút thuật toán DFS, các thao tác vào/ra ngăn xếp được thực hiện không quá n lần. Vậy nên nếu đồ thị có n đỉnh và m cung thì độ phức tạp tính toán của thuật toán Tarjan vẫn là O(n + m) trong trường hợp biểu diễn đồ thị bằng danh sách kề, là $O(n^2)$ trong trường hợp biểu diễn bằng ma trận kề và là O(n.m) trong trường hợp biểu diễn bằng danh sách cạnh.

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

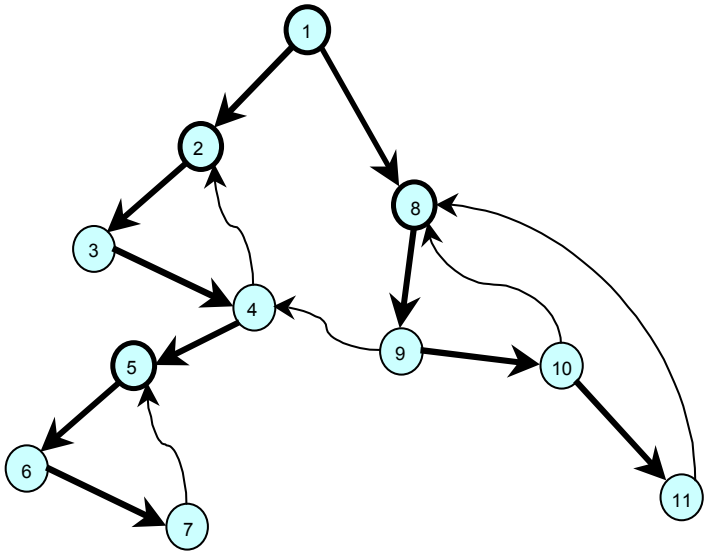
- Ma trận kề A để biểu diễn đồ thị.
- Mảng Free kiểu Boolean, Free[u] = True nếu u chưa bị liệt kê vào thành phần liên thông nào, tức là u chưa bị loại khỏi đồ thị.
- Mảng Numbering và Low với công dụng như trên, quy ước Numbering[u] = 0 nếu đỉnh u chưa được thăm.
- Mảng Stack, thủ tục Push, hàm Pop để mô tả cấu trúc ngăn xếp.

Input: file văn bản GRAPH.INP:

- Dòng đầu: Ghi số đỉnh n (≤ 100) 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

Output: file văn bản GRAPH.OUT

Liệt kê các thành phần liên thông mạnh



GRAPH.INP	GRAPH.OUT
11 15 1 2 1 8 2 3 3 4 4 2 4 5 5 6 6 7 7 5 8 9 9 4 9 10 10 8 10 11 11 8	Component 1: 7, 6, 5, Component 2: 4, 3, 2, Component 3: 11, 10, 9, 8, Component 4: 1,

PROG04_2.PAS * Thuật toán Tarjan liệt kê các thành phần liên thông mạnh

```
program Strong_connectivity;
const
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  Free: array[1..max] of Boolean;
  Numbering, Low, Stack: array[1..max] of Integer;
  n, Count, ComponentCount, Last: Integer;

procedure Enter;          {Nhập dữ liệu (từ thiết bị nhập chuẩn)}
var
  i, u, v, m: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m);
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
    end;
end;

procedure Init;     {Khởi tạo}
begin
  FillChar(Numbering, SizeOf(Numbering), 0);     {Mọi đỉnh đều chưa thăm}
  FillChar(Free, SizeOf(Free), True);            {Chưa đỉnh nào bị loại}
  Last := 0;                                     {Ngăn xếp rỗng}
  Count := 0;                                    {Biến đánh số thứ tự thăm}
  ComponentCount := 0;                           {Biến đánh số các thành phần liên thông}
end;

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

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

```
function Min(x, y: Integer): Integer;
begin
  if x < y then Min := x else Min := y;
end;

procedure Visit(u: Integer);        {Thuật toán tìm kiếm theo chiều sâu bắt đầu từ u}
var
  v: Integer;
begin
  Inc(Count); Numbering[u] := Count;  {Trước hết đánh số cho u}
  Low[u] := Numbering[u]; {Coi u có cung tới u, nên có thể khởi gán Low[u] thế này rồi sau cực tiểu hoá dần}
  Push(u);                  {Đẩy u vào ngăn xếp}
  for v := 1 to n do
    if Free[v] and a[u, v] then   {Xét những đỉnh v kề u}
      if Numbering[v] <> 0 then   {Nếu v đã thăm}
        Low[u] := Min(Low[u], Numbering[v])   {Cực tiểu hoá Low[u] theo công thức này}
      else                        {Nếu v chưa thăm}
        begin
          Visit(v);               {Tiếp tục tìm kiếm theo chiều sâu bắt đầu từ 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    {Liệt kê thành phần liên thông mạnh có chốt u}
      Inc(ComponentCount);
      WriteLn('Component ', ComponentCount, ': ');
      repeat
        v := Pop;           {Lấy dần các đỉnh ra khỏi ngăn xếp}
        Write(v, ', ');     {Liệt kê các đỉnh đó}
        Free[v] := False;   {Rồi loại luôn khỏi đồ thị}
      until v = u;          {Cho tới khi lấy tới đỉnh u}
      WriteLn;
    end;
end;

procedure Solve;
var
  u: Integer;
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 mỗi một đỉnh giả đó}
  for u := 1 to n do
    if Numbering[u] = 0 then Visit(u);
end;

begin
  Assign(Input, 'GRAPH.INP'); Reset(Input);
  Assign(Output, 'GRAPH.OUT'); Rewrite(Output);
  Enter;
  Init;
  Solve;
  Close(Input);
  Close(Output);
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 hay DFS) 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.

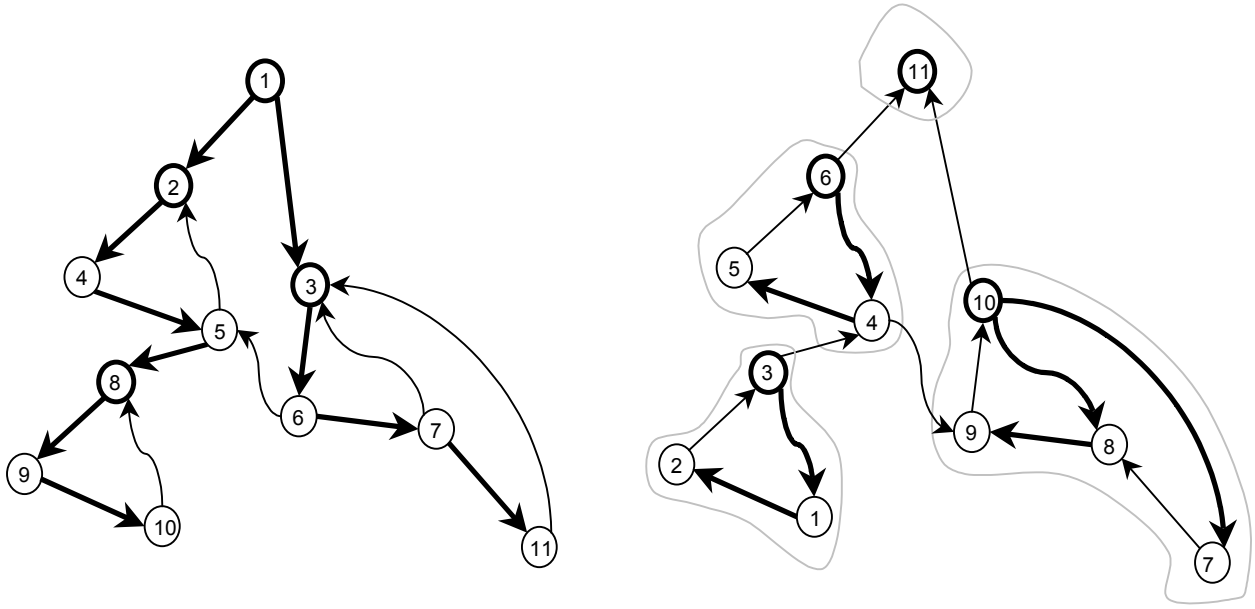
Tính đúng đắn của phương pháp có thể hình dung không mấy khó khăn:

Trước hết ta thêm vào đồ thị đỉnh x và các cung (x, v) với mọi v, sau đó gọi Visit(x) để xây dựng cây DFS gốc x. Hiển nhiên x là chốt của thành phần liên thông chỉ gồm mỗi x. Sau đó bỏ đỉnh x khỏi cây DFS, cây sẽ phân rã thành các cây con.

Đỉnh r duyệt xong sau cùng chắc chắn là gốc của một cây con (bởi khi duyệt xong nó chắc chắn sẽ lùi về x) suy ra r là chốt. Hơn thế nữa, nếu một đỉnh u nào đó tới được r thì u cũng phải thuộc cây con gốc r. Bởi nếu giả sử phản chứng rằng u thuộc cây con khác thì u phải được thăm trước r (do cây con gốc r được thăm tới sau cùng), có nghĩa là khi Visit(u) thì r chưa thăm. Vậy nên r sẽ thuộc nhánh DFS gốc u, mâu thuẫn với lập luận r là gốc. Từ đó suy ra nếu u tới được r thì r tới được u, tức là khi đảo chiều các cung, nếu r tới được đỉnh nào thì đỉnh đó thuộc thành phần liên thông chốt r.

Loại bỏ thành phần liên thông với chốt r khỏi đồ thị. Cây con gốc r lại phân rã thành nhiều cây con. Lập luận tương tự như trên với v' là đỉnh duyệt xong sau cùng.

Ví dụ:



Đánh số lại, đảo chiều các cung và duyệt BFS với cách chọn các đỉnh xuất phát ngược lại với thứ tự duyệt xong (Thứ tự 11, 10... 3, 2, 1)

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. 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. VÀI ỨNG DỤNG CỦA CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

I. XÂY DỰNG CÂY KHUNG CỦA ĐỒ THỊ

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

1. Định lý (Daisy Chain Theorem)

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

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

Chứng minh:

1⇒2: **"T là cây" ⇒ "T không chứa chu trình đơn và có n - 1 cạnh"**

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

2⇒3: **"T không chứa chu trình đơn và có n - 1 cạnh"⇒"T liên thông và mỗi cạnh của nó đều là cầu"**

Giả sử T có k thành phần liên thông T_1, T_2, ..., T_k. Vì T không chứa chu trình đơn nên các thành phần liên thông của T cũng không chứa chu trình đơn, tức là các T_1, T_2, ..., T_k đều là cây. Gọi n_1, n_2, ..., n_k lần lượt là số đỉnh của T_1, T_2, ..., T_k thì cây T_1 có n_1 - 1 cạnh, cây T_2 có n_2 - 1 cạnh..., cây T_k có n_k - 1 cạnh. Cộng lại ta có số cạnh của T là $n_1 + n_2 + ... + n_k - k = n - k$ cạnh. Theo giả thiết, cây T có n - 1 cạnh, suy ra k = 1, đồ thị chỉ có một thành phần liên thông là đồ thị liên thông.
Bây giờ khi T đã liên thông, nếu bỏ đi một cạnh của T thì T sẽ còn n - 2 cạnh và sẽ không liên thông bởi nếu T vẫn liên thông thì do T không có chu trình nên T sẽ là cây và có n - 1 cạnh. Điều đó chứng tỏ mỗi cạnh của T đều là cầu.

3⇒4: **"T liên thông và mỗi cạnh của nó đều là cầu"⇒"Giữa hai đỉnh bất kỳ của T có đúng một đường đi đơn"**

Gọi x và y là 2 đỉnh bất kỳ trong T, vì T liên thông nên sẽ có một đường đi đơn từ x tới y. Nếu tồn tại một đường đi đơn khác từ x tới y thì nếu ta bỏ đi một cạnh (u, v) nằm trên đường đi thứ nhất nhưng không nằm trên đường đi thứ hai thì từ u vẫn có thể đến được v bằng cách: đi từ u đi theo

chiều tới x theo các cạnh thuộc đường thứ nhất, sau đó đi từ x tới y theo đường thứ hai, rồi lại đi từ y tới v theo các cạnh thuộc đường đi thứ nhất. Điều này mâu thuẫn với giả thiết (u, v) là cầu.

4⇒5: **"Giữa hai đỉnh bất kỳ của T có đúng một đường đi đơn"⇒"T không chứa chu trình đơn nhưng hễ cứ thêm vào một cạnh ta thu được một chu trình đơn"**

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

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

5⇒6: **"T không chứa chu trình đơn nhưng hễ cứ thêm vào một cạnh ta thu được một chu trình đơn"⇒"T liên thông và có n - 1 cạnh"**

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

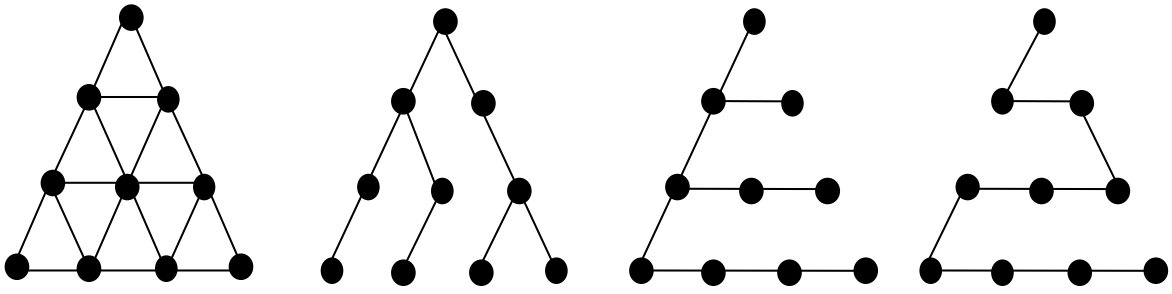
6⇒1: **"T liên thông và có n - 1 cạnh"⇒"T là cây"**

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

2. Định nghĩa

Giả sử G = (V, E) là đồ thị vô hướng. 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.



Hình 13: Đồ thị G và một số ví dụ cây khung T_1, T_2, T_3 của nó

- Điều kiện cần và đủ để một đồ thị vô hướng có cây khung là đồ thị đó phải liên thông
- Số cây khung của đồ thị đầy đủ K_n là n^{n-2}.

3. Thuật toán xây dựng cây khung

Xét đồ thị vô hướng liên thông G = (V, E) có n đỉnh, có nhiều thuật toán xây dựng cây khung của G

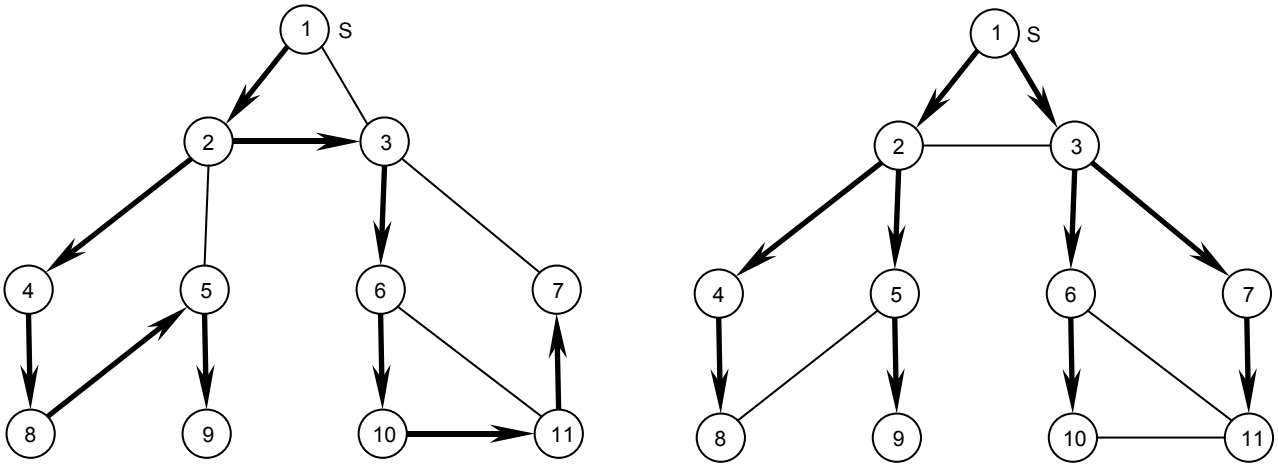
a) Xây dựng cây khung bằng thuật toán hợp nhất

Trước hết, đặt T = (V, ∅); T không chứa cạnh nào thì có thể coi T gồm n cây rời rạc, mỗi cây chỉ có 1 đỉnh. Sau đó xét lần lượt các cạnh của G, nếu cạnh đang xét nối hai cây khác nhau trong T thì thêm cạnh đó vào T, đồng thời hợp nhất hai cây đó lại thành một cây. Cứ làm như vậy cho tới khi kết nạp đủ n - 1 cạnh vào T thì ta được T là cây khung của đồ thị. Các phương pháp kiểm tra cạnh

có nối hai cây khác nhau hay không cũng như kỹ thuật hợp nhất hai cây sẽ được bàn kỹ hơn trong thuật toán Kruskal ở §9.

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

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



Hình 14: Cây khung DFS và cây khung BFS (Mũi tên chỉ chiều đi thăm các đỉnh)

II. TẬP CÁC CHU TRÌNH CƠ BẢN CỦA ĐỒ THỊ

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

Nếu thêm một cạnh ngoài $e \in E \setminus F$ vào cây khung T, thì ta được đúng một chu trình đơn trong T, ký hiệu chu trình này là C_e. Tập các chu trình:

$$\Omega = \{C_e \mid e \in E \setminus F\}$$

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

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

1. Tập các chu trình cơ sở là phụ thuộc vào cây khung, **hai cây khung khác nhau có thể cho hai tập chu trình cơ sở khác nhau**.
2. Nếu đồ thị liên thông có n đỉnh và m cạnh, thì trong cây khung có n - 1 cạnh, còn lại m - n + 1 cạnh ngoài. Tương ứng với mỗi cạnh ngoài có một chu trình cơ sở, vậy **số chu trình cơ sở của đồ thị liên thông là m - n + 1**.
3. **Tập các chu trình cơ sở là tập nhiều nhất các chu trình** thoả mãn: Mỗi chu trình có đúng một cạnh riêng, cạnh đó không nằm trong bất cứ một chu trình nào khác. Bởi nếu có một tập gồm t chu trình thoả mãn điều đó thì việc loại bỏ cạnh riêng của một chu trình sẽ không làm mất tính liên thông của đồ thị, đồng thời không ảnh hưởng tới sự tồn tại của các chu trình khác. Như vậy nếu loại bỏ tất cả các cạnh riêng thì đồ thị vẫn liên thông và còn m - t cạnh. Đồ thị liên thông thì không thể có ít hơn n - 1 cạnh nên ta có $m - t \geq n - 1$ hay $t \leq m - n + 1$.
4. **Mọi cạnh trong một chu trình đơn bất kỳ đều phải thuộc một chu trình cơ sở**. Bởi nếu có một cạnh (u, v) không thuộc một chu trình cơ sở nào, thì khi ta bỏ cạnh đó đi đồ thị vẫn liên thông và không ảnh hưởng tới sự tồn tại của các chu trình cơ sở. Lại bỏ tiếp những cạnh ngoài

của các chu trình cơ sở thì đồ thị vẫn liên thông và còn lại m - (m - n + 1) - 1 = n - 2 cạnh. Điều này vô lý.

5. Đối với đồ thị G = (V, E) có n đỉnh và m cạnh, có k thành phần liên thông, ta có thể xét các thành phần liên thông và xét rừng các cây khung của các thành phần đó. Khi đó có thể mở rộng khái niệm tập các chu trình cơ sở cho đồ thị vô hướng tổng quát: Mỗi khi thêm một cạnh không nằm trong các cây khung vào rừng, ta được đúng một chu trình đơn, tập các chu trình đơn tạo thành bằng cách ghép các cạnh ngoài như vậy gọi là tập các chu trình cơ sở của đồ thị G. **Số các chu trình cơ sở là m - n + k**.

III. ĐỊNH CHIỀU ĐỒ THỊ VÀ BÀI TOÁN LIỆT KÊ CẦU

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

1. Phép định chiều DFS

Xét mô hình duyệt đồ thị bằng thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh 1. Vì đồ thị là vô hướng liên thông nên quá trình tìm kiếm sẽ thăm được hết các đỉnh.

```
procedure Visit(u ∈ V);
begin
  <Thông báo thăm u và đánh dấu u đã thăm>;
  for (∀v: (u, v) ∈ E) do
    if <v chưa thăm> then Visit(v);
end;

begin
  <Đánh dấu mọi đỉnh đều chưa thăm>;
  Visit(1);
end;
```

Coi một cạnh của đồ thị tương đương với hai cung có hướng ngược chiều nhau. Thuật toán tìm kiếm theo chiều sâu theo mô hình trên sẽ duyệt qua hết các đỉnh của đồ thị và tất cả các cung nữa. Quá trình duyệt cho ta một cây tìm kiếm DFS. Ta có các nhận xét sau:

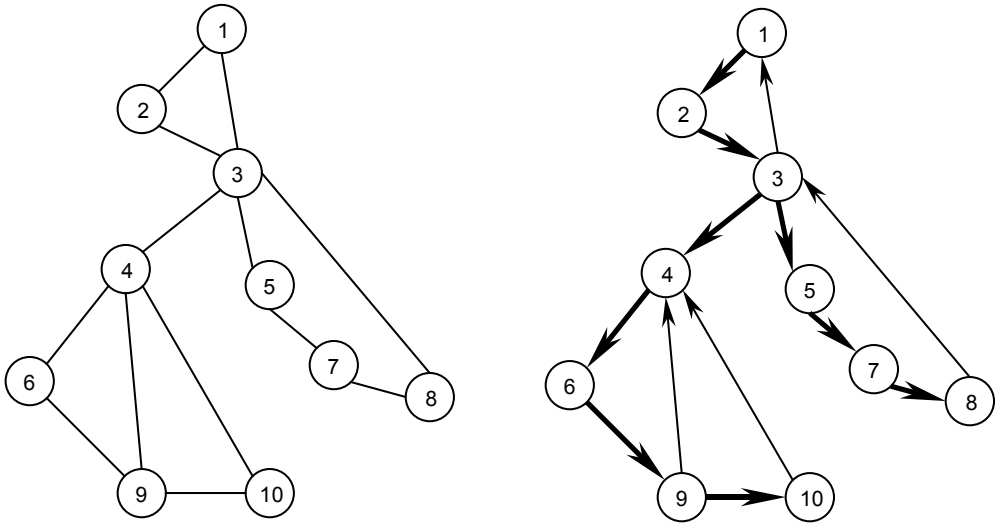
Nhận xét 1:

Quá trình duyệt sẽ không có cung chéo (cung đi từ một nhánh DFS thăm sau tới nhánh DFS thăm trước). Thật vậy, nếu quá trình duyệt xét tới một cung (u, v):

- Nếu u thăm trước v có nghĩa là khi Visit(u) được gọi thì v chưa thăm, vì thủ tục Visit(u) sẽ xây dựng nhánh DFS gốc u gồm những đỉnh chưa thăm đến được từ u, suy ra v nằm trong nhánh DFS gốc u $\Rightarrow$ v là hậu duệ của u, hay (u, v) là cung DFS hoặc cung xuôi.
- Nếu u thăm sau v (v thăm trước u), tương tự trên, ta suy ra u nằm trong nhánh DFS gốc v, v là tiền bối của u $\Rightarrow$ (u, v) là cung ngược.

Nhận xét 2:

Trong quá trình duyệt đồ thị theo chiều sâu, nếu cứ duyệt qua cung (u, v) nào thì ta bỏ đi cung (v, u). (Tức là hễ duyệt qua cung (u, v) thì ta định chiều luôn cạnh (u, v) theo chiều từ u tới v), ta được một phép định chiều đồ thị gọi là phép định chiều DFS.



Hình 15: Phép định chiều DFS

Nhận xét 3:

Với phép định chiều như trên, thì sẽ chỉ còn các cung trên cây DFS và cung ngược, không còn lại cung xuôi. Bởi trên đồ thị vô hướng ban đầu, nếu ta coi một cạnh là hai cung có hướng ngược chiều nhau thì với một cung xuôi ta có cung ngược chiều với nó là cung ngược. Do tính chất DFS, cung ngược được duyệt trước cung xuôi tương ứng, nên khi định chiều cạnh theo cung ngược thì cung xuôi sẽ bị huỷ và không bị xét tới nữa.

Nhận xét 4:

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

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

Chứng minh:

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

"⇒"

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

"⇐"

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

- Trước hết ta chứng minh rằng nếu (u, v) là cạnh của G thì sẽ có một đường đi từ u tới v trong G'. Thật vậy, vì (u, v) nằm trong một chu trình đơn, mà mọi cạnh của một chu trình đơn đều phải thuộc một chu trình cơ sở nào đó, nên sẽ có một chu trình cơ sở chứa cả u và v. Chu trình

cơ sở qua phép định chiều DFS vẫn là chu trình trong G' nên đi theo các cạnh định hướng của chu trình đó, ta có thể đi từ u tới v và ngược lại.

- Nếu u và v là 2 đỉnh bất kỳ của G thì do G liên thông, tồn tại một đường đi ($u=x_0, x_1, ..., x_n=v$). Vì (x_i, x_{i+1}) là cạnh của G nên trong G', từ x_i có thể đến được x_{i+1}. Suy ra từ u cũng có thể đến được v bằng các cạnh định hướng của G'.

2. Cài đặt

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

Ta sẽ cài đặt một thuật toán với một đồ thị vô hướng: liệt kê các cầu và định chiều các cạnh để được một đồ thị mới có ít thành phần liên thông mạnh nhất:

Đánh số các đỉnh theo thứ tự thăm DFS, gọi Numbering[u] là số thứ tự của đỉnh u theo cách đánh số đó. Trong quá trình tìm kiếm DFS, duyệt qua cạnh nào định chiều luôn cạnh đó. Định nghĩa thêm Low[u] là giá trị Numbering nhỏ nhất của những đỉnh đến được từ nhánh DFS gốc u bằng một cung ngược. Tức là nếu nhánh DFS gốc u có nhiều cung ngược hướng lên trên phía gốc cây thì ta ghi nhận lại cung ngược hướng lên cao nhất. Nếu nhánh DFS gốc u không chứa cung ngược thì ta cho Low[u] = +∞. Cụ thể cách cực tiểu hoá Low[u] như sau:

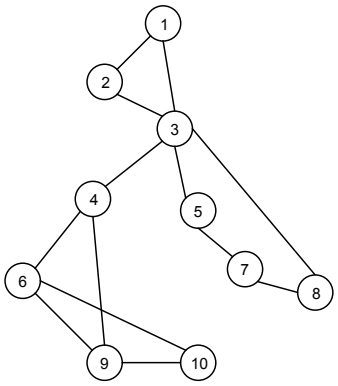
- Trong thủ tục Visit(u), trước hết ta đánh số thứ tự thăm cho đỉnh u (Numbering[u]) và khởi gán Low[u] = +∞.
- Sau đó, xét tất cả những đỉnh v kề u, định chiều cạnh (u, v) thành cung (u, v). Có hai khả năng xảy ra:
  - ♦ v chưa thăm thì ta gọi Visit(v) để thăm v và cực tiểu hoá Low[u] theo công thức:

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

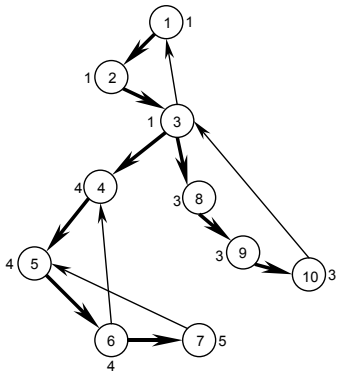
  - ♦ v đã thăm thì ta cực tiểu hoá Low[u] theo công thức:

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

  Dễ thấy cách tính như vậy là đúng đắn bởi nếu v chưa thăm thì nhánh DFS gốc v nằm trong nhánh DFS gốc u và những cung ngược trong nhánh DFS gốc v cũng là cung ngược trong nhánh DFS gốc u. Còn nếu v đã thăm thì (u, v) sẽ là cung ngược.



Đồ thị vô hướng



Đồ thị định chiều, Giá trị Numbering[u] ghi trong vòng tròn, Giá trị Low[u] ghi bên cạnh

Hình 16: Phép đánh số và ghi nhận cung ngược lên cao nhất

Nếu từ đỉnh u tới thăm đỉnh v, (u, v) là cung DFS. Khi đỉnh v được **duyệt xong**, lùi về thủ tục Visit(u), ta so sánh Low[v] và Numbering[u]. Nếu Low[v] > Numbering[u] thì tức là nhánh DFS gốc v không có cung ngược thoát lên phía trên v. Tức là cạnh (u, v) không thuộc một chu trình cơ sở nào cả, tức cạnh đó là cầu.

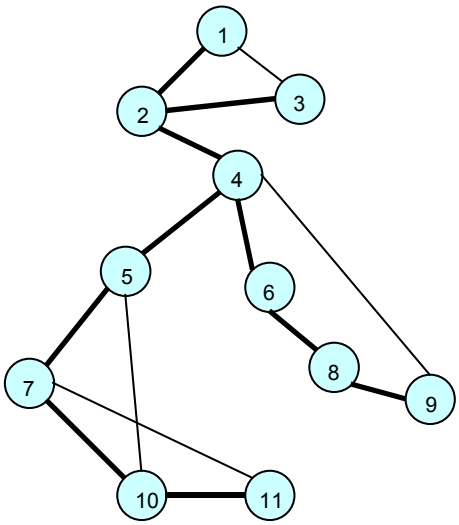
```
{Đồ thị G = (V, E)}
procedure Visit(u∈V);
begin
  <Đánh số thứ tự thăm cho đỉnh u (Numbering[u]); Khởi gán Low[u] := +∞>;
  for (∀v: (u, v)∈E) do
    begin
      <Định chiều cạnh (u, v) thành cung (u, v) ⇔ Loại bỏ cung (v, u)>;
      if <v chưa thăm> then
        begin
          Visit(v);
          if Low[v] > Numbering[u] then <In ra cầu (u, v)>;
          Low[u] := Min(Low[u], Low[v]);          {Cực tiểu hoá Low[u] theo Low[v]}
        end
      else {v đã thăm}
        Low[u] := Min(Low[u], Numbering[v]);      {Cực tiểu hoá Low[u] theo Numbering[v]}
    end;
end;

begin
  for (∀u∈V) do
    if <u chưa thăm> then Visit(u);
  <In ra cách định chiều>;
end.
```

Input: file văn bản GRAPH.INP

- Dòng 1 ghi số đỉnh n (n ≤ 100) và số cạnh m của đồ thị cách nhau ít nhất một dấu cách
- m dòng tiếp theo, mỗi dòng ghi hai số nguyên dương u, v cách nhau ít nhất một dấu cách, cho biết đồ thị có cạnh nối đỉnh u với đỉnh v

Output: file văn bản GRAPH.OUT

Thông báo các cầu và phép định chiều có ít thành phần liên thông mạnh nhất



GRAPH.INP	GRAPH.OUT
11 14 1 2 1 3 2 3 2 4 4 5 4 6 4 9 5 7 5 10 6 8 7 10 7 11 8 9 10 11	Bridges: (4, 5) (2, 4) Directed Edges: 1 -> 2 2 -> 3 2 -> 4 3 -> 1 4 -> 5 4 -> 6 5 -> 7 6 -> 8 7 -> 10 8 -> 9 9 -> 4 10 -> 5 10 -> 11 11 -> 7

PROG05_1.PAS * Phép định chiều DFS và liệt kê cầu

```
program Directivity_and_Bridges;
```

```
const
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;      {Ma trận kề của đồ thị}
  Numbering, Low: array[1..max] of Integer;
  n, Count: Integer;

procedure Enter;
var
  f: Text;
  i, m, u, v: Integer;
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;
      a[v, u] := True;
    end;
  Close(f);
end;

procedure Init;
begin
  FillChar(Numbering, SizeOf(Numbering), 0); {Numbering[u] = 0 ⇔ u chưa thăm}
  Count := 0;
end;

procedure Visit(u: Integer);
var
  v: Integer;
begin
  Inc(Count);
  Numbering[u] := Count;   {Đánh số thứ tự thăm cho đỉnh u, u trở thành đã thăm}
  Low[u] := n + 1;         {Khởi gán Low[u] bằng một giá trị đủ lớn hơn tất cả Numbering}
  for v := 1 to n do
    if a[u, v] then {Xét mọi đỉnh v kề u}
      begin
        a[v, u] := False;          {Định chiều cạnh (u, v) thành cung (u, v)}
        if Numbering[v] = 0 then     {Nếu v chưa thăm}
          begin
            Visit(v);                {Đi thăm v}
            if Low[v] > Numbering[u] then {(u, v) là cầu}
              WriteLn('(', u, ', ', v, ')');
            if Low[u] > Low[v] then Low[u] := Low[v];              {Cực tiểu hoá Low[u] }
          end
        else
          if Low[u] > Numbering[v] then Low[u] := Numbering[v]; {Cực tiểu hoá Low[u] }
      end;
end;

procedure Solve;
var
  u, v: Integer;
begin
  WriteLn('Bridges: ');            {Dùng DFS để định chiều đồ thị và liệt kê cầu}
  for u := 1 to n do
    if Numbering[u] = 0 then Visit(u);
  WriteLn('Directed Edges: ');     {Quét lại ma trận kề để in ra các cạnh định hướng}
  for u := 1 to n do
    for v := 1 to n do
      if a[u, v] then WriteLn(u, ' -> ', v);
```

```
end;

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

IV. LIỆT KÊ KHỚP

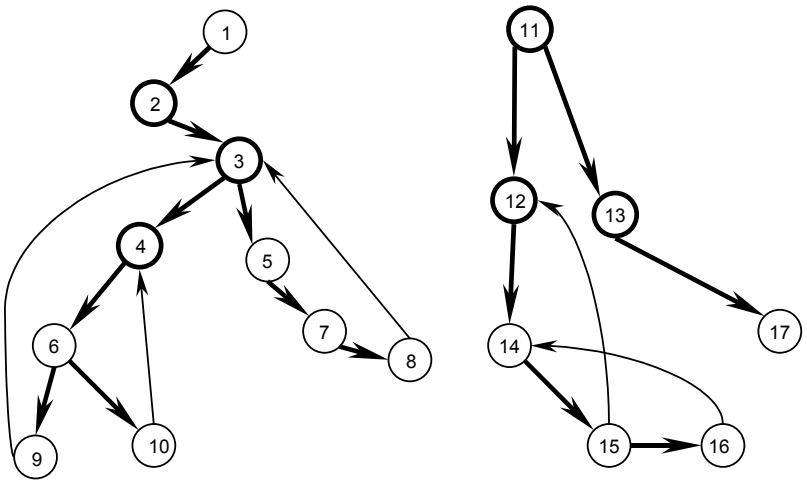
Trong đồ thị vô hướng, Một đỉnh C được gọi là khớp, nếu như ta bỏ đi đỉnh C và các cạnh liên thuộc với nó thì sẽ làm tăng số thành phần liên thông của đồ thị. Bài toán đặt ra là phải liệt kê hết các khớp của đồ thị.

Rõ ràng theo cách định nghĩa trên, các đỉnh treo và đỉnh cô lập sẽ không phải là khớp. Đồ thị liên thông có ≥ 3 đỉnh, không có khớp (cho dù bỏ đi đỉnh nào đồ thị vẫn liên thông) được gọi là đồ thị song liên thông. Giữa hai đỉnh phân biệt của đồ thị song liên thông, tồn tại ít nhất 2 đường đi không có đỉnh trung gian nào chung.

Coi mỗi cạnh của đồ thị ban đầu là hai cung có hướng ngược chiều nhau và dùng phép duyệt đồ thị theo chiều sâu:

```
{Đồ thị G = (V, E)}
procedure Visit(u ∈ V): ∈ V;
begin
  <Thông báo thăm u và đánh dấu u đã thăm>;
  for (∀v: (u, v) ∈ E) do
    if <v chưa thăm> then Visit(v);
end;

begin
  <Đánh dấu mọi đỉnh đều chưa thăm>;
  for (∀u∈V) do
    if <u chưa thăm> then Visit(u);
end;
```

Quá trình duyệt cho một rừng các cây DFS. Các cung duyệt qua có ba loại: cung DFS, cung ngược và cung xuôi, để không bị rối hình, ta chỉ ưu tiên vẽ cung DFS hoặc cung ngược:



Hình 17: Duyệt DFS, xác định cây DFS và các cung ngược

Hãy để ý nhánh DFS gốc ở đỉnh r nào đó

- Nếu **mọi** nhánh con của nhánh DFS gốc r đều có một cung ngược lên tới một tiền bối của r thì r không là khớp. Bởi nếu trong đồ thị ban đầu, ta bỏ r đi thì từ mỗi đỉnh bất kỳ của nhánh con, ta

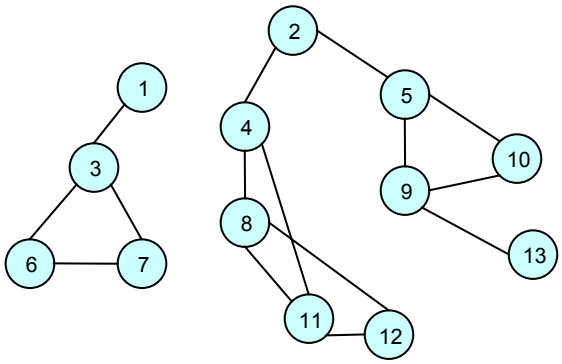
vẫn có thể đi lên một tiền bối của r, rồi đi sang nhánh con khác hoặc đi sang tất cả những đỉnh còn lại của cây. Số thành phần liên thông của đồ thị không thay đổi.

- Nếu r không phải là gốc của một cây DFS, và **tồn tại** một nhánh con của nhánh DFS gốc r không có cung ngược lên một tiền bối của r thì r là khớp. Bởi khi đó, tất cả những cung xuất phát từ nhánh con đó chỉ đi tới những đỉnh nội bộ trong nhánh DFS gốc r mà thôi, trên đồ thị ban đầu, không tồn tại cạnh nối từ những đỉnh thuộc nhánh con tới một tiền bối của r. Vậy từ nhánh đó muốn đi lên một tiền bối của r, tất phải đi qua r. Huỷ r khỏi đồ thị sẽ làm mất tất cả các đường đi đó, tức là làm tăng số thành phần liên thông của đồ thị.
- Nếu r là gốc của một cây DFS, thì r là khớp khi và chỉ khi r có ít nhất hai nhánh con. Bởi khi r có 2 nhánh con thì đường đi giữa hai đỉnh thuộc hai nhánh con đó tất phải đi qua r.

Vậy thì thuật toán liệt kê khớp lại là những kỹ thuật quen thuộc, duyệt DFS, đánh số, ghi nhận cạnh ngược lên cao nhất từ một nhánh con, chỉ thêm vào đó một thao tác nhỏ: Nếu từ đỉnh u gọi đệ quy thăm đỉnh v ((u, v) là cung DFS) thì sau khi duyệt xong đỉnh v, lùi về thủ tục Visit(u), ta so sánh Low[v] và Numbering[u] để kiểm tra xem từ nhánh con gốc v có cạnh ngược nào lên tiền bối của u hay không, nếu không có thì tạm thời đánh dấu u là khớp. Cuối cùng phải kiểm tra lại điều kiện: nếu u là gốc cây DFS thì nó là khớp khi và chỉ khi nó có ít nhất 2 nhánh con, nếu không thoả mãn điều kiện đó thì đánh dấu lại u không là khớp.

Input: file văn bản GRAPH.INP với khuôn dạng như bài toán liệt kê cầu

Output: Danh sách các khớp của đồ thị



GRAPH.INP	GRAPH.OUT
13 15 1 3 2 4 2 5 3 6 3 7 4 8 4 11 5 9 5 10 6 7 8 11 8 12 9 10 9 13 11 12	Cut vertices: 2, 3, 4, 5, 9,

PROG05_2.PAS * Liệt kê các khớp của đồ thị

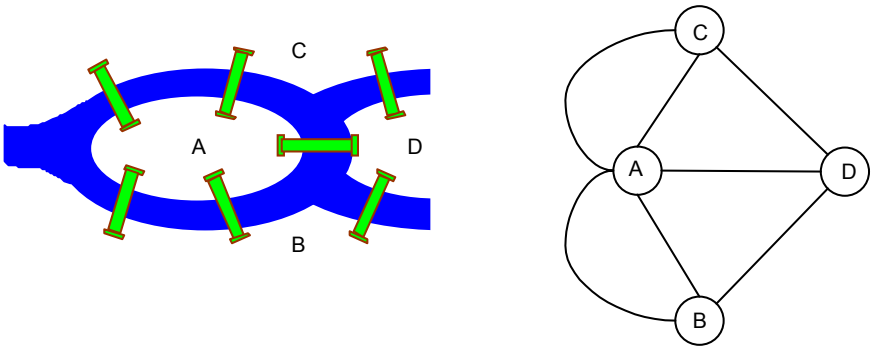
```
program CutVertices;
const
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;          {Ma trận kề của đồ thị}
  Numbering, Low, nC: array[1..max] of Integer; {nC[u]: Số nhánh con của nhánh DFS gốc u}
  Mark: array[1..max] of Boolean;               {Mark[u] = True ⇔ u là khớp}
  n, Count: Integer;

procedure LoadGraph;        {Nhập đồ thị (từ thiết bị nhập chuẩn Input)}
var
  i, m, u, v: Integer;
begin
  FillChar(a, SizeOf(a), False);
```

```
  ReadLn(n, m);
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True; a[v, u] := True;
    end;
end;

procedure Visit(u: Integer);              {Tìm kiếm theo chiều sâu bắt đầu từ u}
var
  v: Integer;
begin
  Inc(Count);
  Numbering[u] := Count; Low[u] := n + 1; nC[u] := 0;
  Mark[u] := False;
  for v := 1 to n do
    if a[u, v] then                       {Xét mọi v kề u}
      if Numbering[v] = 0 then            {Nếu v chưa thăm}
        begin
          Inc(nc[u]);                     {Tăng biến đếm số con của u lên 1}
          Visit(v);                       {Thăm v}
          {Nếu nhánh DFS gốc v không có cung ngược lên một tiền bối của u tức là Low[v] ≥ Numbering[u]}
          Mark[u] := Mark[u] or (Low[v] >= Numbering[u]);     {Tạm đánh dấu u là khớp}
          if Low[u] > Low[v] then Low[u] := Low[v];           {Cực tiểu hoá Low[u] }
        end
      else
        if Low[u] > Numbering[v] then Low[u] := Numbering[v]; {Cực tiểu hoá Low[u] }
end;

procedure Solve;
var
  u: Integer;
begin
  FillChar(Numbering, SizeOf(Numbering), 0);    {Đánh số = 0 ⇔ Đỉnh chưa thăm}
  FillChar(Mark, SizeOf(Mark), False);          {Mảng đánh dấu khớp chưa có gì}
  Count := 0;
  for u := 1 to n do
    if Numbering[u] = 0 then                    {Xét mọi đỉnh u chưa thăm}
      begin
        Visit(u);                               {Thăm u, xây dựng cây DFS gốc u}
        if nC[u] < 2 then                       {Nếu u có ít hơn 2 con}
            Mark[u] := False;                   {Thì u không phải là khớp}
      end;
end;

procedure Result;          {Dựa vào mảng đánh dấu để liệt kê các khớp}
var
  i: Integer;
begin
  WriteLn('Cut vertices:');
  for i := 1 to n do
    if Mark[i] then Write(i, ', ');
end;

begin
  Assign(Input, 'GRAPH.INP'); Reset(Input);
  Assign(Output, 'GRAPH.OUT'); Rewrite(Output);
  LoadGraph;
  Solve;
  Result;
  Close(Input);
  Close(Output);
end.
```

§6. 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 ?.**



Hình 18: Mô hình đồ thị của bài toán bảy cái cầu

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

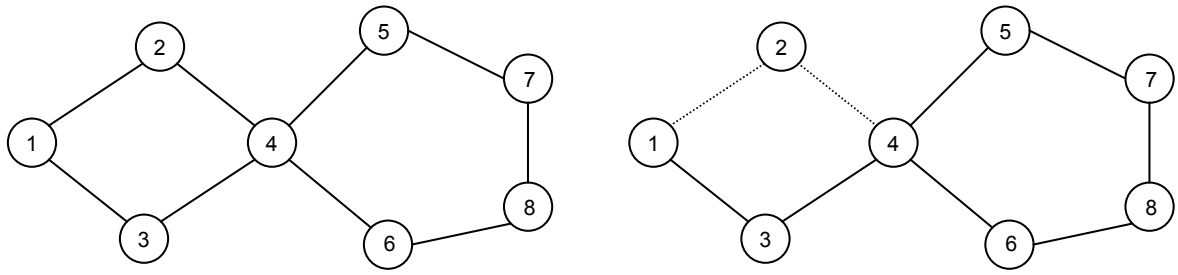
1. Đối với đồ thị vô hướng liên thông, mọi đỉnh đều có bậc chẵn.

Xuất phát từ một đỉnh, ta chọn một cạnh liên thuộc với nó để đi tiếp 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 trên 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.

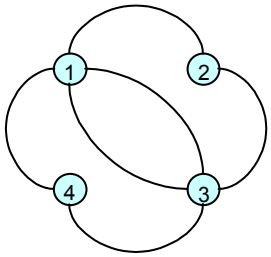
V. CÀI ĐẶT

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.

Input: file văn bản EULER.INP

- Dòng 1: Chứa số đỉnh n của đồ thị ($n \le 100$)
- Các dòng tiếp theo, mỗi dòng chứa 3 số nguyên dương cách nhau ít nhất 1 dấu cách có dạng: u v k cho biết giữa đỉnh u và đỉnh v có k cạnh nối

Output: file văn bản EULER.OUT ghi chu trình EULER



EULER.INP	EULER.OUT
4 1 2 1 1 3 2 1 4 1 2 3 1 3 4 1	1 2 3 1 3 4 1

PROG06_1.PAS * Thuật toán Fleury tìm chu trình Euler

```
program Euler_Circuit;
const
  max = 100;
var
  a: array[1..max, 1..max] of Integer;
  n: Integer;

procedure Enter;        {Nhập dữ liệu từ thiết bị nhập chuẩn Input}
var
  u, v, k: Integer;
begin
  FillChar(a, SizeOf(a), 0);
  ReadLn(n);
  while not SeekEof do
    begin
      ReadLn(u, v, k);
      a[u, v] := k;
      a[v, u] := k;
    end;
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: Integer): Boolean;
var
  Queue: array[1..max] of Integer; {Hàng đợi dùng cho Breadth First Search}
  First, Last: Integer; {First: Chỉ số đầu hàng đợi, Last: Chỉ số cuối hàng đợi}
  u, v: Integer;
  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, count: Integer;
```

```
begin
  Current := 1;
  Write(1:5); {Bắt đầu từ đỉnh Current = 1}
  count := 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:5);            {In kết quả đi tới Next}
        Inc(count);
        if count mod 16 = 0 then WriteLn; {In ra tối đa 16 đỉnh trên một dòng}
        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;

begin
  Assign(Input, 'EULER.INP'); Reset(Input);
  Assign(Output, 'EULER.OUT'); Rewrite(Output);
  Enter;
  FindEulerCircuit;
  Close(Input);
  Close(Output);
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 xoá 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 10^6 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.

PROG06_2.PAS * Thuật toán hiệu quả tìm chu trình Euler

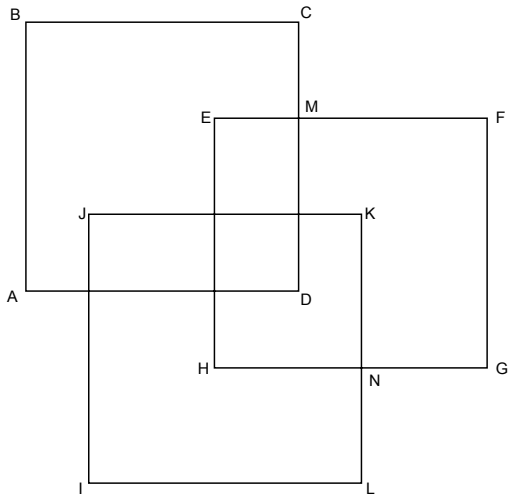
```
program Euler_Circuit;
const
  max = 100;
  maxE = 20000;     {Số cạnh tối đa}
var
  a: array[1..max, 1..max] of Integer;
  stack: array[1..maxE] of Integer;
  n, last: Integer;

procedure Enter;    {Nhập dữ liệu}
var
  u, v, k: Integer;
begin
  FillChar(a, SizeOf(a), 0);
  ReadLn(n);
  while not SeekEof do
    begin
      ReadLn(u, v, k);
      a[u, v] := k;
      a[v, u] := k;
    end;
end;

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

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

function Get: Integer;       {Trả về phần tử ở đỉnh (Top) ngăn xếp}
begin
  Get := Stack[last];
end;

procedure FindEulerCircuit;
var
```

```
  u, v, count: Integer;
begin
  Stack[1] := 1;     {Khởi tạo ngăn xếp ban đầu chỉ gồm đỉnh 1}
  last := 1;
  count := 0;
  while last <> 0 do  {Chừng nào ngăn xếp chưa rỗng}
    begin
      u := Get;        {Xác định u là phần tử ở đỉnh ngăn xếp}
      for v := 1 to n do
        if a[u, v] > 0 then  {Xét tất cả các cạnh liên thuộc với u, nếu thấy}
          begin
            Dec(a[u, v]); Dec(a[v, u]);      {Xoá cạnh đó khỏi đồ thị}
            Push(v);                         {Đẩy đỉnh tiếp theo vào ngăn xếp}
            Break;
          end;
      if u = Get then    {Nếu phần tử ở đỉnh ngăn xếp vẫn là u ⇒ vòng lặp trên không tìm thấy đỉnh nào kề với u}
        begin
          Inc(count);
          Write(Pop:5, ' ');    {In ra phần tử đỉnh ngăn xếp}
          if count mod 16 = 0 then WriteLn;   {Output không quá 16 số trên một dòng}
        end;
    end;
end;

begin
  Assign(Input, 'EULER.INP'); Reset(Input);
  Assign(Output, 'EULER.OUT'); Rewrite(Output);
  Enter;
  FindEulerCircuit;
  Close(Input);
  Close(Output);
end.
```

Bài tập:

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

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



M D A B C M F G N L I J K N H E M

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

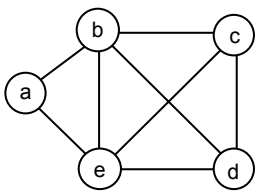
I. ĐỊNH NGHĨA

Cho đồ thị G = (V, E) có n đỉnh

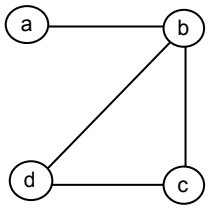
1. Chu trình $(x_1, x_2, ..., x_n, x_1)$ được gọi là chu trình Hamilton nếu $x_i \neq x_j$ với $1 \leq i < j \leq n$
2. Đường đi $(x_1, x_2, ..., x_n)$ được gọi là đường đi Hamilton nếu $x_i \neq x_j$ với $1 \leq i < j \leq n$

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.

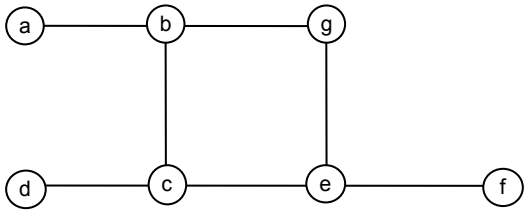
Ví dụ: Xét 3 đơn đồ thị G_1, G_2, G_3 sau:



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 ($n \geq 3$). Khi đó nếu mọi đỉnh v của G đều có $\deg(v) \geq n/2$ 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 $\deg^+(v) \geq n / 2$ và $\deg^-(v) \geq n / 2$ với mọi đỉnh v thì G có chu trình Hamilton

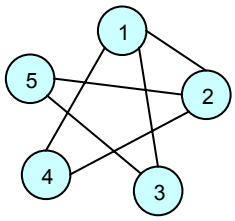
III. CÀI ĐẶT

Dưới đây ta sẽ cài đặt một chương trình liệt kê tất cả các chu trình Hamilton xuất phát từ đỉnh 1, các chu trình Hamilton khác có thể có được bằng cách hoán vị vòng quanh. 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.

Input: file văn bản HAMILTON.INP

- Dòng 1 ghi số đỉnh n (≤ 100) 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

Output: file văn bản HAMILTON.OUT liệt kê các chu trình Hamilton



HAMILTON.INP	HAMILTON.OUT
5 6 1 2 1 3 2 4 3 5 4 1 5 2	1 3 5 2 4 1 1 4 2 5 3 1

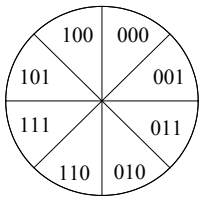
PROG07_1.PAS * Thuật toán quay lui liệt kê chu trình Hamilton

```
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 Integer;           {Chu trình Hamilton sẽ tìm là; 1=X[1]→X[2] → ... →X[n] →X[1]=1}
  n: Integer;

procedure Enter;       {Nhập dữ liệu từ thiết bị nhập chuẩn Input}
var
  i, u, v, m: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m);
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
end;

procedure PrintResult; {In kết quả nếu tìm thấy chu trình Hamilton}
var
  i: Integer;
begin
  for i := 1 to n do Write(X[i], ' ');
  WriteLn(X[1]);
end;

procedure Try(i: Integer); {Thử các cách chọn đỉnh thứ i trong hành trình}
var
  j: Integer;
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
  {Định hướng thiết bị nhập/xuất chuẩn}
  Assign(Input, 'HAMILTON.INP'); Reset(Input);
  Assign(Output, 'HAMILTON.OUT'); Rewrite(Output);
  Enter;
  FillChar(Free, n, True);          {Khởi tạo: Các đỉnh đều chưa đi qua}
  x[1] := 1; Free[1] := False;      {Bắt đầu từ đỉnh 1}
  Try(2);                           {Thử các cách chọn đỉnh kế tiếp}
  Close(Input);
  Close(Output);
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

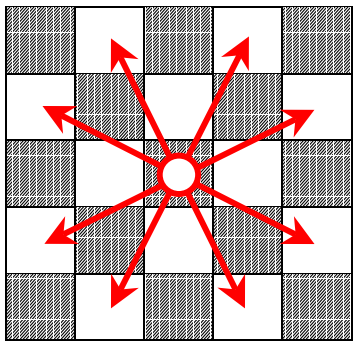


5. *__Thách đố:__ Bài toán mã đi tuần: Trên bàn cờ tổng quát kích thước n x n ô vuông (n chẵn và $6 \le n \le 20$). Trên một ô nào đó có đặt một quân mã. Quân mã đang ở ô (X_1, Y_1) có thể di chuyển sang ô (X_2, Y_2) nếu $|X_1-X_2|.|Y_1-Y_2| = 2$ (Xem hình vẽ).

   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).

§8. 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) $\in$ E thì C[u, v] = trọng số của cạnh (u, v)
- Nếu (u, v) $\notin$ 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 $\in$ V đến đỉnh đích F $\in$ 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 $\in$ V và c[u, v] = +∞ nếu như (u, v) $\notin$ 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:

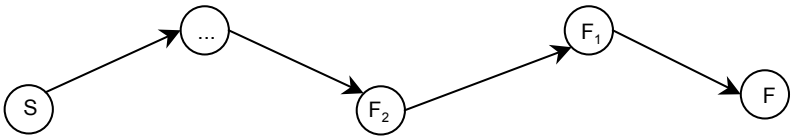
$$d[S, F] = d[S, F_1] + c[F_1, F]$$

(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.



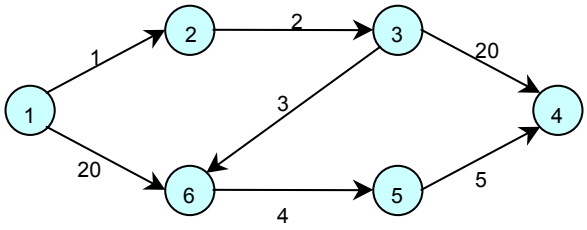
Tuy nhiên, trong đa số trường hợp, người ta không sử dụng phương pháp này mà sẽ kết hợp lưu vết đường đi ngay trong quá trình tìm kiếm.

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. Lưu ý rằng các thuật toán dưới đây sẽ luôn luôn tìm được đường đi ngắn nhất là đường đi cơ bản.

Input: file văn bản MINPATH.INP

- Dòng 1: Chứa số đỉnh n (≤ 100), số cung m của đồ thị, đỉnh xuất phát S, đỉnh đích F cách nhau ít nhất 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng ba số u, v, c[u, v] cách nhau ít nhất 1 dấu cách, thể hiện (u, v) là một cung ∈ E và trọng số của cung đó là c[u,v] (c[u, v] là số nguyên có giá trị tuyệt đối ≤ 100)

Output: file văn bản MINPATH.OUT ghi đường đi ngắn nhất từ S tới F và độ dài đường đi đó



MINPATH.INP	MINPATH.OUT
6 7 1 4 1 2 1 1 6 20 2 3 2 3 4 20 3 6 3 5 4 5 6 5 4	Distance from 1 to 4: 15 4<-5<-6<-3<-2<-1

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[v] được khởi gán bằng c[S, v]

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.

Tính đúng của thuật toán:

- Tại bước lặp 1: Bước khởi tạo d[v] = c[S, v]: thì dãy d[v] chính là độ dài ngắn nhất của đường đi từ S tới v qua không quá 1 cạnh.
- Giả sử tại bước lặp thứ i (i ≥ 1), d[v] bằng độ dài đường đi ngắn nhất từ S tới v qua không quá i cạnh, thì do tính chất: đường đi từ S tới v qua không quá i + 1 cạnh sẽ phải thành lập bằng cách:

lấy một đường đi từ S tới một đỉnh u nào đó qua không quá i cạnh, rồi đi tiếp tới v bằng cung (u, v). Nên độ dài đường đi ngắn nhất từ S tới v qua không quá i + 1 cạnh sẽ được tính bằng giá trị nhỏ nhất trong các giá trị: (Nguyên lý tối ưu Bellman)

- Độ dài đường đi ngắn nhất từ S tới v qua không quá i cạnh
- Độ dài đường đi ngắn nhất từ S tới u qua không quá i cạnh cộng với trọng số cạnh (u, v) ($\forall u$)

Vì vậy, sau bước lặp tối ưu các d[v] bằng công thức

$$d[v]_{\text{bước } i+1} = \min(d[v]_{\text{bước } i}, d[u]_{\text{bước } i} + c[u, v]) \ (\forall u)$$

thì các d[v] sẽ bằng độ dài đường đi ngắn nhất từ S tới v qua không quá i + 1 cạnh.

Sau bước lặp tối ưu thứ n - 2, ta có d[v] = độ dài đường đi ngắn nhất từ S tới v qua không quá n - 1 cạnh. Vì đồ thị không có chu trình âm nên sẽ có một đường đi ngắn nhất từ S tới v là đường đi cơ bản (qua không quá n - 1 cạnh). Tức là d[v] sẽ là độ dài đường đi ngắn nhất từ S tới v.

Vậy thì số bước lặp tối ưu hoá sẽ không quá n - 2 bước.

Trong khi cài đặt chương trình, nếu mỗi bước ta mô tả dưới dạng:

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

Thì do sự tối ưu bắc cầu (dùng d[u] tối ưu d[v] rồi lại có thể dùng d[v] tối ưu d[w] nữa...) nên chỉ làm tốc độ tối ưu nhãn d[v] tăng nhanh lên chứ không thể giảm đi được.

PROG08_1.PAS * Thuật toán Ford-Bellman

```
program Shortest_Path_by_Ford_Bellman;
const
  max = 100;
  maxC = 10000;
var
  c: array[1..max, 1..max] of Integer;
  d: array[1..max] of Integer;
  Trace: array[1..max] of Integer;
  n, S, F: Integer;

procedure LoadGraph;      {Nhập đồ thị từ thiết bị nhập chuẩn (Input), đồ thị không được có chu trình âm}
var
  i, m: Integer;
  u, v: Integer;
begin
  ReadLn(n, m, S, F);
  {Những cạnh không có trong đồ thị được gán trọng số +∞}
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(u, v, c[u, v]);
end;

procedure Init;           {Khởi tạo}
var
  i: Integer;
begin
  for i := 1 to n do
    begin
      d[i] := c[S, i];      {Độ dài đường đi ngắn nhất từ S tới i = c(S, i)}
      Trace[i] := S;
    end;
end;

procedure Ford_Bellman;           {Thuật toán Ford-Bellman}
```

```
var
  Stop: Boolean;
  u, v, CountLoop: Integer;
begin
  CountLoop := 0;                {Biến đếm số lần lặp}
  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 ∃u, v thoả mãn d[v] > d[u] + c[u, v] thì tối ưu lại d[v]}
          begin
            d[v] := d[u] + c[u, v];
            Trace[v] := u;               {Lưu vết đường đi}
            Stop := False;
          end;
    Inc(CountLoop);
  until Stop or (CountLoop >= n - 2);
  {Thuật toán kết thúc khi không sửa nhãn các d[v] được nữa hoặc đã lặp n-2 lần }
end;

procedure PrintResult;  {In đường đi từ S tới F}
begin
  if d[F] = maxC then   {Nếu d[F] vẫn là +∞ thì tức là không có đường}
    WriteLn('Path from ', S, ' to ', F, ' not found')
  else                  {Truy vết tìm đường đi}
    begin
      WriteLn('Distance from ', S, ' to ', F, ': ', d[F]);
      while F <> S do
        begin
          Write(F, '<-');
          F := Trace[F];
        end;
      WriteLn(S);
    end;
end;

begin
  Assign(Input, 'MINPATH.INP'); Reset(Input);
  Assign(Output, 'MINPATH.OUT'); Rewrite(Output);
  LoadGraph;
  Init;
  Ford_Bellman;
  PrintResult;
  Close(Input);
  Close(Output);
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[v] = c[S, v]. 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 ∈ 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[v] được khởi gán bằng c[S, v]. 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 kỹ thuật đánh dấu: Free[v] = TRUE hay FALSE tuỳ theo d[v] tự do hay cố định. Ban đầu các nhãn đều tự do.

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 +∞ (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] = +∞).

PROG08_2.PAS * Thuật toán Dijkstra

```
program Shortest_Path_by_Dijkstra;
const
  max = 100;
  maxC = 10000;
var
  c: array[1..max, 1..max] of Integer;
  d: array[1..max] of Integer;
  Trace: array[1..max] of Integer;
  Free: array[1..max] of Boolean;
  n, S, F: Integer;

procedure LoadGraph;        {Nhập đồ thị, trọng số các cung phải là số không âm}
var
  i, m: Integer;
  u, v: Integer;
begin
  ReadLn(n, m, S, F);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(u, v, c[u, v]);
end;

procedure Init;             {Khởi tạo các nhãn d[v], các đỉnh đều được coi là tự do}
var
  i: Integer;
begin
```

```
  for i := 1 to n do
    begin
      d[i] := c[S, i];
      Trace[i] := S;
    end;
  FillChar(Free, SizeOf(Free), True);
end;

procedure Dijkstra;       {Thuật toán Dijkstra}
var
  i, u, v: Integer;
  min: Integer;
begin
  repeat
    {Tìm trong các đỉnh có nhãn tự do ra đỉnh u có d[u] nhỏ nhất}
    u := 0; min := maxC;
    for i := 1 to n do
      if Free[i] and (d[i] < min) then
        begin
          min := d[i];
          u := i;
        end;
    {Thuật toán sẽ kết thúc khi các đỉnh  tự do đều có nhãn +∞ hoặc đã chọn đến đỉnh F}
    if (u = 0) or (u = F) then Break;
    {Cố định nhãn đỉnh u}
    Free[u] := False;
    {Dùng đỉnh u tối ưu nhãn những đỉnh tự do kề với u}
    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;      {In đường đi từ S tới F}
begin
  if d[F] = maxC then
    WriteLn('Path from ', S, ' to ', F, ' not found')
  else
    begin
      WriteLn('Distance from ', S, ' to ', F, ': ', d[F]);
      while F <> S do
        begin
          Write(F, '<-');
          F := Trace[F];
        end;
      WriteLn(S);
    end;
end;

begin
  Assign(Input, 'MINPATH.INP'); Reset(Input);
  Assign(Output, 'MINPATH.OUT'); Rewrite(Output);
  LoadGraph;
  Init;
  Dijkstra;
  PrintResult;
  Close(Input);
  Close(Output);
end.
```

V. THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC HEAP

Nếu đồ thị có nhiều đỉnh, ít cạnh, ta có thể sử dụng danh sách kề kèm trọng số để biểu diễn đồ thị, tuy nhiên tốc độ của thuật toán DIJKSTRA vẫn khá chậm vì trong trường hợp xấu nhất, nó cần n lần cố định nhãn và mỗi lần tìm đỉnh để cố định nhãn sẽ mất một đoạn chương trình với độ phức tạp O(n). Để tăng tốc độ, người ta thường sử dụng cấu trúc dữ liệu Heap để lưu các đỉnh chưa cố định nhãn. Heap ở đây là một cây nhị phân hoàn chỉnh thoả mãn: Nếu u là đỉnh lưu ở nút cha và v là đỉnh lưu ở nút con thì d[u] ≤ d[v]. (Đỉnh r lưu ở gốc Heap là đỉnh có d[r] nhỏ nhất).

Tại mỗi bước lặp của thuật toán Dijkstra có hai thao tác: Tìm đỉnh cố định nhãn và Sửa nhãn.

- Thao tác tìm đỉnh cố định nhãn sẽ lấy đỉnh lưu ở gốc Heap, cố định nhãn, đưa phần tử cuối Heap vào thế chỗ và thực hiện việc vun đống (Adjust)
- Thao tác sửa nhãn, sẽ duyệt danh sách kề của đỉnh vừa cố định nhãn và sửa nhãn những đỉnh tự do kề với đỉnh này, mỗi lần sửa nhãn một đỉnh nào đó, ta xác định đỉnh này nằm ở đâu trong Heap và thực hiện việc chuyển đỉnh đó lên (UpHeap) phía gốc Heap nếu cần để bảo toàn cấu trúc Heap.

Cài đặt dưới đây có Input/Output giống như trên nhưng có thể thực hiện trên đồ thị 5000 đỉnh, 10000 cạnh, trọng số mỗi cạnh ≤ 10000.

PROG08_3.PAS * Thuật toán Dijkstra và cấu trúc Heap

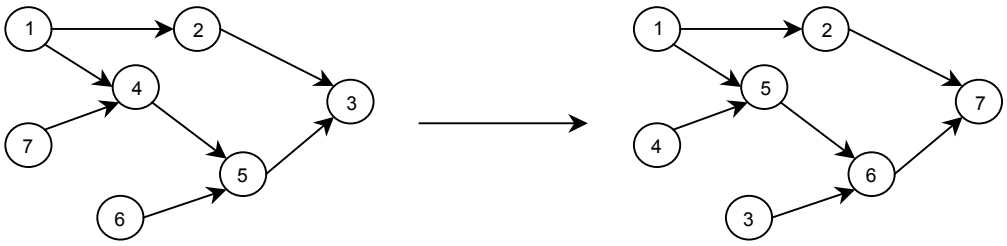
```
program Shortest_Path_by_Dijkstra_and_Heap;
const
  max = 5000;
  maxE = 10000;
  maxC = 1000000000;
type
  TAdj = array[1..maxE] of Integer;
  TAdjCost = array[1..maxE] of LongInt;
  THeader = array[1..max + 1] of Integer;
var
  adj: ^TAdj;                  {Danh sách kề dạng Forward Star}
  adjCost: ^TAdjCost;          {Kèm trọng số}
  head: ^THeader;              {Mảng đánh dấu các đoạn của Forward Star}
  d: array[1..max] of LongInt;
  Trace: array[1..max] of Integer;
  Free: array[1..max] of Boolean;
  heap, Pos: array[1..max] of Integer;
  n, S, F, nHeap: Integer;

procedure LoadGraph;  {Nhập dữ liệu}
var
  i, m: Integer;
  u, v, c: Integer;
  inp: Text;
begin
  {Đọc file lần 1, để xác định các đoạn}
  Assign(inp, 'MINPATH.INP'); Reset(inp);
  ReadLn(inp, n, m, S, F);
  New(head);
  New(adj); New(adjCost);
  {Phép đếm phân phối (Distribution Counting)}
  FillChar(head^, SizeOf(head^), 0);
  for i := 1 to m do
    begin
      ReadLn(inp, u);
      Inc(head^[u]);
    end;
  for i := 2 to n do head^[i] := head^[i - 1] + head^[i];
```

```
  Close(inp);
  {Đến đây, ta xác định được head[u] là vị trí cuối của danh sách kề đỉnh u trong adj^}
  Reset(inp);           {Đọc file lần 2, vào cấu trúc Forward Start}
  ReadLn(inp);          {Bỏ qua dòng đầu tiên Input file}
  for i := 1 to m do
    begin
      ReadLn(inp, u, v, c);
      adj^[head^[u]] := v;          {Điền v và c vào vị trí đúng trong danh sách kề của u}
      adjCost^[head^[u]] := c;
      Dec(head^[u]);
    end;
  head^[n + 1] := m;
  Close(inp);
end;

procedure Init;      {Khởi tạo d[i] = độ dài đường đi ngắn nhất từ S tới i qua 0 cạnh, Heap rỗng}
var
  i: Integer;
begin
  for i := 1 to n do d[i] := maxC;
  d[S] := 0;
  FillChar(Free, SizeOf(Free), True);
  FillChar(Pos, SizeOf(Pos), 0);
  nHeap := 0;
end;

procedure Update(v: Integer);      {Đỉnh v vừa được sửa nhãn, cần phải chỉnh lại Heap}
var
  parent, child: Integer;
begin
  child := Pos[v];     {child là vị trí của v trong Heap}
  if child = 0 then    {Nếu v chưa có trong Heap thì Heap phải bổ sung thêm 1 phần tử và coi child = nút lá cuối Heap}
    begin
      Inc(nHeap); child := nHeap;
    end;
  parent := child div 2;   {parent là nút cha của child}
  while (parent > 0) and (d[heap[parent]] > d[v]) do
    begin    {Nếu đỉnh lưu ở nút parent ưu tiên kém hơn v thì đỉnh đó sẽ bị đẩy xuống nút con child}
      heap[child] := heap[parent];  {Đẩy đỉnh lưu trong nút cha xuống nút con}
      Pos[heap[child]] := child;    {Ghi nhận lại vị trí mới của đỉnh đó}
      child := parent;              {Tiếp tục xét lên phía nút gốc}
      parent := child div 2;
    end;
  {Thao tác "kéo xuống" ở trên tạo ra một "khoảng trống" tại nút child của Heap, đỉnh v sẽ được đặt vào đây}
  heap[child] := v;
  Pos[v] := child;
end;

function Pop: Integer;
var
  r, c, v: Integer;
begin
  Pop := heap[1];      {Nút gốc Heap chứa đỉnh có nhãn tự do nhỏ nhất}
  v := heap[nHeap];    {v là đỉnh ở nút lá cuối Heap, sẽ được đảo lên đầu và vun đống}
  Dec(nHeap);
  r := 1;    {Bắt đầu từ nút gốc}
  while r * 2 <= nHeap do {Chừng nào r chưa phải là lá}
    begin
      {Chọn c là nút chứa đỉnh ưu tiên hơn trong hai nút con}
      c := r * 2;
      if (c < nHeap) and (d[heap[c + 1]] < d[heap[c]]) then Inc(c);
      {Nếu v ưu tiên hơn cả đỉnh chứa trong C, thì thoát ngay}
      if d[v] <= d[heap[c]] then Break;
      heap[r] := heap[c]; {Chuyển đỉnh lưu ở nút con c lên nút cha r}
```

```
      Pos[heap[r]] := r;   {Ghi nhận lại vị trí mới trong Heap của đỉnh đó}
      r := c;              {Gán nút cha := nút con và lặp lại}
    end;
  heap[r] := v;  {Đỉnh v sẽ được đặt vào nút r để bảo toàn cấu trúc Heap}
  Pos[v] := r;
end;

procedure Dijkstra;
var
  i, u, iv, v: Integer;
  min: Integer;
begin
  Update(1);
  repeat
    u := Pop;              {Chọn đỉnh tự do có nhãn nhỏ nhất}
    if u = F then Break;   {Nếu đỉnh đó là F thì dừng ngay}
    Free[u] := False;      {Cố định nhãn đỉnh đó}
    for iv := head^[u] + 1 to head^[u + 1] do   {Xét danh sách kề}
      begin
        v := adj^[iv];
        if Free[v] and (d[v] > d[u] + adjCost^[iv]) then
          begin
            d[v] := d[u] + adjCost^[iv];    {Tối ưu hoá nhãn của các đỉnh tự do kề với u}
            Trace[v] := u;                  {Lưu vết đường đi}
            Update(v);                      {Tổ chức lại Heap}
          end;
      end;
  until nHeap = 0;         {Không còn đỉnh nào mang nhãn tự do}
end;

procedure PrintResult;
var
  out: Text;
begin
  Assign(out, 'MINPATH.OUT'); Rewrite(out);
  if d[F] = maxC then
    WriteLn(out, 'Path from ', S, ' to ', F, ' not found')
  else
    begin
      WriteLn(out, 'Distance from ', S, ' to ', F, ': ', d[F]);
      while F <> S do
        begin
          Write(out, F, '<-');
          F := Trace[F];
        end;
      WriteLn(out, S);
    end;
  Close(out);
end;

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

VI. TRƯỜNG HỢP ĐỒ THỊ KHÔNG CÓ CHU TRÌNH - THỨ TỰ TÔ PÔ

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.



Hình 19: Phép đánh lại chỉ số theo thứ tự tôpô

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. (Thuật toán đánh số này có thể cải tiến bằng cách dùng một hàng đợi và cho những đỉnh không có cung đi vào đứng chờ lần lượt trong hàng đợi đó, lần lượt rút các đỉnh khỏi hàng đợi và đánh số cho nó, đồng thời huỷ những cung đi ra khỏi đỉnh vừa đánh số, lưu ý sau mỗi lần loại bỏ cung (u, v), nếu thấy bán bậc vào của v = 0 thì đẩy v vào chờ trong hàng đợi, như vậy đỡ mất công duyệt để tìm những đỉnh có bán bậc vào = 0)

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[v] = c[S, v]. 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.**

PROG08_4.PAS * Đường đi ngắn nhất trên đồ thị không có chu trình

```
program Critical_Path;
const
  max = 100;
  maxC = 10000;
var
  c: array[1..max, 1..max] of Integer;
  List, d, Trace: array[1..max] of Integer;   {List là danh sách các đỉnh theo cách đánh số mới}
  n, S, F, count: Integer;

procedure LoadGraph;  {Nhập dữ liệu, đồ thị không được có chu trình}
var
  i, m: Integer;
  u, v: Integer;
begin
  ReadLn(n, m, S, F);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(u, v, c[u, v]);
end;
```

```
procedure Number;       {Thuật toán đánh số các đỉnh}
var
  deg: array[1..max] of Integer;
  u, v: Integer;
  front: Integer;
begin
  {Trước hết, tính bán bậc vào của các đỉnh (deg-)}
  FillChar(deg, SizeOf(deg), 0);
  for u := 1 to n do
    for v := 1 to n do
      if (v <> u) and (c[v, u] < maxC) then Inc(deg[u]);
  {Đưa những đỉnh có bán bậc vào = 0 vào danh sách List}
  count := 0;
  for u := 1 to n do
    if deg[u] = 0 then
      begin
        Inc(count); List[count] := u;
      end;
  {front: Chỉ số phần tử đang xét, count: Số phần tử trong danh sách}
  front := 1;
  while front <= count do     {Chừng nào chưa xét hết các phần tử trong danh sách}
    begin
      {Xét phần tử thứ front trong danh sách, đẩy con trỏ front sang phần tử kế tiếp}
      u := List[front]; Inc(front);
      for v := 1 to n do
        if c[u, v] <> maxC then   {Xét những cung (u, v) và "loại" khỏi đồ thị ⇔ deg-(v) giảm 1}
          begin
            Dec(deg[v]);
            if deg[v] = 0 then    {Nếu v trở thành đỉnh không có cung đi vào}
              begin               {Đưa tiếp v vào danh sách List}
                Inc(count);
                List[count] := v;
              end;
          end;
    end;
end;

procedure Init;
var
  i: Integer;
begin
  for i := 1 to n do
    begin
      d[i] := c[S, i];
      Trace[i] := S;
    end;
end;

procedure FindPath;          {Thuật toán quy hoạch động tìm đường đi ngắn nhất trên đồ thị không chu trình}
var
  i, j, u, v: Integer;
begin
  for i := 1 to n - 1 do
    for j := i + 1 to n do
      begin
        u := List[i]; v := List[j];       {Dùng List[i] tối ưu nhãn List[j] với i < j}
        if d[v] > d[u] + c[u, v] then
          begin
            d[v] := d[u] + c[u, v];
            Trace[v] := u;
          end
      end;
end;
```

```
procedure PrintResult;      {In đường đi từ S tới F}
begin
  if d[F] = maxC then
    WriteLn('Path from ', S, ' to ', F, ' not found')
  else
    begin
      WriteLn('Distance from ', S, ' to ', F, ': ', d[F]);
      while F <> S do
        begin
          Write(F, '<-');
          F := Trace[F];
        end;
      WriteLn(S);
    end;
end;

begin
  Assign(Input, 'MINPATH.INP'); Reset(Input);
  Assign(Output, 'MINPATH.OUT'); Rewrite(Output);
  LoadGraph;
  Number;
  if Count < n then
    WriteLn('Error: Circuit Exist')
  else
    begin
      Init;
      FindPath;
      PrintResult;
    end;
  Close(Input);
  Close(Output);
end.
```

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

Cho đơn 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):

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

Tính đúng của thuật toán:

Gọi $c^k[u, v]$ là độ dài đường đi ngắn nhất từ u tới v mà chỉ đi qua các đỉnh trung gian thuộc tập {1, 2, ..., k}. Rõ ràng khi k = 0 thì $c^0[u, v] = c[u, v]$ (đường đi ngắn nhất là đường đi trực tiếp).

Giả sử ta đã tính được các $c^{k-1}[u, v]$ thì $c^k[u, v]$ sẽ được xây dựng như sau:

Nếu đường đi ngắn nhất từ u tới v mà chỉ qua các đỉnh trung gian thuộc tập {1, 2, ..., k} lại:

- Không đi qua đỉnh k thì tức là chỉ qua các đỉnh trung gian thuộc tập {1, 2, ..., k - 1} thì

$$c^k[u, v] = c^{k-1}[u, v]$$

- Có đi qua đỉnh k thì đường đi đó sẽ là nối của một đường đi từ u tới k và một đường đi từ k tới v, hai đường đi này chỉ đi qua các đỉnh trung gian thuộc tập {1, 2, ..., k - 1}.

$$c^k[u, v] = c^{k-1}[u, k] + c^{k-1}[k, v].$$

Vì ta muốn $c^k[u, v]$ là cực tiểu nên suy ra: $c^k[u, v] = \min(c^{k-1}[u, v], c^{k-1}[u, k] + c^{k-1}[k, v])$.

Và cuối cùng, ta quan tâm tới $c^n[u, v]$: Độ dài đường đi ngắn nhất từ u tới v mà chỉ đi qua các đỉnh trung gian thuộc tập {1, 2, ..., n}.

Khi cài đặt, thì ta sẽ không có các khái niệm $c^k[u, v]$ mà sẽ thao tác trực tiếp trên các trọng số c[u, v]. c[u, v] tại bước tối ưu thứ k sẽ được tính toán để tối ưu qua các giá trị c[u, v]; c[u, k] và c[k, v] tại bước thứ k - 1. Và nếu cài đặt dưới dạng ba vòng lặp for lồng như trên, do có sự tối ưu bắc cầu tại mỗi bước, tốc độ tối ưu c[u, v] chỉ tăng lên chứ không thể giảm đi được.

PROG08_5.PAS * Thuật toán Floyd

```
program Shortest_Path_by_Floyd;
const
  max = 100;
  maxC = 10000;
var
  c: array[1..max, 1..max] of Integer;
  Trace: array[1..max, 1..max] of Integer; {Trace[u, v] = Đỉnh liền sau u trên đường đi từ u tới v}
  n, S, F: Integer;

procedure LoadGraph;      {Nhập dữ liệu, đồ thị không được có chu trình âm}
var
  i, m: Integer;
  u, v: Integer;
begin
  ReadLn(n, m, S, F);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(u, v, c[u, v]);
end;

procedure Floyd;
var
  k, u, v: Integer;
begin
  for u := 1 to n do
    for v := 1 to n do Trace[u, v] := v; {Giả sử đường đi ngắn nhất giữa mọi cặp đỉnh là đường trực tiếp}
  {Thuật toán Floyd}
  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   {Đường đi từ qua k tốt hơn}
          begin
            c[u, v] := c[u, k] + c[k, v];     {Ghi nhận đường đi đó thay cho đường cũ}
            Trace[u, v] := Trace[u, k];       {Lưu vết đường đi}
          end;
end;

procedure PrintResult;  {In đường đi từ S tới F}
begin
  if c[S, F] = maxC
    then WriteLn('Path from ', S, ' to ', F, ' not found')
  else
    begin
      WriteLn('Distance from ', S, ' to ', F, ': ', c[S, F]);
      repeat
```

```
        Write(S, '->');
        S := Trace[S, F];      {Nhắc lại rằng Trace[S, F] là đỉnh liền sau S trên đường đi tới F}
      until S = F;
      WriteLn(F);
    end;
end;

begin
  Assign(Input, 'MINPATH.INP'); Reset(Input);
  Assign(Output, 'MINPATH.OUT'); Rewrite(Output);
  LoadGraph;
  Floyd;
  PrintResult;
  Close(Input);
  Close(Output);
end.
```

Khác biệt rõ ràng của thuật toán Floyd là khi cần tìm đường đi ngắn nhất giữa một cặp đỉnh khác, chương trình chỉ việc in kết quả chứ không phải thực hiện lại thuật toán Floyd nữa.

VIII. NHẬN XÉT

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

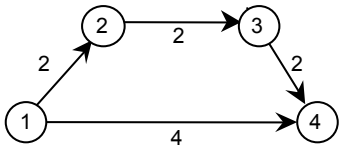
Trong tất cả các cài đặt trên, vì sử dụng ma trận trọng số chứ không sử dụng danh sách cạnh hay danh sách kề có trọng số, nên ta đều đưa về đồ thị đầy đủ và **đem trọng số +∞ gán cho những cạnh không có trong đồ thị ban đầu**. Trên máy tính thì không có khái niệm trừu tượng +∞ nên ta sẽ phải chọn một số dương đủ lớn để thay. Như thế nào là đủ lớn? **số đó phải đủ lớn hơn tất cả trọng số của các đường đi cơ bản** để cho dù đường đi thật có tồi tệ đến đâu vẫn tốt hơn đường đi trực tiếp theo cạnh tưởng tượng ra đó. Vậy nên nếu đồ thị cho số đỉnh cũng như trọng số các cạnh vào cỡ 300 chẳng hạn thì giá trị đó **không thể chọn trong phạm vi Integer hay Word.** Ma trận c sẽ phải khai báo là **ma trận LongInt** và giá trị hằng số maxC trong các chương trình trên phải đổi lại là 300 * 299 + 1 - điều đó có thể gây ra nhiều phiền toái, chẳng hạn như vấn đề lãng phí bộ nhớ. Để khắc phục, người ta có thể cài đặt bằng danh sách kề kèm trọng số hoặc sử dụng những kỹ thuật đánh dấu khéo léo trong từng trường hợp cụ thể. Tuy nhiên có một điều chắc chắn: khi đồ thị cho số đỉnh cũng như trọng số các cạnh vào khoảng 300 thì các **trọng số c[u, v] trong thuật toán Floyd** và các **nhãn d[v] trong ba thuật toán còn lại** chắc chắn **không thể khai báo là Integer** được.

Xét về độ phức tạp tính toán, nếu cài đặt như trên, thuật toán Ford-Bellman có độ phức tạp là $O(n^3)$, thuật toán Dijkstra là $O(n^2)$, thuật toán tối ưu nhãn theo thứ tự tôpô là $O(n^2)$ còn thuật toán Floyd là $O(n^3)$. Tuy nhiên nếu sử dụng danh sách kề, thuật toán tối ưu nhãn theo thứ tự tôpô sẽ có độ phức tạp tính toán là O(m). Thuật toán Dijkstra kết hợp với cấu trúc dữ liệu Heap có độ phức tạp O(max(n, m).logn).

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 ≤ 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. Chi phí di chuyển trên mỗi đường tròn bằng 0. Chi phí di chuyển giữa hai đường tròn bằng khoảng cách giữa chúng. Hãy tìm phương án di chuyển giữa hai đường tròn S, F cho trước với chi phí ít nhất.

3. Cho một dãy n số nguyên A[1], A[2], ..., A[n] (n ≤ 10000; 1 ≤ A[i] ≤ 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. 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.

5. Cho một bảng các số tự nhiên kích thước mxn (1 ≤ m, n ≤ 100). Từ một ô có thể di chuyển sang một ô kề cạnh với nó. Hãy tìm một cách đi từ ô (x, y) ra một ô biên sao cho tổng các số ghi trên các ô đi qua là cực tiểu.

§9. BÀI TOÁN CÂY KHUNG NHỎ NHẤT

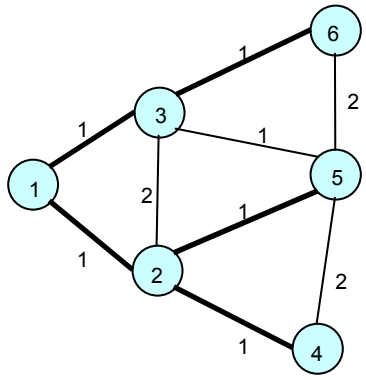
I. BÀI TOÁN CÂY KHUNG NHỎ NHẤT

Cho G = (V, E) là đồ thị vô hướng liên thông có trọng số, 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 của đơn đồ thị vô hướng có trọng số.

Input: file văn bản MINTREE.INP:

- Dòng 1: Ghi hai số số đỉnh n (≤ 100) và số cạnh m của đồ thị cách nhau ít nhất 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng 3 số u, v, c[u, v] cách nhau ít nhất 1 dấu cách thể hiện đồ thị có cạnh (u, v) và trọng số cạnh đó là c[u, v]. (c[u, v] là số nguyên có giá trị tuyệt đối không quá 100).

Output: file văn bản MINTREE.OUT ghi các cạnh thuộc cây khung và trọng số cây khung



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

II. THUẬT TOÁN KRUSKAL (JOSEPH KRUSKAL - 1956)

Thuật toán Kruskal dựa trên mô hình xây dựng cây khung bằng thuật toán hợp nhất (§5), chỉ có điều thuật toán không phải xét các cạnh với thứ tự tuỳ ý mà xét các cạnh theo thứ tự đã sắp xếp: 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 đơn. 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:

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 hiệu quả để đạt được tốc độ nhanh trong trường hợp số cạnh lớn. Trong trường hợp tổng quát, thuật toán HeapSort là hiệu quả nhất bởi nó cho phép chọn lần lượt các cạnh từ cạnh trọng nhỏ nhất tới cạnh trọng số lớn nhất ra khỏi Heap và có thể xử lý (bỏ qua hay thêm vào cây) luôn.

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. Ban đầu, ta khởi tạo rừng T gồm n cây, mỗi cây chỉ gồm đúng một đỉnh, sau đó, mỗi khi xét đến **cạnh nối hai cây khác nhau** của rừng T thì ta **kết nạp cạnh đó** vào T, đồng thời **hợp nhất hai cây đó lại thành một cây**.

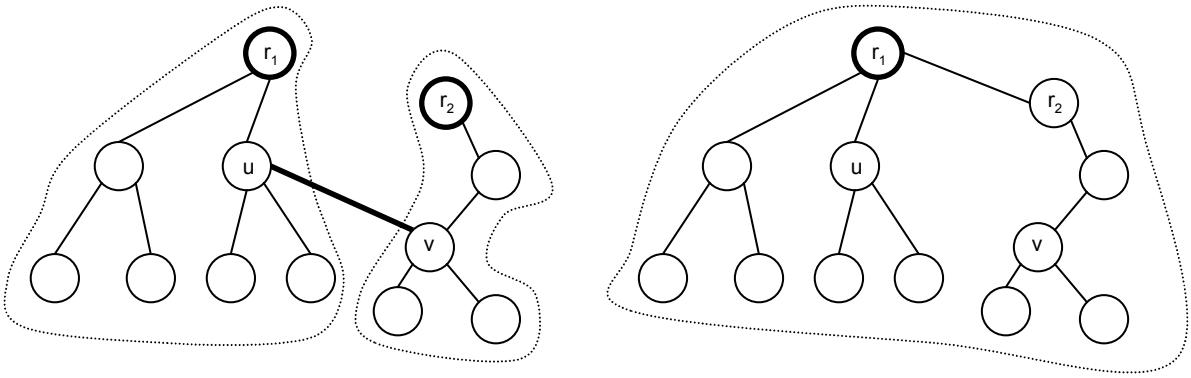
Nếu cho mỗi đỉnh v trên cây một nhãn Lab[v] là số hiệu đỉnh cha của đỉnh v trong cây, trong trường hợp v là gốc của một cây thì Lab[v] được gán một giá trị âm. Khi đó ta hoàn toàn có thể xác định được gốc của cây chứa đỉnh v bằng hàm GetRoot như sau:

```
function GetRoot(v∈V): ∈V;
begin
  while Lab[v] > 0 do v := Lab[v];
  GetRoot := v;
end;
```

Vậy để kiểm tra một cạnh (u, v) có nối hai cây khác nhau của rừng T hay không? ta có thể kiểm tra GetRoot(u) có khác GetRoot(v) hay không, bởi mỗi cây chỉ có duy nhất một gốc.

Để hợp nhất cây gốc r_1 và cây gốc r_2 thành một cây, ta lưu ý rằng mỗi cây ở đây chỉ dùng để ghi nhận một tập hợp đỉnh thuộc cây đó chứ cấu trúc cạnh trên cây thế nào thì không quan trọng. Vậy để hợp nhất cây gốc r_1 và cây gốc r_2, ta chỉ việc coi r_1 là nút cha của r_2 trong cây bằng cách đặt:

$$Lab[r_2] := r_1.$$



Hai cây gốc r_1 và r_2 và cây mới khi hợp nhất chúng

Tuy nhiên, để thuật toán làm việc hiệu quả, tránh trường hợp cây tạo thành bị suy biến khiến cho hàm GetRoot hoạt động chậm, người ta thường đánh giá: Để hợp hai cây lại thành một, thì gốc cây nào ít nút hơn sẽ bị coi là con của gốc cây kia.

Thuật toán hợp nhất cây gốc r_1 và cây gốc r_2 có thể viết như sau:

```
{Count[k] là số đỉnh của cây gốc k}
procedure Union(r1, r2 ∈ V);
begin
  if Count[r1] < Count[r2] then      {Hợp nhất thành cây gốc r2}
    begin
      Count[r2] := Count[r1] + Count[r2];
      Lab[r1] := r2;
    end
  else                               {Hợp nhất thành cây gốc r1}
    begin
      Count[r1] := Count[r1] + Count[r2];
      Lab[r2] := r1;
    end;
end;
```

Khi cài đặt, ta có thể tận dụng ngay nhãn Lab[r] để lưu số đỉnh của cây gốc r, bởi như đã giải thích ở trên, Lab[r] chỉ cần mang một giá trị âm là đủ, vậy ta có thể coi Lab[r] = -Count[r] với r là gốc của một cây nào đó.

```
procedure Union(r1, r2 ∈ V);        {Hợp nhất cây gốc r1 với cây gốc r2}
begin
  x := Lab[r1] + Lab[r2];           {-x là tổng số nút của cả hai cây}
  if Lab[r1] > Lab[r2] then         {Cây gốc r1 ít nút hơn cây gốc r2, hợp nhất thành cây gốc r2}
    begin
      Lab[r1] := r2;  {r2 là cha của r1}
      Lab[r2] := x;   {r2 là gốc cây mới, -Lab[r2] giờ đây là số nút trong cây mới}
    end
  else                              {Hợp nhất thành cây gốc r1}
    begin
      Lab[r1] := x;   {r1 là gốc cây mới, -Lab[r1] giờ đây là số nút trong cây mới}
      Lab[r2] := r1;  {cha của r2 sẽ là r1}
    end;
end;
```

Mô hình thuật toán Kruskal:

```
for ∀k∈V do Lab[k] := -1;
for ∀(u, v)∈E (theo thứ tự từ cạnh trọng số nhỏ tới cạnh trọng số lớn) do
  begin
    r1 := GetRoot(u); r2 := GetRoot(v);
    if r1 ≠ r2 then   {(u, v) nối hai cây khác nhau}
      begin
        <Kết nạp (u, v) vào cây, nếu đã đủ n - 1 cạnh thì thuật toán dừng>
        Union(r1, r2);  {Hợp nhất hai cây lại thành một cây}
      end;
  end;
```

PROG09_1.PAS * Thuật toán Kruskal

```
program Minimal_Spanning_Tree_by_Kruskal;
const
  maxV = 100;
  maxE = (maxV - 1) * maxV div 2;
type
  TEdge = record          {Cấu trúc một cạnh}
    u, v, c: Integer;     {Hai đỉnh và trọng số}
    Mark: Boolean;        {Đánh dấu có được kết nạp vào cây khung hay không}
  end;
var
  e: array[1..maxE] of TEdge;        {Danh sách cạnh}
  Lab: array[1..maxV] of Integer;    {Lab[v] là đỉnh cha của v, nếu v là gốc thì Lab[v] = - số con cây gốc v}
  n, m: Integer;
  Connected: Boolean;

procedure LoadGraph;      {Nhập đồ thị từ thiết bị nhập chuẩn (Input)}
var
  i: Integer;
begin
  ReadLn(n, m);
  for i := 1 to m do
    with e[i] do
      ReadLn(u, v, c);
end;

procedure Init;
var
  i: Integer;
begin
  for i := 1 to n do Lab[i] := -1;        {Rừng ban đầu, mọi đỉnh là gốc của cây gồm đúng một nút}
  for i := 1 to m do e[i].Mark := False;
end;
```

```
function GetRoot(v: Integer): Integer;   {Lấy gốc của cây chứa v}
begin
  while Lab[v] > 0 do v := Lab[v];
  GetRoot := v;
end;

procedure Union(r1, r2: Integer);        {Hợp nhất hai cây lại thành một cây}
var
  x: Integer;
begin
  x := Lab[r1] + Lab[r2];
  if Lab[r1] > Lab[r2] then
    begin
      Lab[r1] := r2;
      Lab[r2] := x;
    end
  else
    begin
      Lab[r1] := x;
      Lab[r2] := r1;
    end;
end;

procedure AdjustHeap(root, last: Integer);    {Vun thành đống, dùng cho HeapSort}
var
  Key: TEdge;
  child: Integer;
begin
  Key := e[root];
  while root * 2 <= Last do
    begin
      child := root * 2;
      if (child < Last) and (e[child + 1].c < e[child].c)
        then Inc(child);
      if Key.c <= e[child].c then Break;
      e[root] := e[child];
      root := child;
    end;
  e[root] := Key;
end;

procedure Kruskal;
var
  i, r1, r2, Count, a: Integer;
  tmp: TEdge;
begin
  Count := 0;
  Connected := False;
  for i := m div 2 downto 1 do AdjustHeap(i, m);
  for i := m - 1 downto 1 do
    begin
      tmp := e[1]; e[1] := e[i + 1]; e[i + 1] := tmp;
      AdjustHeap(1, i);
      r1 := GetRoot(e[i + 1].u); r2 := GetRoot(e[i + 1].v);
      if r1 <> r2 then                {Cạnh e[i + 1] nối hai cây khác nhau}
        begin
          e[i + 1].Mark := True;      {Kết nạp cạnh đó vào cây}
          Inc(Count);                 {Đếm số cạnh}
          if Count = n - 1 then       {Nếu đã đủ số thì thành công}
            begin
              Connected := True;
              Exit;
            end;
```

```
          Union(r1, r2);              {Hợp nhất hai cây thành một cây}
        end;
    end;
end;

procedure PrintResult;
var
  i, Count, W: Integer;
begin
  if not Connected then
    WriteLn('Error: Graph is not connected')
  else
    begin
      WriteLn('Minimal spanning tree: ');
      Count := 0;
      W := 0;
      for i := 1 to m do                  {Duyệt danh sách cạnh}
        with e[i] do
          begin
            if Mark then                  {Lọc ra những cạnh đã kết nạp vào cây khung}
              begin
                WriteLn('(', u, ', ', v, ') = ', c);
                Inc(Count);
                W := W + c;
              end;
            if Count = n - 1 then Break;  {Cho tới khi đủ n - 1 cạnh}
          end;
      WriteLn('Weight = ', W);
    end;
end;

begin
  Assign(Input, 'MINTREE.INP'); Reset(Input);
  Assign(Output, 'MINTREE.OUT'); Rewrite(Output);
  LoadGraph;
  Init;
  Kruskal;
  PrintResult;
  Close(Input);
  Close(Output);
end.
```

Xét về độ phức tạp tính toán, ta có thể chứng minh được rằng thao tác GetRoot có độ phức tạp là $O(\log_2 n)$, còn thao tác Union là O(1). Giả sử ta đã có danh sách cạnh đã sắp xếp rồi thì xét vòng lặp dựng cây khung, nó duyệt qua danh sách cạnh và với mỗi cạnh nó gọi 2 lần thao tác GetRoot, vậy thì độ phức tạp là $O(m\log_2 n)$, nếu đồ thị có cây khung thì $m \geq n-1$ nên ta thấy chi phí thời gian chủ yếu sẽ nằm ở thao tác sắp xếp danh sách cạnh bởi độ phức tạp của HeapSort là $O(m\log_2 m)$. Vậy độ phức tạp tính toán của thuật toán là $O(m\log_2 m)$ trong trường hợp xấu nhất. Tuy nhiên, phải lưu ý rằng để xây dựng cây khung thì ít khi thuật toán phải duyệt toàn bộ danh sách cạnh mà chỉ một phần của danh sách cạnh mà thôi.

III. 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 được cho bởi ma trận trọng số C. Qui ước $c[u, v] = +\infty$ 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).

PROG09_2.PAS * Thuật toán Prim

```
program Minimal_Spanning_Tree_by_Prim;
const
  max = 100;
  maxC = 10000;
var
  c: array[1..max, 1..max] of Integer;
  d: array[1..max] of Integer;
  Free: array[1..max] of Boolean;
  Trace: array[1..max] of Integer; {Vết, Trace[v] là đỉnh cha của  v trong cây khung nhỏ nhất}
  n, m: Integer;
  Connected: Boolean;

procedure LoadGraph;     {Nhập đồ thị}
var
  i, u, v: Integer;
begin
  ReadLn(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] := maxC; {Khởi tạo ma trận trọng số}
  for i := 1 to m do
    begin
      ReadLn(u, v, c[u, v]);
      c[v, u] := c[u, v];
    end;
end;

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

procedure Prim;
var
  k, i, u, v, min: Integer;
begin
  Connected := True;
  for k := 1 to n do
    begin
```

```
      u := 0; min := maxC; {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 dĩ 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] (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, W: Integer;
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, ') = ', c[Trace[v], v]);
          W := W + c[Trace[v], v];
        end;
      WriteLn('Weight = ', W);
    end;
end;

begin
  Assign(Input, 'MINTREE.INP'); Reset(Input);
  Assign(Output, 'MINTREE.OUT'); Rewrite(Output);
  LoadGraph;
  Init;
  Prim;
  PrintResult;
  Close(Input);
  Close(Output);
end.
```

Xét về độ phức tạp tính toán, thuật toán Prim có độ phức tạp là $O(n^2)$. Tương tự thuật toán Dijkstra, nếu kết hợp thuật toán Prim với cấu trúc Heap sẽ được một thuật toán với độ phức tạp O((m+n)logn).

Bài tập

1. Viết chương trình tạo đồ thị với số đỉnh ≤ 100, trọng số các cạnh là các số 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 và về tốc độ.

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 $0 < p(e) \leq 1$. 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.

5. Hãy thử cài đặt thuật toán Prim với cấu trúc dữ liệu Heap chứa các đỉnh ngoài cây, tương tự như đối với thuật toán Dijkstra.

§10. BÀI TOÁN LUỒNG CỰC ĐẠI TRÊN 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.

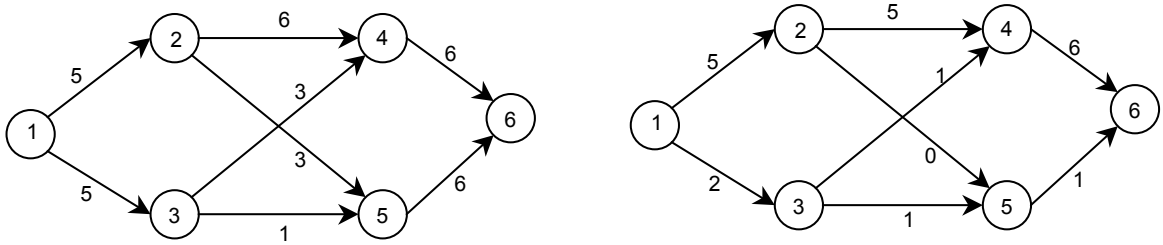
Nếu có mạng G = (V, E). Ta gọi luồng f trong mạng G là một phép gán cho mỗi cung e = (u, v) ∈ E một số thực không âm f(e) = f[u, v] gọi là luồng trên cung e, thoả mãn các điều kiện sau:

- Luồng trên mỗi cung không vượt quá khả năng thông qua của nó: $0 \le f[u, v] \le c[u, v]$ ($\forall$ (u, v) ∈ 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.



Hình 20: Mạng với các khả năng thông qua (1 phát, 6 thu) 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:

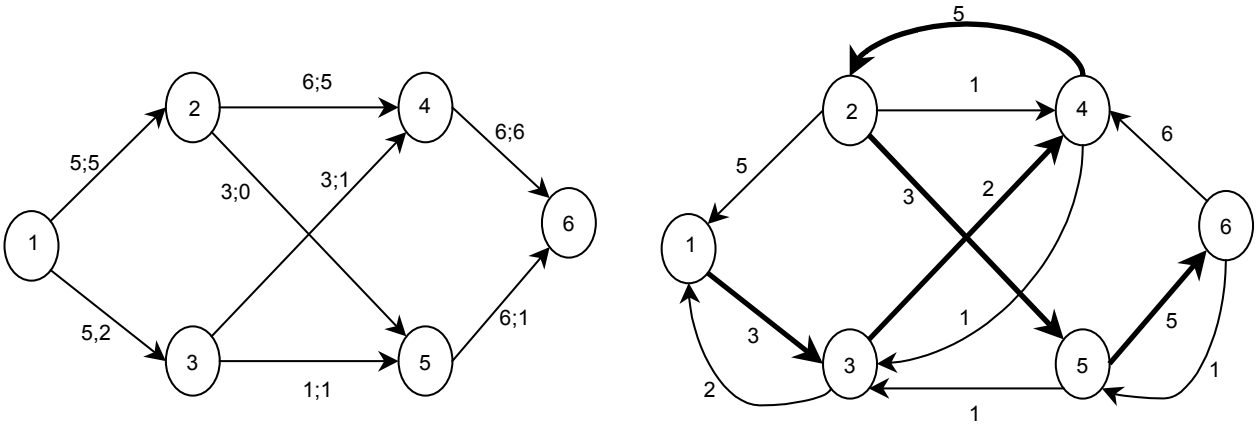
Giá trị luồng cực đại trên mạng đúng bằng khả năng thông qua của lát cắt hẹp nhất. Việc chứng minh định lý Ford- Fulkerson đã xây dựng được một thuật toán tìm luồng cực đại trên mạng:

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):

- 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**. Về ý nghĩa, trọng số cung này cho biết còn có thể tăng luồng f trên cung (u, v) một lượng không quá trọng số đó.
- 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**. Về ý nghĩa, trọng số cung này cho biết còn có thể giảm luồng f trên cung (u, v) một lượng không quá trọng số đó.

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



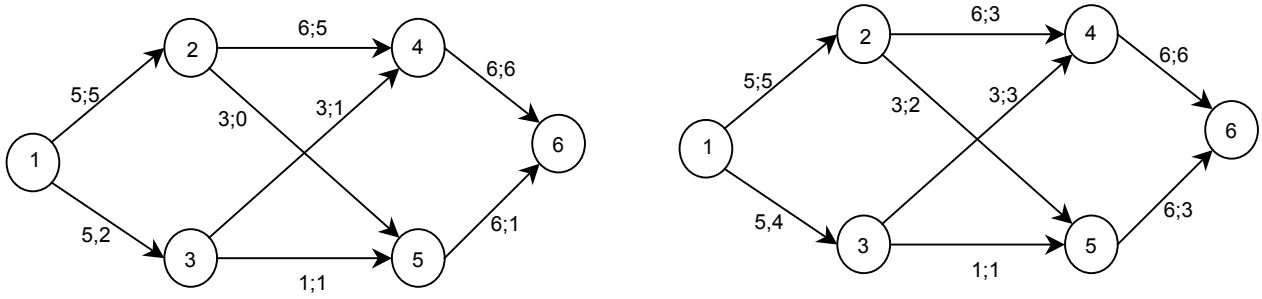
Hình 21: Mạng và luồng trên các cung (1 phát, 6 thu) và đồ thị tăng luồng tương ứng

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.



Hình 22: 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

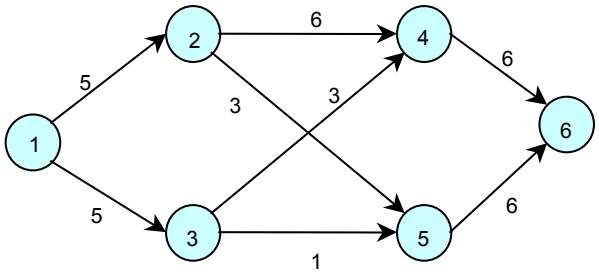
Bước 3: Thông báo giá trị luồng cực đại tìm được.

III. CÀI ĐẶT

Input: file văn bản MAXFLOW.INP. Trong đó:

- Dòng 1: Chứa số đỉnh n (≤ 100), số cạnh m của đồ thị, đỉnh phát A, đỉnh thu B theo đúng thứ tự cách nhau ít nhất một dấu cách
- m dòng tiếp theo, mỗi dòng có dạng ba số u, v, c[u, v] cách nhau ít nhất một dấu cách thể hiện có cung (u, v) trong mạng và khả năng thông qua của cung đó là c[u, v] (c[u, v] là số nguyên dương không quá 100)

Output: file văn bản MAXFLOW.OUT ghi luồng trên các cung và giá trị luồng cực đại tìm được



MAXFLOW.INP	MAXFLOW.OUT
6 8 1 6 1 2 5 1 3 5 2 4 6 2 5 3 3 4 3 3 5 1 4 6 6 5 6 6	f(1, 2) = 5 f(1, 3) = 4 f(2, 4) = 3 f(2, 5) = 2 f(3, 4) = 3 f(3, 5) = 1 f(4, 6) = 6 f(5, 6) = 3 Max Flow: 9

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.

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...

PROG10_1.PAS * Thuật toán tìm luồng cực đại trên mạng

```
program Max_Flow;
const
  max = 100;
  maxC = 10000;
var
  c, f, cf: array[1..max, 1..max] of Integer; {c: khả năng thông, f: Luồng}
  Trace: array[1..max] of Integer;
  n, A, B: Integer;

procedure Enter;      {Nhập mạng}
var
  m, i, u, v: Integer;
begin
  FillChar(c, SizeOf(c), 0);
  ReadLn(n, m, A, B);
  for i := 1 to m do
    ReadLn(u, v, c[u, v]);
end;

procedure CreateGf; {Tìm đồ thị tăng luồng, tức là xây dựng cf từ c và f}
var
  u, v: Integer;
begin
  for u := 1 to n do
    for v := 1 to n do cf[u, v] := maxC;
  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
  Queue: array[1..max] of Integer; {Hàng đợi dùng cho BFS}
  Free: array[1..max] of Boolean;
  u, v, First, Last: Integer;
begin
  FillChar(Free, SizeOf(Free), True);
  First := 1; Last := 1; Queue[1] := A; {Queue chỉ gồm một đỉnh phát A}
  Free[A] := False;                     {đánh dấu A}
  repeat
    u := Queue[First]; Inc(First);      {Lấy u khỏi Queue}
    for v := 1 to n do
      if Free[v] and (cf[u, v] <> maxC) 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
  u, v, IncValue: Integer;
begin
  {Trước hết dò đường theo vết để tìm trọng số nhỏ nhất của các cung trên đường}
  IncValue := maxC;
  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, m: Integer;
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]);
          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);
end;

begin
  Assign(Input, 'MAXFLOW.INP'); Reset(Input);
  Assign(Output, 'MAXFLOW.OUT'); Rewrite(Output);
  Enter;              {Nhập dữ liệu}
  FillChar(f, SizeOf(f), 0); {Khởi tạo luồng 0}
  repeat              {Bước lặp}
    CreateGf;                       {Dựng đồ thị tăng luồng}
    if not FindPath then Break;     {Nếu không tìm được đường tăng luồng thì thoát ngay}
    IncFlow;                        {Tăng luồng dọc đường tăng luồng}
  until False;
  PrintResult;
  Close(Input);
  Close(Output);
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]). Sau đó tăng luồng dọc theo đường tăng luồng nếu tìm thấy.

PROG10_2.PAS * Thuật toán Ford-Fulkerson

```
program Max_Flow_by_Ford_Fulkerson;
const
  max = 100;
  maxC = 10000;
var
  c, f: array[1..max, 1..max] of Integer;
  Trace: array[1..max] of Integer;
  Delta: array[1..max] of Integer;
  n, A, B: Integer;

procedure Enter;    {Nhập dữ liệu}
var
  m, i, u, v: Integer;
begin
  FillChar(c, SizeOf(c), 0);
  ReadLn(n, m, A, B);
  for i := 1 to m do
    ReadLn(u, v, c[u, v]);
end;

function Min(X, Y: Integer): Integer;
begin
  if X < Y then Min := X else Min := Y;
end;

function FindPath: Boolean;
var
  u, v: Integer;
  Queue: array[1..max] of Integer;
  First, Last: Integer;
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 cần nó khác 0 để đánh dấu mà thôi, số dương nào cũng được cả}
```

```
  Delta[A] := maxC;      {Khởi tạo nhãn}
  repeat
    u := Queue[First]; Inc(First);     {Lấy u khỏi Queue}
    for v := 1 to n do
      if Trace[v] = 0 then             {Xét những đỉnh v chưa đánh dấu thăm}
        begin
          if f[u, v] < c[u, v] then    {Nếu (u, v) là cung thuận trên Gf và có trọng số là c[u, v] - f[u, v]}
            begin
              Trace[v] := u;           {Lưu vết, Trace[v] mang dấu dương}
              Delta[v] := min(Delta[u], c[u, v] - f[u, v]);
            end
          else
            if f[v, u] > 0 then        {Nếu (u, v) là cung nghịch trên Gf và có trọng số là f[v, u]}
              begin
                Trace[v] := -u;        {Lưu vết, Trace[v] mang dấu âm}
                Delta[v] := min(Delta[u], f[v, u]);
              end;
          if Trace[v] <> 0 then        {Trace[v] khác 0 tức là từ u có thể thăm v}
            begin
              if v = B then            {Có đường tăng luồng từ A tới B}
                begin
                  FindPath := True; Exit;
                end;
              Inc(Last); Queue[Last] := v; {Đưa v vào Queue}
            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, u, v: Integer;
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;     {In kết quả}
var
  u, v, m: Integer;
begin
  m := 0;
  for u := 1 to n do
    for v := 1 to n do
      if c[u, v] > 0 then
        begin
          WriteLn('f(', u, ', ', v, ') = ', f[u, v]);
          if u = A then m := m + f[A, v];
        end;
  WriteLn('Max Flow: ', m);
end;

begin
```

```
  Assign(Input, 'MAXFLOW.INP'); Reset(Input);
  Assign(Output, 'MAXFLOW.OUT'); Rewrite(Output);
  Enter;
  FillChar(f, SizeOf(f), 0);
  repeat
    if not FindPath then Break;
    IncFlow;
  until False;
  PrintResult;
  Close(Input);
  Close(Output);
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 $A \in X$) và tập Y gồm những đỉnh còn lại (tất nhiên $B \in Y$) thì (X, Y) là lát cắt hẹp nhất đó. Có thể có nhiều lát cắt hẹp nhất, ví dụ nếu đặt tập Y gồm những đỉnh đến được đỉnh thu B trên đồ thị tăng luồng (tất nhiên $B \in Y$) và tập X gồm những đỉnh còn lại thì (X, Y) cũng là một lát cắt hẹp nhất.

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.

Định lý về chi phí thời gian thực hiện giải thuật:

Trong phương pháp Ford-Fulkerson, nếu dùng đường đi ngắn nhất (qua ít cạnh nhất) từ đỉnh phát tới đỉnh thu trên đồ thị tăng luồng thì cần ít hơn n.m lần chọn đường đi để tìm ra luồng cực đại.

Edmonds và Karp đã chứng minh tính chất này và đề nghị một phương pháp cải tiến: Tại mỗi bước, ta nên tìm đường tăng luồng sao cho giá trị tăng luồng được gia tăng nhiều nhất.

Nói chung đối với thuật toán Ford-Fulkerson, các đánh giá lý thuyết bị lệch rất nhiều so với thực tế, mặc dù với sự phân tích trong trường hợp xấu, chi phí thời gian thực hiện của thuật toán là khá lớn. Nhưng trên thực tế thì thuật toán này hoạt động rất nhanh và hiệu quả.

Bài tập:

1. Mạng với nhiều điểm phát và nhiều điểm thu: 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. Mạng với khả năng thông qua của các đỉnh và các cung: 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 khác với A và B 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.

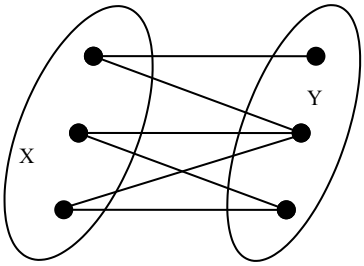
3. Lát cắt hẹp nhất: Cho một đồ thị liên thông gồm n đỉnh và m cạnh, hãy tìm cách bỏ đi một số ít nhất các cạnh để làm cho đồ thị mất đi tính liên thông

4. Tập đại diện: Một lớp học có n bạn nam, n bạn nữ. Cho m món quà lưu niệm, ($n \le m$). Mỗi bạn có sở thích về một số món quà nào đó. Hãy tìm cách phân cho mỗi bạn nam tặng một món quà cho một bạn nữ thoả mãn:

- Mỗi bạn nam chỉ tặng quà cho đúng một bạn nữ
- Mỗi bạn nữ chỉ nhận quà của đúng một bạn nam
- Bạn nam nào cũng đi tặng quà và bạn nữ nào cũng được nhận quà, món quà đó phải hợp sở thích của cả hai người.
- Món quà nào đã được một bạn nam chọn thì bạn nam khác không được chọn nữa.

§11. BÀI TOÁN TÌM BỘ GHÉP CỰC ĐẠI TRÊN ĐỒ THỊ HAI PHÍA

I. ĐỒ THỊ HAI PHÍA (BIPARTITE GRAPH)



Các tên gọi đồ thị hai phía, đồ thị lưỡng phân, đồ thị phân đôi, 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 hai phía 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 hai phía 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 hai phía 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...

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

Cho một đồ thị hai phía 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.

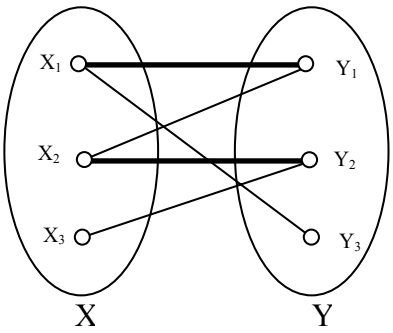
- Các đỉnh trong M gọi là các đỉnh đã ghép (matched vertices), các đỉnh khác là chưa ghép.
- Các cạnh trong M gọi là các cạnh đã ghép, các cạnh khác là chưa ghép

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 về X. Trên đồ thị định hướng đó: Một đường đi xuất phát từ một X_đỉnh chưa ghép gọi là đường pha, một đường đi từ một X_đỉnh chưa ghép tới một Y_đỉnh chưa ghép gọi là đường mở.

Một cách dễ hiểu, có thể quan niệm như sau:

- 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** sang Y, rồi đến một cạnh **đã ghép** về X, rồi lại đến một cạnh **chưa ghép** sang Y... 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 hai phía như hình bên, và bộ ghép
M = {(X1, Y1), (X2, 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:

- 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.

```
<Khởi tạo một bộ ghép M>;
while <Có đường mở xuất phát từ x tới một đỉnh y chưa ghép ∈Y> do
     <Dọc trên đường mở, xoá bỏ khỏi M các cạnh đã ghép và thêm vào M những cạnh
     chưa ghép, đỉnh x và y trở thành đã ghép, số cạnh đã ghép tăng lên 1>;
```

Như ví dụ trên, với bộ ghép hai cạnh M = {(X1, Y1), (X2, Y2)} 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 hai phía

Giả sử đồ thị hai phía 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 hai phía này bằng ma trận A cỡ mxn. Trong đó:

A[i, j] = TRUE ⇔ 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.

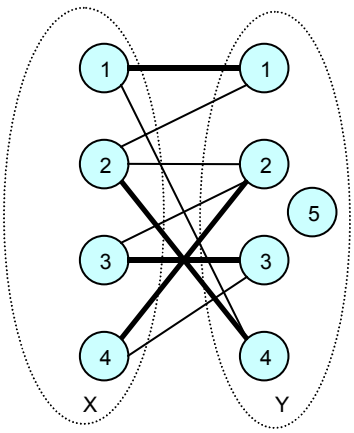
Vì đường mở bắt đầu từ một X_đỉnh chưa ghép, đi theo một cạnh chưa ghép sang tập Y, rồi theo một đã ghép để về tập X, rồi lại một cạnh chưa ghép sang tập Y ... **cuối cùng là cạnh chưa ghép** tới một Y_đỉnh chưa ghép. Nên có thể thấy ngay rằng độ dài đường mở là lẻ và trên đường mở số cạnh $\in$ M ít hơn số cạnh $\notin$ M là 1 cạnh. Và cũng dễ thấy rằng giải thuật tìm đường mở nên sử dụng thuật toán tìm kiếm theo chiều rộng để đường mở tìm được là đường đi ngắn nhất, giảm bớt công việc cho bước tăng cặp ghép.

Ta khởi tạo một hàng đợi (Queue) ban đầu chứa tất cả các X_đỉnh chưa ghép. Thuật toán tìm kiếm theo chiều rộng làm việc theo nguyên tắc lấy một đỉnh v khỏi Queue và lại đẩy Queue những nối từ v chưa được thăm. Như vậy nếu thăm tới một Y_đỉnh chưa ghép thì tức là ta tìm đường mở kết thúc ở Y_đỉnh chưa ghép đó, quá trình tìm kiếm dừng ngay. Còn nếu ta thăm tới một đỉnh j $\in$ Y đã ghép, dựa vào sự kiện: **từ j chỉ có thể tới được matchY[j]** theo duy nhất một cạnh đã ghép định hướng ngược từ Y về X, nên ta có thể **đánh dấu thăm j, thăm luôn cả matchY[j], và đẩy vào Queue phần tử matchY[j] $\in$ X** (Thăm liền 2 bước).

Input: file văn bản MATCH.INP

- Dòng 1: chứa hai số m, n (m, n ≤ 100) theo thứ tự là số X_đỉnh và số Y_đỉnh cách nhau ít nhất một dấu cách
- Các dòng tiếp theo, mỗi dòng ghi hai số i, j cách nhau ít nhất một dấu cách thể hiện có cạnh nối hai đỉnh (X[i], Y[j]) .

Output: file văn bản MATCH.OUT chứa bộ ghép cực đại tìm được

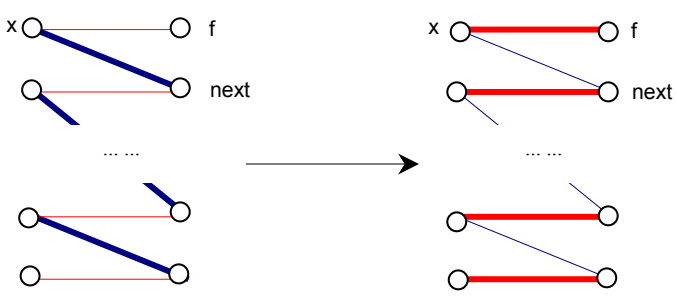


MATCH.INP	MATCH.OUT
4 5 1 1 1 4 2 1 2 2 2 4 3 2 3 3 4 2 4 3	Match: 1) X[1] - Y[1] 2) X[2] - Y[4] 3) X[3] - Y[3] 4) X[4] - Y[2]

PROG11_1.PAS * Thuật toán đường mở tìm bộ ghép cực đại

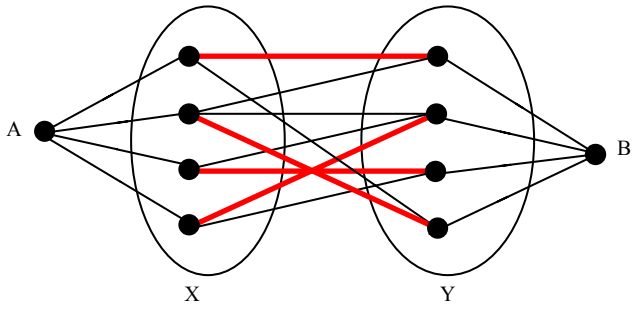
```
program MatchingProblem;
const
  max = 100;
var
  m, n: Integer;
  a: array[1..max, 1..max] of Boolean;
  matchX, matchY: array[1..max] of Integer;
  Trace: array[1..max] of Integer;

procedure Enter;     {Đọc dữ liệu, (từ thiết bị nhập chuẩn)}
var
  i, j: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(m, n);
```

```
  while not SeekEof do
    begin
      ReadLn(i, j);
      a[i, j] := True;
    end;
end;

procedure Init;      {Khởi tạo bộ ghép rỗng}
begin
  FillChar(matchX, SizeOf(matchX), 0);
  FillChar(matchY, SizeOf(matchY), 0);
end;

{Tìm đường mở, nếu thấy trả về một Y_đỉnh chưa ghép là đỉnh kết thúc đường mở, nếu không thấy trả về 0}
function FindAugmentingPath: Integer;
var
  Queue: array[1..max] of Integer;
  i, j, first, last: Integer;
begin
  FillChar(Trace, SizeOf(Trace), 0);  {Trace[j] = X_đỉnh liền trước Y[j] trên đường mở}
  last := 0;                          {Khởi tạo hàng đợi rỗng}
  for i := 1 to m do                  {Đẩy tất cả những X_đỉnh chưa ghép vào hàng đợi}
    if matchX[i] = 0 then
      begin
        Inc(last);
        Queue[last] := i;
      end;
  {Thuật toán tìm kiếm theo chiều rộng}
  first := 1;
  while first <= last do
    begin
      i := Queue[first]; Inc(first);  {Lấy một X_đỉnh ra khỏi Queue (X[i])}
      for j := 1 to n do              {Xét những Y_đỉnh chưa thăm kề với X[i] qua một cạnh chưa ghép}
        if (Trace[j] = 0) and a[i, j] and (matchX[i] <> j) then
          begin {lệnh if trên hơi thừa đk matchX[i] <> j, điều kiện Trace[j] = 0 đã bao hàm luôn điều kiện này rồi}
            Trace[j] := i;            {Lưu vết đường đi}
            if matchY[j] = 0 then     {Nếu j chưa ghép thì ghi nhận đường mở và thoát ngay}
              begin
                FindAugmentingPath := j;
                Exit;
              end;
            Inc(last);                {Đẩy luôn matchY[j] vào hàng đợi}
            Queue[last] := matchY[j];
          end;
    end;
  FindAugmentingPath := 0;    {Ở trên không Exit được tức là không còn đường mở}
end;

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

procedure Solve;    {Thuật toán đường mở}
var
```



```
  finish: Integer;
begin
  repeat
    finish := FindAugmentingPath;    {Đầu tiên thử tìm một đường mở}
    if finish <> 0 then Enlarge(finish);    {Nếu thấy thì tăng cặp và lặp lại}
  until finish = 0;                          {Nếu không thấy thì dừng}
end;

procedure PrintResult;     {In kết quả}
var
  i, Count: Integer;
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
  Assign(Input, 'MATCH.INP'); Reset(Input);
  Assign(Output, 'MATCH.OUT'); Rewrite(Output);
  Enter;
  Init;
  Solve;
  PrintResult;
  Close(Input);
  Close(Output);
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.

Người ta đã chứng minh được chi phí thời gian thực hiện giải thuật này trong trường hợp xấu nhất sẽ là $O(n^3)$ đối với đồ thị dày và $O(n(n + m)\log n)$ đối với đồ thị thưa. Tuy nhiên, cũng giống như thuật toán Ford-Fulkerson, trên thực tế phương pháp này hoạt động rất nhanh.

Bài tập

1. Có n thợ và n công việc (n ≤ 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 ≤ 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.
3. Có n thợ và m công việc (n, m ≤ 100). Mỗi thợ cho biết mình có thể làm được những việc nào, hãy phân công thực hiện các công việc đó sao cho số công việc phân cho người thợ làm nhiều nhất thực hiện là cực tiểu.

§12. 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 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 hơn chi phí của mọi phép phân công có thể. 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 hai phía đầy đủ** (Đồ thị hai phía 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[i, j] = trọng số cạnh nối đỉnh X_i với Y_j. Giả thiết c[i, j] ≥ 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.

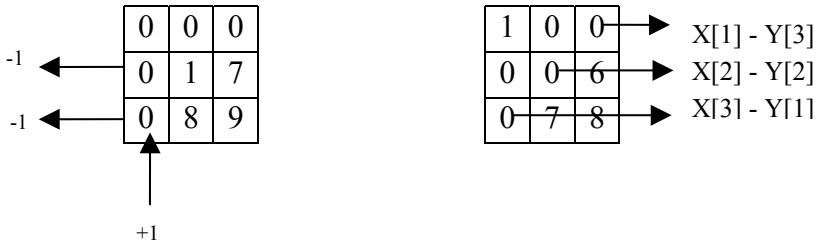
Chứng minh: Với một bộ ghép đầy đủ bất kỳ thì có một và chỉ một cạnh ghép với X[i]. Nên việc cộng thêm Δ vào tất cả các cạnh liên thuộc với X[i] sẽ làm tăng trọng số bộ ghép đó lên Δ. Vì vậy

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.

Nếu ta định hướng lại các 0_cạnh như sau: Những 0_cạnh chưa ghép cho hướng từ tập X sang tập Y, những 0_cạnh đã ghép cho hướng từ tập Y về tập X. Khi đó:

- Đường pha (Alternating Path) là một đường đi cơ bản xuất phát từ một X_đỉnh chưa ghép đi theo các 0_cạnh đã định hướng ở trên. Như vậy dọc trên đường pha, các 0_cạnh chưa ghép và những 0_cạnh đã ghép xen kẽ nhau. Vì đường pha chỉ là đường đi cơ bản trên đồ thị định hướng nên việc xác định những đỉnh nào có thể đến được từ x ∈ 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 đi từ một X_đỉnh chưa ghép tới một Y_đỉnh chưa ghép. Như vậy:
  - 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.
  - 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 $\Delta > 0$ bởi nếu $\Delta = 0$ 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 $y \in$ VisitedY, điều này vô lý.
  - ❖ Biến đổi đồ thị G như sau: Với $\forall x \in$ VisitedX, trừ Δ vào trọng số những cạnh liên thuộc với x, Với $\forall y \in$ 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.

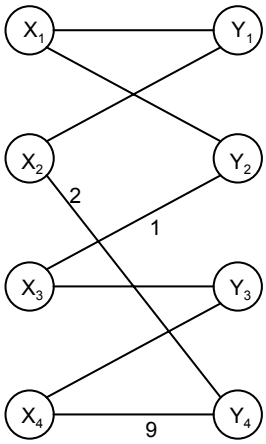
Bước 3: Sau bước 2 thì mọi X_đỉnh đều được ghép, in kết quả về bộ ghép tìm được.

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.

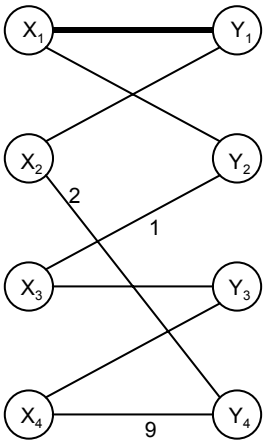


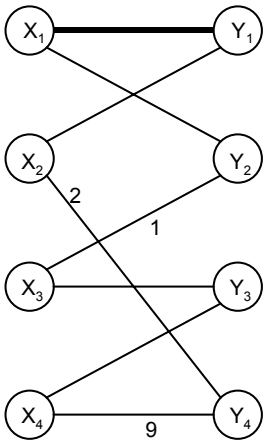
$x^* = X_1$

Tìm được đường mở:

$X_1 \rightarrow Y_1$

Tăng cặp



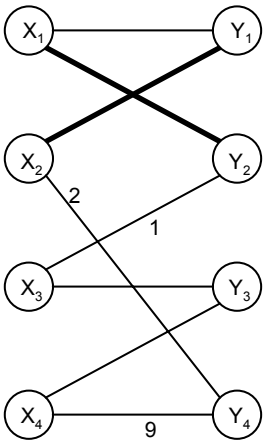


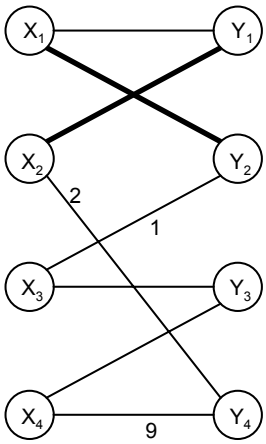
$x^* = X_2$

Tìm được đường mở:

$X_2 \rightarrow Y_1 \rightarrow X_1 \rightarrow Y_2$

Tăng cặp



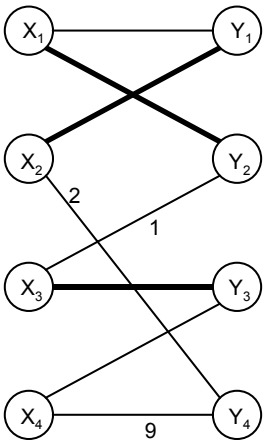


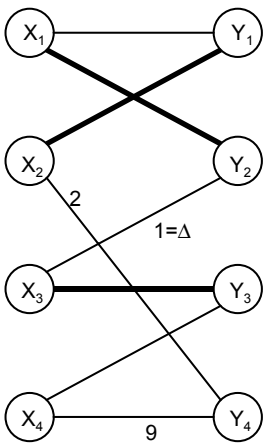
$x^* = X_3$

Tìm được đường mở:

$X_3 \rightarrow Y_3$

Tăng cặp





$x^* = X_4$

Không tìm được đường mở:

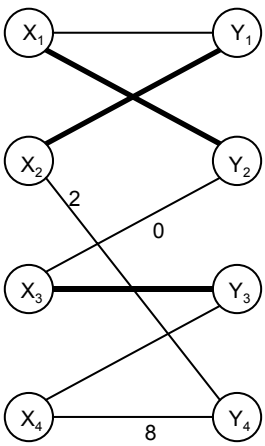
Tập những X_đỉnh đến được từ X_4 bằng một đường pha: $\{X_3, X_4\}$

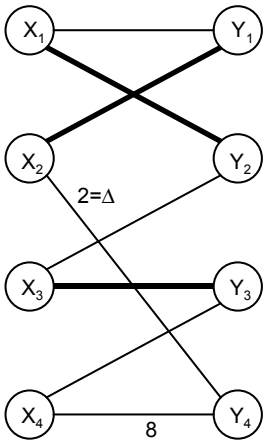
Tập những Y_đỉnh đến được từ X_4 bằng một đường pha: $\{Y_3\}$

Giá trị xoay $\Delta = 1$ (Cạnh X_3-Y_2)

Trừ tất cả trọng số những cạnh liên thuộc với $\{X_3, X_4\}$ đi 1

Cộng tất cả trọng số những cạnh liên thuộc với Y_3 lên 1





$x^* = X_4$

Vẫn không tìm được đường mở:

Tập những X_đỉnh đến được từ X_4 bằng một đường pha:

$\{X_1, X_2, X_3, X_4\}$

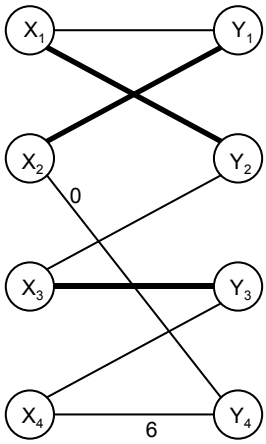
Tập những Y_đỉnh đến được từ X_4 bằng một đường pha:

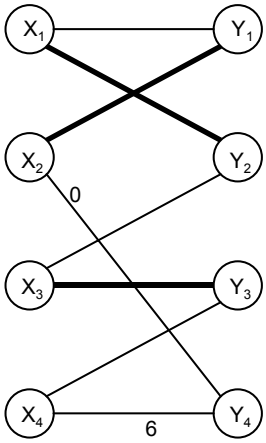
$\{Y_1, Y_2, Y_3\}$

Giá trị xoay $\Delta = 2$ (Cạnh X_2-Y_4)

Trừ tất cả trọng số những cạnh liên thuộc với $\{X_1, X_2, X_3, X_4\}$ đi 2

Cộng tất cả trọng số những cạnh liên thuộc với $\{Y_1, Y_2, Y_3\}$ lên 2





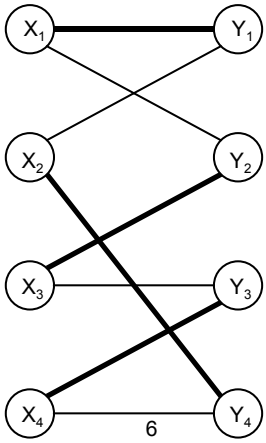
$x^* = X_4$

Tìm được 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 $x \in X$, 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 Kônig 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] tức là 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.

	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].**

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 $\Delta := \min\{c[i, j] - Fx[i] - Fy[j] \mid \forall X[i] \in VisitedX; \forall Y[j] \notin VisitedY\}$
  - Với $\forall X[i] \in VisitedX$: $Fx[i] := Fx[i] + \Delta$;
  - Với $\forall Y[j] \in VisitedY$: $Fy[j] := Fy[j] - \Delta$;
  - 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;

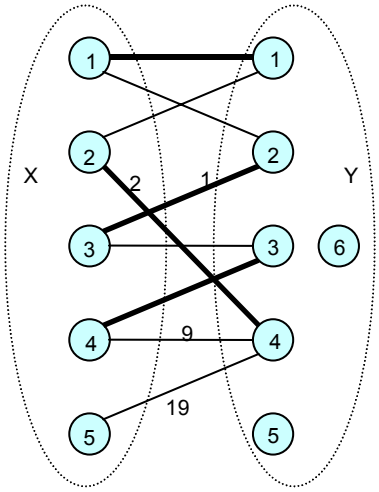
b) Tìm đường mở như thế nào

Ta sẽ tìm đường mở và xây dựng hai tập VisitedX và VisitedY bằng thuật toán tìm kiếm theo chiều rộng chỉ xét tới những đỉnh và những 0_cạnh đã định hướng như đã nói trong phần đầu:

Khởi tạo một hàng đợi (Queue) ban đầu chỉ có một đỉnh x^*. Thuật toán tìm kiếm theo chiều rộng làm việc theo nguyên tắc lấy một đỉnh v khỏi Queue và lại đẩy Queue những nối từ v chưa được thăm. Như vậy nếu thăm tới một Y_đỉnh chưa ghép thì tức là ta tìm đường mở kết thúc ở Y_đỉnh chưa ghép đó, quá trình tìm kiếm dừng ngay. Còn nếu ta thăm tới một đỉnh $y \in Y$ đã ghép, dựa vào sự kiện: **từ y chỉ có thể tới được matchY[y]** theo duy nhất một 0_cạnh định hướng, nên ta có thể **đánh dấu thăm y, thăm luôn cả matchY[y], và đẩy vào Queue phần tử matchY[y] ∈ X**.

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 ($m, n \le 100$)
- 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] ($1 \le i \le m; 1 \le j \le n; 0 \le c[i, j] \le 100$).



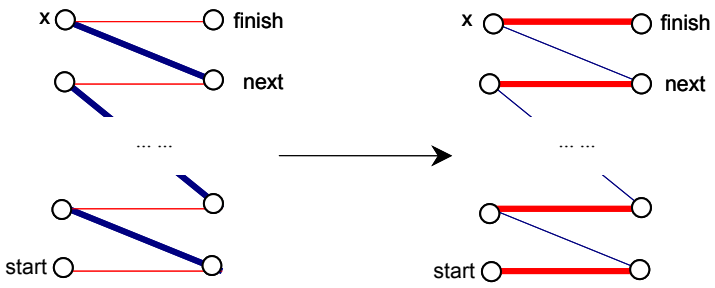
ASSIGN.INP	ASSIGN.OUT
5 6 1 1 0 1 2 0 2 1 0 2 4 2 3 2 1 3 3 0 4 3 0 4 4 9 5 4 9	Optimal assignment: 1) X[1] - Y[1] 0 2) X[2] - Y[4] 2 3) X[3] - Y[2] 1 4) X[4] - Y[3] 0 Cost: 3

PROG12_1.PAS * Thuật toán Hungari

```
program AssignmentProblemSolve;
const
  max = 100;
  maxC = 10001;
var
  c: array[1..max, 1..max] of Integer;
  Fx, Fy, matchX, matchY, Trace: array[1..max] of Integer;
  m, n, k, start, finish: Integer;     {đường mở sẽ bắt đầu từ start∈X và kết thúc ở finish∈Y}

procedure Enter;       {Nhập dữ liệu từ thiết bị nhập chuẩn (Input)}
var
  i, j: Integer;
begin
  ReadLn(m, n);
  if m > n then k := m else k := n;
  for i := 1 to k do
    for j := 1 to k do c[i, j] := maxC;
  while not SeekEof do ReadLn(i, j, c[i, j]);
end;

procedure Init;        {Khởi tạo}
var
  i, j: Integer;
begin
  {Bộ ghép rỗng}
  FillChar(matchX, SizeOf(matchX), 0);
  FillChar(matchY, SizeOf(matchY), 0);
  {Fx[i] := Trọng số nhỏ nhất của các cạnh liên thuộc với X[i]}
  for i := 1 to k do
    begin
      Fx[i] := maxC;
      for j := 1 to k do
        if c[i, j] < Fx[i] then Fx[i] := c[i, j];
    end;
  {Fy[j] := Trọng số nhỏ nhất của các cạnh liên thuộc với Y[j]}
  for j := 1 to k do
    begin
      Fy[j] := maxC;
      for i := 1 to k do   {Lưu ý là trọng số cạnh (x[i], y[j]) bây giờ là c[i, j] - Fx[i] chứ không còn là c[i, j] nữa}
        if c[i, j] - Fx[i] < Fy[j] then Fy[j] := c[i, j] - Fx[i];
    end;
  {Việc khởi tạo các Fx và Fy như thế này chỉ đơn giản là để cho số 0_cạnh trở nên càng nhiều càng tốt mà thôi}
  {Ta hoàn toàn có thể khởi gán các Fx và Fy bằng giá trị 0}
```

```
end;
{Hàm cho biết trọng số cạnh (X[i], Y[j]) }
function GetC(i, j: Integer): Integer;
begin
  GetC := c[i, j] - Fx[i] - Fy[j];
end;

procedure FindAugmentingPath;  {Tìm đường mở bắt đầu ở start}
var
  Queue: array[1..max] of Integer;
  i, j, first, last: Integer;
begin
  FillChar(Trace, SizeOf(Trace), 0);        {Trace[j] = X_đỉnh liền trước Y[j] trên đường mở}
  {Thuật toán BFS}
  Queue[1] := start;                        {Đẩy start vào hàng đợi}
  first := 1; last := 1;
  repeat
    i := Queue[first]; Inc(first);  {Lấy một đỉnh X[i] khỏi hàng đợi}
    for j := 1 to k do              {Duyệt những Y_đỉnh chưa thăm kề với X[i] qua một 0_cạnh chưa ghép}
      if (Trace[j] = 0) and (GetC(i, j) = 0) then
        begin
          Trace[j] := i;                {Lưu vết đường đi, cùng với việc đánh dấu (≠0) luôn}
          if matchY[j] = 0 then         {Nếu j chưa ghép thì ghi nhận nơi kết thúc đường mở và thoát luôn}
            begin
              finish := j;
              Exit;
            end;
          Inc(last); Queue[last] := matchY[j];  {Đẩy luôn matchY[j] vào Queue}
        end;
  until first > last;    {Hàng đợi rỗng}
end;

procedure SubX_AddY;        {Xoay các trọng số cạnh}
var
  i, j, t, Delta: Integer;
  VisitedX, VisitedY: set of Byte;
begin
  (* Để ý rằng:
     VisitedY = {y | Trace[y] ≠ 0}
     VisitedX = {start} ∪ match(VisitedY) = {start} ∪ {matchY[y] | Trace[y] ≠ 0}
  *)
  VisitedX := [start];
  VisitedY := [];
  for j := 1 to k do
    if Trace[j] <> 0 then
      begin
        Include(VisitedX, matchY[j]);
        Include(VisitedY, j);
      end;
  {Sau khi xác định được VisitedX và VisitedY, ta tìm Δ là trọng số nhỏ nhất của cạnh nối từ VisitedX ra Y\VisitedY}
  Delta := maxC;
  for i := 1 to k do
    if i in VisitedX then
      for j := 1 to k do
        if not (j in VisitedY) and (GetC(i, j) < Delta) then
          Delta := GetC(i, j);
  {Xoay trọng số cạnh}
  for t := 1 to k do
    begin
      {Trừ trọng số những cạnh liên thuộc với VisitedX đi Delta}
      if t in VisitedX then Fx[t] := Fx[t] + Delta;
      {Cộng trọng số những cạnh liên thuộc với VisitedY lên Delta}
      if t in VisitedY then Fy[t] := Fy[t] - Delta;
    end;
```

```
end;
{Nới rộng bộ ghép bởi đường mở tìm được}
procedure Enlarge;
var
  x, next: Integer;
begin
  repeat
    x := Trace[finish];
    next := matchX[x];
    matchX[x] := finish;
    matchY[finish] := x;
    finish := Next;
  until finish = 0;
end;
```




```
procedure Solve;    {Thuật toán Hungari}
var
  x, y: Integer;
begin
  for x := 1 to k do
    begin
      start := x; finish := 0; {Khởi gán nơi xuất phát đường mở, finish = 0 nghĩa là chưa tìm thấy đường mở}
      repeat
        FindAugmentingPath;     {Thử tìm đường mở}
        if finish = 0 then SubX_AddY; {Nếu không thấy thì xoay các trọng số cạnh và lặp lại}
      until finish <> 0;        {Cho tới khi tìm thấy đường mở}
      Enlarge;    {Tăng cặp dựa trên đường mở tìm được}
    end;
end;

procedure Result;
var
  x, y, Count, W: Integer;
begin
  WriteLn('Optimal assignment:');
  W := 0; Count := 0;
  for x := 1 to m do {In ra phép phân công thì chỉ cần xét đến m, không cần xét đến k}
    begin
      y := matchX[x];
      {Những cạnh có trọng số maxC tương ứng với một thợ không được giao việc và một việc không được phân công}
      if c[x, y] < maxC then
        begin
          Inc(Count);
          WriteLn(Count:5, ') X[', x, '] - Y[', y, ']  ', c[x, y]);
          W := W + c[x, y];
        end;
    end;
  WriteLn('Cost: ', W);
end;

begin
  Assign(Input, 'ASSIGN.INP'); Reset(Input);
  Assign(Output, 'ASSIGN.OUT'); Rewrite(Output);
  Enter;
  Init;
  Solve;
  Result;
  Close(Input);
  Close(Output);
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 trên đồ thị ban đầu.
3. Một vấn đề nữa phải hết sức cẩn thận trong việc ước lượng độ lớn của các phần tử Fx và Fy. Nếu như giả thiết cho các trọng số không quá 500 thì ta không thể dựa vào bất đẳng thức $Fx(x) + Fy(y) \le c(x, y)$ mà khẳng định các phần tử trong Fx và Fy cũng ≤ 500. Hãy tự tìm ví dụ để hiểu rõ hơn bản chất thuật toán.

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

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 mà không cần đổi dấu trọng số. 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 $\Delta := \min\{Fx[i] + Fy[j] - c[i, j] \mid \forall X[i] \in VisitedX; \forall Y[j] \notin VisitedY\}$
  - 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.

VI. ĐỘ PHỨC TẠP TÍNH TOÁN

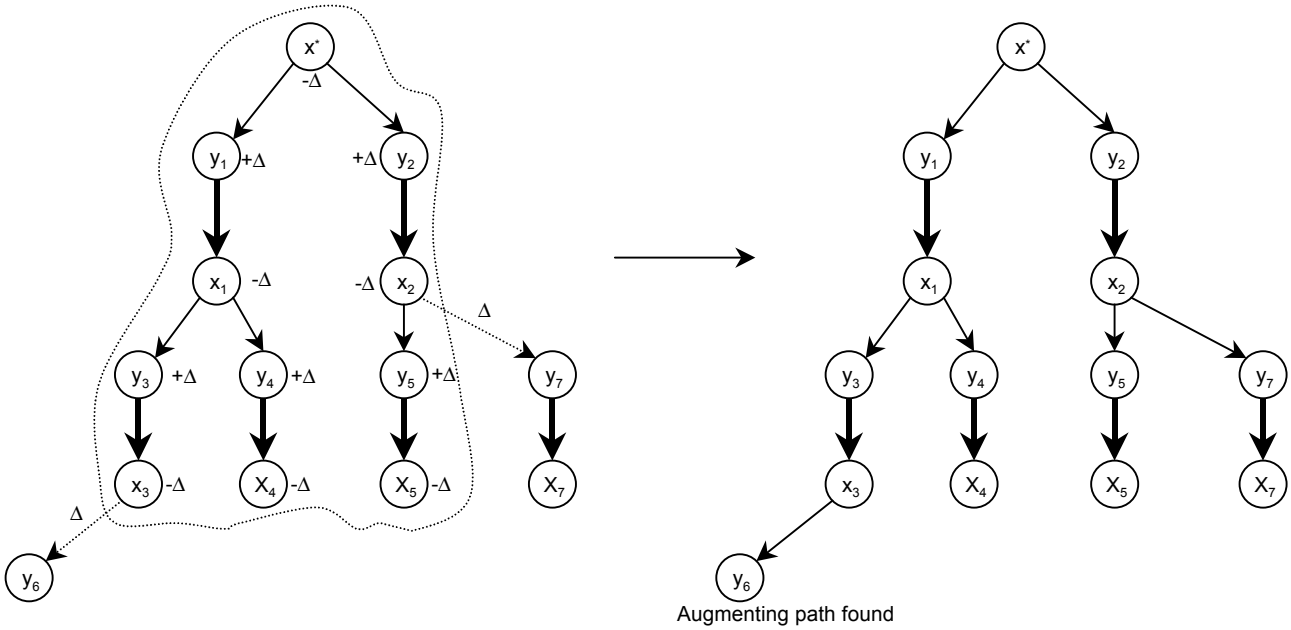
Dựa vào mô hình cài đặt thuật toán Kuhn-Munkres ở trên, ta có thể đánh giá về độ phức tạp tính toán lý thuyết của cách cài đặt này:

Thuật toán tìm kiếm theo chiều rộng được sử dụng để tìm đường mở có độ phức tạp $O(k^2)$, mỗi lần xoay trọng số cạnh mất một chi phí thời gian cỡ $O(k^2)$. Vậy mỗi lần tăng cặp, cần tối đa k lần dò đường và k lần xoay trọng số cạnh, mất một chi phí thời gian cỡ $O(k^3)$. Thuật toán cần k lần tăng cặp nên độ phức tạp tính toán trên lý thuyết của phương pháp này cỡ $O(k^4)$.

Có thể cải tiến mô hình cài đặt để được một thuật toán với độ phức tạp $O(k^3)$ dựa trên những nhận xét sau:

Nhận xét 1:

Quá trình tìm kiếm theo chiều rộng bắt đầu từ một đỉnh x^* chưa ghép cho ta một cây pha gốc x^*. Nếu tìm được đường mở thì dừng lại và tăng cặp ngay, nếu không thì xoay trọng số cạnh và bắt đầu tìm kiếm lại để được một cây pha mới lớn hơn cây pha cũ:



Hình 23: Cây pha "mọc" lớn hơn sau mỗi lần xoay trọng số cạnh và tìm đường

Nhận xét 2:

Việc xác định trọng số nhỏ nhất của cạnh nối một X_đỉnh trong cây pha với một Y_đỉnh ngoài cây pha có thể kết hợp ngay trong bước dựng cây pha mà không làm tăng cấp phức tạp tính toán. Để thực hiện điều này, ta sử dụng kỹ thuật như trong thuật toán Prim:

Với mọi $y \in Y$, gọi d[y] := khoảng cách từ y đến cây pha gốc x^*. Ban đầu d[y] được khởi tạo bằng trọng số cạnh $(x^*, y) = c[x^*, y] - Fx[x^*] - Fy[y]$ (cây pha ban đầu chỉ có đúng một đỉnh x^*).

Trong bước tìm đường bằng BFS, mỗi lần rút một đỉnh x ra khỏi Queue, ta xét những đỉnh $y \in Y$ chưa thăm và đặt lại $d[y]_{mới} := \min(d[y]_{cũ}$, trọng số cạnh (x, y)) sau đó mới kiểm tra xem (x, y) có phải là 0_cạnh hay không để tiếp tục các thao tác như trước. Nếu quá trình BFS không tìm ra đường mở thì giá trị xoay Δ chính là giá trị nhỏ nhất trong các d[y] dương. Ta bớt được một đoạn chương trình tìm giá trị xoay có độ phức tạp $O(k^2)$. Công việc tại mỗi bước xoay chỉ là tìm giá trị nhỏ nhất trong các d[y] dương và thực hiện phép cộng, trừ trên hai dãy đối ngẫu Fx và Fy, nó có độ phức tạp tính toán O(k), tối đa có k lần xoay để tìm đường mở nên tổng chi phí thời gian thực hiện các lần xoay cho tới khi tìm ra đường mở cỡ $O(k^2)$. Lưu ý rằng đồ thị đang xét là đồ thị hai phía đầy đủ nên sau khi xoay các trọng số cạnh bằng giá trị xoay Δ, tất cả các cạnh nối từ X_đỉnh trong cây pha tới

Y_đỉnh ngoài cây pha đều bị giảm trọng số đi Δ, chính vì vậy ta phải trừ tất cả các d[y] > 0 đi Δ để giữ được tính hợp lý của các d[y].

Nhận xét 3:

Ta có thể tận dụng kết quả của quá trình tìm kiếm theo chiều rộng ở bước trước để nới rộng cây pha cho bước sau (grow alternating tree) mà không phải tìm lại từ đầu (BFS lại bắt đầu từ x^*).

Khi không tìm thấy đường mở, quá trình tìm kiếm theo chiều rộng sẽ đánh dấu được những đỉnh đã thăm (thuộc cây pha) và hàng đợi các X_đỉnh trong quá trình tìm kiếm trở thành rỗng. Tiếp theo là phải xác định được Δ = trọng số nhỏ nhất của cạnh nối một X_đỉnh đã thăm với một Y_đỉnh chưa thăm và xoay các trọng số cạnh để những cạnh này trở thành 0_cạnh. Tại đây ta sẽ dùng kỹ thuật sau: Thăm luôn những đỉnh $y \in Y$ chưa thăm tạo với một X_đỉnh đã thăm một 0_cạnh (những Y_đỉnh chưa thăm có d[y] = 0), nếu tìm thấy đường mở thì dừng ngay, nếu không thấy thì đẩy tiếp những đỉnh matchY[y] vào hàng đợi và lặp lại thuật toán tìm kiếm theo chiều rộng bắt đầu từ những đỉnh này. Vậy nếu xét tổng thể, mỗi lần tăng cặp ta chỉ thực hiện một lần dựng cây pha, tức là tổng chi phí thời gian của những lần thực hiện giải thuật tìm kiếm trên đồ thị sau mỗi lần tăng cặp chỉ còn là $O(k^2)$.

Nhận xét 4:

Thủ tục tăng cặp dựa trên đường mở (Enlarge) có độ phức tạp O(k)

Từ 3 nhận xét trên, phương pháp đối ngẫu Kuhn-Munkres có thể cài đặt bằng một chương trình có độ phức tạp tính toán $O(k^3)$ bởi nó cần k lần tăng cặp và chi phí cho mỗi lần là $O(k^2)$.

PROG12_2.PAS * Cài đặt phương pháp Kuhn-Munkres $O(n^3)$

```
program AssignmentProblemSolve;
const
  max = 100;
  maxC = 10001;
var
  c: array[1..max, 1..max] of Integer;
  Fx, Fy, matchX, matchY: array[1..max] of Integer;
  Trace, Queue, d, arg: array[1..max] of Integer;
  first, last: Integer;
  start, finish: Integer;
  m, n, k: Integer;

procedure Enter;       {Nhập dữ liệu}
var
  i, j: Integer;
begin
  ReadLn(m, n);
  if m > n then k := m else k := n;
  for i := 1 to k do
    for j := 1 to k do c[i, j] := maxC;
  while not SeekEof do ReadLn(i, j, c[i, j]);
end;

procedure Init;       {Khởi tạo bộ ghép rỗng và hai dãy đối ngẫu Fx, Fy}
var
  i, j: Integer;
begin
  FillChar(matchX, SizeOf(matchX), 0);
  FillChar(matchY, SizeOf(matchY), 0);
  for i := 1 to k do
    begin
      Fx[i] := maxC;
      for j := 1 to k do
        if c[i, j] < Fx[i] then Fx[i] := c[i, j];
```

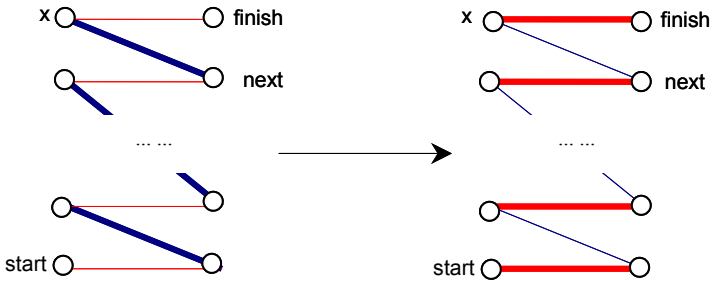
```
    end;
  for j := 1 to k do
    begin
      Fy[j] := maxC;
      for i := 1 to k do
        if c[i, j] - Fx[i] < Fy[j] then Fy[j] := c[i, j] - Fx[i];
    end;
end;

function GetC(i, j: Integer): Integer;      {Hàm trả về trọng số cạnh (X[i], Y[j])}
begin
  GetC := c[i, j] - Fx[i] - Fy[j];
end;

procedure InitBFS;     {Thủ tục khởi tạo trước khi tìm cách ghép start∈X}
var
  y: Integer;
begin
  {Hàng đợi chỉ gồm mỗi một đỉnh Start ⇔ cây pha khởi tạo chỉ có 1 đỉnh start}
  first := 1; last := 1;
  Queue[1] := start;
  {Khởi tạo các Y_đỉnh đều chưa thăm ⇔ Trace[y] = 0, ∀y}
  FillChar(Trace, SizeOf(Trace), 0);
  {Khởi tạo các d[y]}
  for y := 1 to k do
    begin
      d[y] := GetC(start, y);      {d[y] là khoảng cách từ y tới cây pha gốc start}
      arg[y] := start;             {arg[y] là X_đỉnh thuộc cây pha tạo ra khoảng cách đó}
    end;
  finish := 0;
end;

procedure Push(v: Integer);        {Đẩy một đỉnh v∈X vào hàng đợi}
begin
  Inc(last); Queue[last] := v;
end;

function Pop: Integer;             {Rút một X_đỉnh khỏi hàng đợi, trả về trong kết quả hàm}
begin
  Pop := Queue[first]; Inc(first);
end;

procedure FindAugmentingPath;      {Thủ tục tìm đường mở}
var
  i, j, w: Integer;
begin
  repeat
    i := Pop;                      {Rút một đỉnh X[i] khỏi hàng đợi}
    for j := 1 to k do             {Quét những Y_đỉnh chưa thăm}
      if Trace[j] = 0 then
        begin
          w := GetC(i, j);         {xét cạnh (X[i], Y[j])}
          if w = 0 then            {Nếu là 0_cạnh}
            begin
              Trace[j] := i;       {Lưu vết đường đi}
              if matchY[j] = 0 then    {Nếu j chưa ghép thì ghi nhận nơi kết thúc đường mở và thoát}
                begin
                  finish := j;
                  Exit;
                end;
              Push(matchY[j]);     {Nếu j đã ghép thì đẩy tiếp matchY[j] vào hàng đợi}
            end;
          if d[j] > w then    {Cập nhật lại khoảng cách d[j] nếu thấy cạnh (X[i], Y[j]) ngắn hơn khoảng cách này}
            begin
```

```
              d[j] := w;
              arg[j] := i;
            end;
        end;
  until first > last;
end;
```

{Xoay các trọng số cạnh}

```
procedure SubX_AddY;
var
  Delta: Integer;
  x, y: Integer;
begin
  {Trước hết tính Δ = giá trị nhỏ nhất trọng số các d[y], với y∈Y chưa thăm (y không thuộc cây pha)}
  Delta := maxC;
  for y := 1 to k do
    if (Trace[y] = 0) and (d[y] < Delta) then Delta := d[y];
  {Trừ trọng số những cạnh liên thuộc với start∈X đi Δ}
  Fx[start] := Fx[start] + Delta;
  for y := 1 to k do          {Xét các đỉnh y∈Y}
    if Trace[y] <> 0 then     {Nếu y thuộc cây pha}
      begin
        x := matchY[y];       {Thì x = matchY[y] cũng phải thuộc cây pha}
        Fy[y] := Fy[y] - Delta;      {Cộng trọng số những cạnh liên thuộc với y lên Δ}
        Fx[x] := Fx[x] + Delta;      {Trừ trọng số những cạnh liên thuộc với x đi Δ}
      end
    else
      d[y] := d[y] - Delta;  {Nếu y ∉ cây pha thì sau bước xoay, khoảng cách từ y đến cây pha sẽ giảm Δ}
  {Chuẩn bị tiếp tụcBFS}
  for y := 1 to k do
    if (Trace[y] = 0) and (d[y] = 0) then  {Thăm luôn những đỉnh y∈Y tạo với cây pha một 0_cạnh}
      begin
        Trace[y] := arg[y];          {Lưu vết đường đi}
        if matchY[y] = 0 then        {Nếu y chưa ghép thì ghi nhận đỉnh kết thúc đường mở và thoát ngay}
          begin
            finish := y;
            Exit;
          end;
        Push(matchY[y]);             {Nếu y đã ghép thì đẩy luôn matchY[y] vào hàng đợi để chờ loang tiếp}
      end;
end;
```

```
procedure Enlarge;     {Nới rộng bộ ghép bằng đường mở kết thúc ở finish}
var
  x, next: Integer;
begin
  repeat
    x := Trace[finish];
    next := matchX[x];
    matchX[x] := finish;
    matchY[finish] := x;
    finish := Next;
  until finish = 0;
end;
```




```
procedure Solve;
var
  x, y: Integer;
begin
  for x := 1 to k do     {Với mỗi X_đỉnh: }
    begin
      start := x;        {Đặt nơi khởi đầu đường mở}
      InitBFS;           {Khởi tạo cây pha}
      repeat
```

```
        FindAugmentingPath;                {Tìm đường mở}
        if finish = 0 then SubX_AddY;      {Nếu không thấy thì xoay các trọng số cạnh ...}
      until finish <> 0;       {Cho tới khi tìm ra đường mở}
      Enlarge;    {Nới rộng bộ ghép bởi đường mở tìm được}
    end;
end;

procedure Result;
var
  x, y, Count, W: Integer;
begin
  WriteLn('Optimal assignment:');
  W := 0; Count := 0;
  for x := 1 to m do     {Với mỗi X_đỉnh, xét cặp ghép tương ứng}
    begin
      y := matchX[x];
      if c[x, y] < maxC then    {Chỉ quan tâm đến những cặp ghép có trọng số < maxC}
        begin
          Inc(Count);
          WriteLn(Count:5, ') X[', x, '] - Y[', y, ']  ', c[x, y]);
          W := W + c[x, y];
        end;
    end;
  WriteLn('Cost: ', W);
end;

begin
  Assign(Input, 'ASSIGN.INP'); Reset(Input);
  Assign(Output, 'ASSIGN.OUT'); Rewrite(Output);
  Enter;
  Init;
  Solve;
  Result;
  Close(Input);
  Close(Output);
end.
```

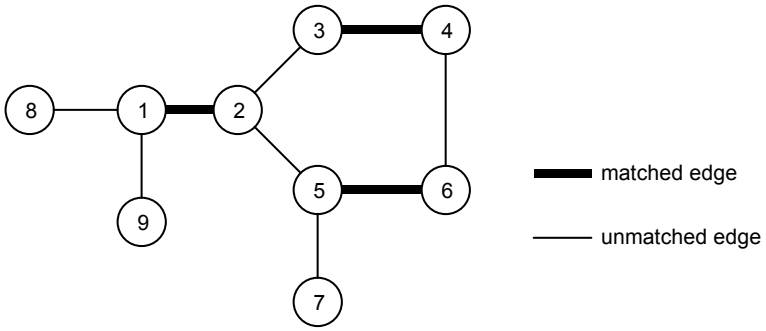
§13. BÀI TOÁN TÌM BỘ GHÉP CỰC ĐẠI TRÊN ĐỒ THỊ

I. CÁC KHÁI NIỆM

Xét đồ thị G = (V, E), một bộ ghép trên đồ thị G là một tập các cạnh đôi một không có đỉnh chung.
Bài toán tìm bộ ghép cực đại trên đồ thị tổng quát phát biểu như sau:
Cho một đồ thị G, phải tìm một bộ ghép cực đại trên G (bộ ghép có nhiều cạnh nhất).
Với một bộ ghép M của đồ thị G, ta gọi:

- Những cạnh thuộc M được gọi là cạnh đã ghép hay **cạnh đậm**
- Những cạnh không thuộc M được gọi là cạnh chưa ghép hay **cạnh nhạt**
- Những đỉnh đầu mút của các cạnh đậm được gọi là **đỉnh đã ghép**, những đỉnh còn lại gọi là **đỉnh chưa ghép**
- Một đường đi cơ bản (đường đi không có đỉnh lặp lại) được gọi là **đường pha** nếu nó bắt đầu bằng một cạnh nhạt và tiếp theo là các cạnh đậm, nhạt nằm nối tiếp xen kẽ nhau.
- Một chu trình cơ bản (chu trình không có đỉnh trong lặp lại) được gọi là một **Blossom** nếu nó đi qua ít nhất 3 đỉnh, bắt đầu và kết thúc bằng cạnh nhạt và dọc trên chu trình, các cạnh đậm, nhạt nằm nối tiếp xen kẽ nhau. Đỉnh xuất phát của chu trình (cũng là đỉnh kết thúc) được gọi là **đỉnh cơ sở** (base) của Blossom.
- **Đường mở** là một đường pha bắt đầu ở một đỉnh chưa ghép và kết thúc ở một đỉnh chưa ghép.

Ví dụ: Với đồ thị G và bộ ghép M dưới đây:



Hình 24: Đồ thị G và một bộ ghép M

- Đường (8, 1, 2, 5, 6, 4) là một đường pha
- Chu trình (2, 3, 4, 6, 5, 2) là một Blossom
- Đường (8, 1, 2, 3, 4, 6, 5, 7) là một đường mở
- Đường (8, 1, 2, 3, 4, 6, 5, 2, 1, 9) tuy có các cạnh đậm/nhạt xen kẽ nhưng không phải đường pha (và tất nhiên không phải đường mở) vì đây không phải là đường đi cơ bản.

Ta dễ dàng suy ra được các tính chất sau

- Đường mở cũng như Blossom đều là đường đi độ dài lẻ với số cạnh nhạt nhiều hơn số cạnh đậm đúng 1 cạnh.
- Trong mỗi Blossom, những đỉnh không phải đỉnh cơ sở đều là đỉnh đã ghép và đỉnh ghép với đỉnh đó cũng phải thuộc Blossom.
- Vì Blossom là một chu trình nên trong mỗi Blossom, những đỉnh không phải đỉnh cơ sở đều tồn tại hai đường pha từ đỉnh cơ sở đi đến nó, một đường kết thúc bằng cạnh đậm và một đường kết thúc bằng cạnh nhạt, hai đường pha này được hình thành bằng cách đi dọc theo chu trình theo hai hướng ngược nhau. Như ví dụ trên, đỉnh 4 có hai đường pha đi đỉnh cơ sở 2 đi tới: (2, 3, 4) là đường pha kết thúc bằng cạnh đậm và (2, 5, 6, 4) là đường pha kết thúc bằng cạnh nhạt

II. THUẬT TOÁN EDMONDS (1965)

Cơ sở của thuật toán là định lý (C.Berge): Một bộ ghép M của đồ thị G là cực đại khi và chỉ khi không tồn tại đường mở đối với M.

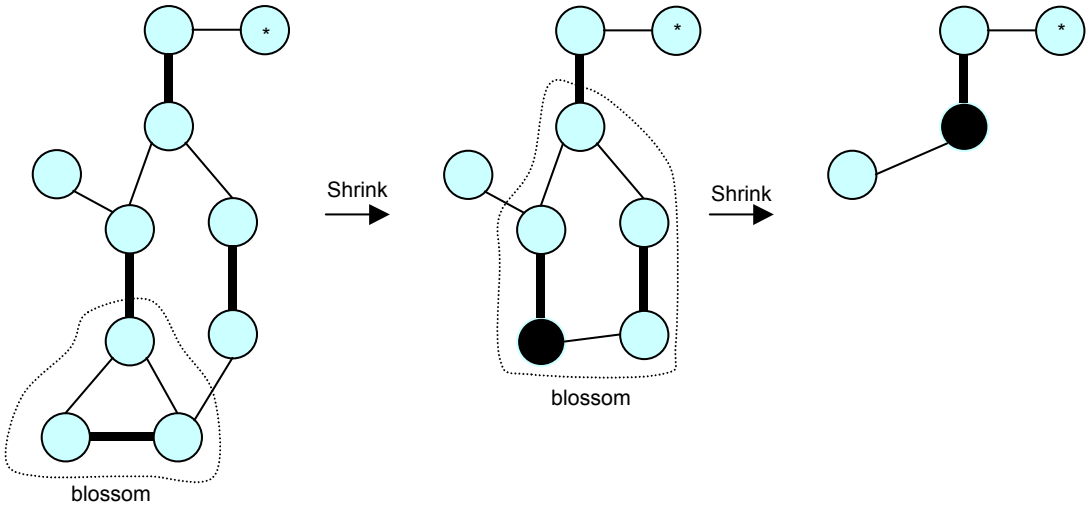
Thuật toán Edmonds:

```
M := ∅;
for (∀ đỉnh u chưa ghép) do
  if <Tìm đường mở xuất phát từ u> then
    <
     Dọc trên đường mở:
       Loại bỏ những cạnh đậm khỏi M;
       Thêm vào M những cạnh nhạt;
    >
Result: M là bộ ghép cực đại trên G
```

Điều khó nhất trong thuật toán Edmonds là phải xây dựng thuật toán tìm đường mở xuất phát từ một đỉnh chưa ghép. Thuật toán đó được xây dựng bằng cách kết hợp một thuật toán tìm kiếm trên đồ thị với phép chập Blossom.

Xét những đường pha xuất phát từ một đỉnh x chưa ghép. Những đỉnh có thể đến được từ x bằng một đường pha kết thúc là cạnh nhạt được gán nhãn "nhạt", những đỉnh có thể đến được từ x bằng một đường pha kết thúc là cạnh đậm được gán nhãn "đậm".

Với một Blossom, ta định nghĩa phép chập (shrink) là phép thay thế các đỉnh trong Blossom bằng một đỉnh duy nhất. Những cạnh nối giữa một đỉnh thuộc Blossom tới một đỉnh v nào đó không thuộc Blossom được thay thế bằng cạnh nối giữa đỉnh chập này với v và giữ nguyên tính đậm/nhạt. Dễ thấy rằng sau mỗi phép chập, các cạnh đậm vẫn được đảm bảo là bộ ghép trên đồ thị mới:



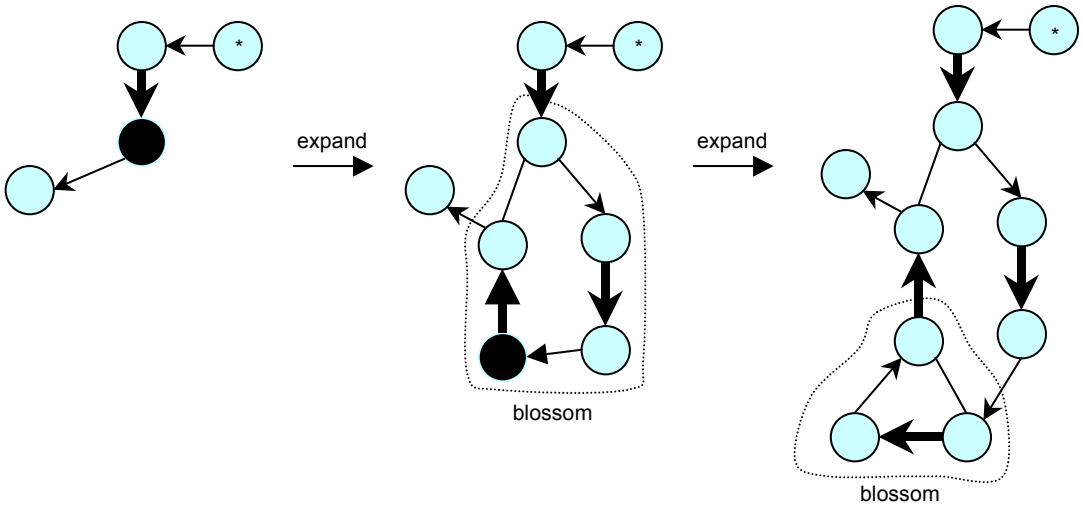
Hình 25: Phép chập Blossom

Thuật toán tìm đường mở có thể phát biểu như sau.

- Trước hết đỉnh xuất phát x được gán nhãn đậm.
- Tiếp theo là thuật toán tìm kiếm trên đồ thị bắt đầu từ x, theo nguyên tắc: từ đỉnh đậm chỉ được phép đi tiếp theo cạnh nhạt và từ đỉnh nhạt chỉ được đi tiếp theo cạnh đậm. Mỗi khi thăm tới một đỉnh, ta gán nhãn đậm/nhạt cho đỉnh đó và tiếp tục thao tác tìm kiếm trên đồ thị như bình thường. Cũng trong quá trình tìm kiếm, mỗi khi phát hiện thấy một cạnh nhạt nối hai đỉnh đậm, ta dừng lại ngay vì nếu gán nhãn tiếp sẽ gặp tình trạng một đỉnh có cả hai nhãn đậm/nhạt, trong trường hợp này, Blossom được phát hiện (xem tính chất của Blossom) và bị chập thành một

đỉnh, thuật toán được bắt đầu lại với đồ thị mới cho tới khi trả lời được câu hỏi: "có tồn tại đường mở xuất phát từ x hay không?"

- Nếu đường mở tìm được không đi qua đỉnh chập nào thì ta chỉ việc tăng cặp dọc theo đường mở. Nếu đường mở có đi qua một đỉnh chập thì ta lại nở đỉnh chập đó ra thành Blossom để thay đỉnh chập này trên đường mở bằng một đoạn đường xuyên qua Blossom:



Hình 26: Nở Blossom để dò đường xuyên qua Blossom

Lưu ý rằng không phải Blossom nào cũng bị chập, chỉ những Blossom ảnh hưởng tới quá trình tìm đường mở mới phải chập để đảm bảo rằng đường mở tìm được là đường đi cơ bản. Tuy nhiên việc cài đặt trực tiếp các phép chập Blossom và nở đỉnh khá rắc rối, đòi hỏi một chương trình với độ phức tạp $O(n^4)$.

Dưới đây ta sẽ trình bày một phương pháp cài đặt hiệu quả hơn với độ phức tạp $O(n^3)$, phương pháp này cài đặt không phức tạp, nhưng yêu cầu phải hiểu rất rõ bản chất thuật toán.

III. PHƯƠNG PHÁP LAWLER (1973)

Trong phương pháp Edmonds, sau khi chập mỗi Blossom thành một đỉnh thì đỉnh đó hoàn toàn lại có thể nằm trên một Blossom mới và bị chập tiếp. Phương pháp Lawler chỉ quan tâm đến đỉnh chập cuối cùng, đại diện cho Blossom ngoài nhất (Outermost Blossom), đỉnh chập cuối cùng này được định danh (đánh số) bằng đỉnh cơ sở của Blossom ngoài nhất.

Cũng chính vì thao tác chập/nở nói trên mà ta cần mở rộng khái niệm Blossom, có thể **coi một Blossom là một tập đỉnh nở ra từ một đỉnh chập** chứ không đơn thuần chỉ là một chu trình pha cơ bản nữa.

Xét một Blossom B có đỉnh cơ sở là đỉnh r. Với $\forall v \in B$, $v \neq r$, ta lưu lại hai đường pha từ r tới v, một đường kết thúc bằng cạnh đậm và một đường kết thúc bằng cạnh nhạt, như vậy có hai loại vết gãn cho mỗi đỉnh v:

- S[v] là đỉnh liền trước v trên đường pha kết thúc bằng cạnh đậm, nếu không tồn tại đường pha loại này thì S[v] = 0.
- T[v] là đỉnh liền trước v trên đường pha kết thúc bằng cạnh nhạt, nếu không tồn tại đường pha loại này thì T[v] = 0.

Bên cạnh hai nhãn S và T, mỗi đỉnh v còn có thêm

- Nhãn b[v] là đỉnh cơ sở của Blossom chứa v. Hai đỉnh u và v thuộc cùng một Blossom $\Leftrightarrow$ b[u] = b[v].
- Nhãn match[v] là đỉnh ghép với đỉnh v. Nếu v chưa ghép thì match[v] = 0.

Khi đó thuật toán tìm đường mở bắt đầu từ đỉnh x chưa ghép có thể phát biểu như sau:

Bước 1: (Init)

- Hàng đợi Queue dùng để chứa những đỉnh đậm chờ duyệt, ban đầu chỉ gồm một đỉnh đậm x.
- Với mọi đỉnh u, khởi gán b[u] = u và match[u] = 0 với ∀u.
- Gán S[x] ≠ 0; Với ∀u≠x, gán S[u] = 0;Với ∀v: gán T[v] = 0

Bước 2: (BFS)

Lặp lại các bước sau cho tới khi hàng đợi rỗng:

Với mỗi đỉnh đậm u lấy ra từ Queue, xét những cạnh nhạt (u, v):

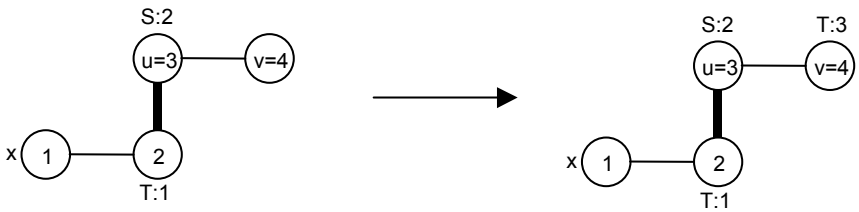
- Nếu v chưa thăm:
  - ♦ Nếu v là đỉnh chưa ghép ⇒ Tìm thấy đường mở kết thúc ở v, dừng
  - ♦ Nếu v là đỉnh đã ghép ⇒ thăm v ⇒ thăm luôn match[v] và đẩy match[v] vào Queue.

  Sau mỗi lần thăm, chú ý việc lưu vết (hai nhãn S và T)
- Nếu v đã thăm
  - ♦ Nếu v là đỉnh nhạt hoặc b[v] = b[u] ⇒ bỏ qua
  - ♦ Nếu v là đỉnh đậm và b[v] ≠ b[u] ta phát hiện được blossom mới chứa u và v, khi đó:
    - **Phát hiện đỉnh cơ sở:** Truy vết đường đi ngược từ hai đỉnh đậm u và v theo hai đường pha về nút gốc, chọn lấy đỉnh a là đỉnh đậm chung gặp đầu tiên trong quá trình truy vết ngược. Khi đó Blossom mới phát hiện sẽ có đỉnh cơ sở là a.
    - **Gán lại vết:** Gọi ($a = i_1, i_2, ..., i_p = u$) và ($a = j_1, j_2, ..., j_q = v$) lần lượt là hai đường pha dẫn từ a tới u và v. Khi đó ($a = i_1, i_2, ..., i_p = u, j_q = v, j_{q-1}, ..., j_1 = a$) là một chu trình pha đi từ a tới u và v rồi quay trở về a. Bằng cách đi dọc theo chu trình này theo hai hướng ngược nhau, ta có thể gán lại tất cả các nhãn S và T của những đỉnh trên chu trình. Lưu ý rằng **không được gán lại nhãn** S và T cho những đỉnh k mà b[k] = a, và với những đỉnh k có b[k] ≠ a thì **bắt buộc phải gán lại nhãn** S và T theo chu trình này bất kể S[k] và T[k] trước đó đã có hay chưa.
    - **Chập Blossom:** Xét những đỉnh v mà $b[v] \in \{b[i_1], b[i_2], ..., b[i_p], b[j_1], b[j_2], ..., b[j_q]\}$, gán lại b[v] = a. Nếu v là đỉnh đậm (có nhãn S[v] ≠ 0) mà chưa được duyệt tới (chưa bao giờ được đẩy vào Queue) thì đẩy v vào Queue chờ duyệt tiếp tại những bước sau.

Nếu quá trình này chỉ thoát khi hàng đợi rỗng thì tức là không tồn tại đường mở bắt đầu từ x.

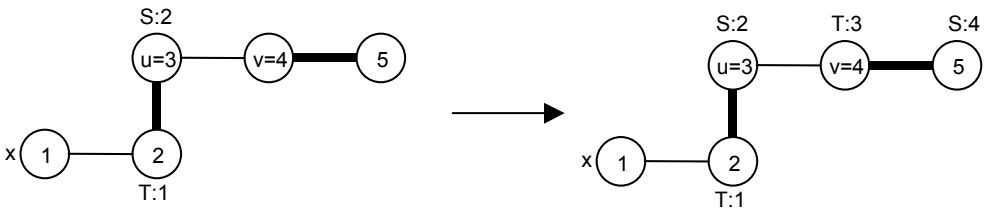
Sau đây là một số ví dụ về các trường hợp từ đỉnh đậm u xét cạnh nhạt (u, v):

Trường hợp 1: v chưa thăm và chưa ghép:



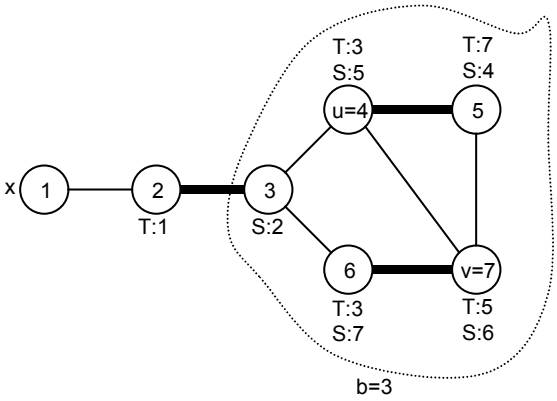
Tìm thấy đường mở

Trường hợp 2: v chưa thăm và đã ghép



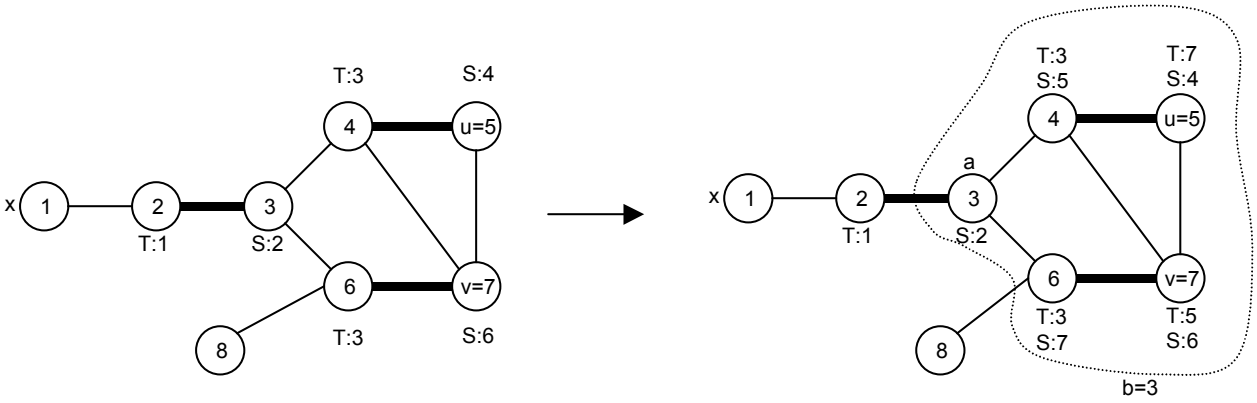
Thăm cả v lẫn match[v], gán nhãn T[v] và S[match[v]]

Trường hợp 3: v đã thăm, là đỉnh đậm thuộc cùng blossom với u



Không xét, bỏ qua

Trường hợp 4: v đã thăm, là đỉnh đậm và b[u] ≠ b[v]



Tìm đỉnh cơ sở a = 3, gán lại nhãn S và T dọc chu trình pha, chập Blossom.
Đẩy hai đỉnh đậm mới 4, 6 vào hàng đợi, Tại những bước sau,
khi duyệt tới đỉnh 6, sẽ tìm thấy đường mở kết thúc ở 8,
truy vết theo nhãn S và T tìm được đường (1, 2, 3, 4, 5, 7, 6, 8)

Tư tưởng chính của phương pháp Lawler là dùng các nhãn b[v] thay cho thao tác chập trực tiếp Blossom, dùng các nhãn S và T để truy vết tìm đường mở, tránh thao tác nở Blossom. Phương pháp này dựa trên một nhận xét: Mỗi khi tìm ra đường mở, nếu đường mở đó xuyên qua một Blossom ngoài nhất thì chắc chắn nó phải đi vào Blossom này từ nút cơ sở và thoát ra khỏi Blossom bằng một cạnh nhạt.

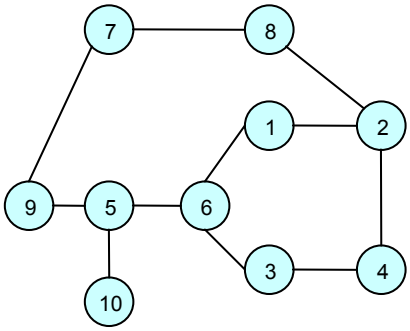
IV. CÀI ĐẶT

Ta sẽ cài đặt phương pháp Lawler với khuôn dạng Input/Output như sau:

Input: file văn bản GMATCH.INP

- Dòng 1: Chứa hai số n, m lần lượt là số cạnh và số đỉnh của đồ thị cách nhau ít nhất một dấu cách (n ≤ 100)
- m dòng tiếp theo, mỗi dòng chứa hai số u, v tượng trưng cho một cạnh (u, v) của đồ thị

Output: file văn bản GMATCH.OUT chứa bộ ghép cực đại tìm được



GMATCH.INP	GMATCH.OUT
10 11 1 2 1 6 2 4 2 8 3 4 3 6 5 6 5 9 5 10 7 8 7 9	1 6 2 8 3 4 5 10 7 9 Number of matched edges: 5

Chương trình này sửa đổi một chút mô hình cài đặt trên dựa vào nhận xét:

- v là một đỉnh đậm ⇔ v = x hoặc match[v] là một đỉnh nhạt
- Nếu v là đỉnh đậm thì S[v] = match[v]

Vậy thì ta không cần phải sử dụng riêng một mảng nhãn S[v], tại mỗi bước sửa vết, ta chỉ cần sửa nhãn vết T[v] mà thôi. Để kiểm tra một đỉnh v ≠ x có phải đỉnh đậm hay không, ta có thể kiểm tra bằng điều kiện: match[v] có là đỉnh nhạt hay không, hay T[match[v]] có khác 0 hay không.

Chương trình sử dụng các biến với vai trò như sau:

- match[v] là đỉnh ghép với đỉnh v
- b[v] là đỉnh cơ sở của Blossom chứa v
- T[v] là đỉnh liền trước v trên đường pha từ đỉnh xuất phát tới v kết thúc bằng cạnh nhạt, T[v] = 0 nếu quá trình BFS chưa xét tới đỉnh nhạt v.
- InQueue[v] là biến Boolean, InQueue[v] = True ⇔ v là đỉnh đậm đã được đẩy vào Queue để chờ duyệt.
- start và finish: Nơi bắt đầu và kết thúc đường mở.

PROG13_1.PAS * Phương pháp Lawler áp dụng cho thuật toán Edmonds

```
program MatchingInGeneralGraph;
const
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  match, Queue, b, T: array[1..max] of Integer;
  InQueue: array[1..max] of Boolean;
  n, first, last, start, finish: Integer;

procedure Enter;     {Nhập dữ liệu, từ thiết bị nhập chuẩn (Input)}
var
  i, m, u, v: Integer;
begin
  FillChar(a, SizeOf(a), 0);
  ReadLn(n, m);
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
end;

procedure Init;      {Khởi tạo bộ ghép rỗng}
```

```
begin
  FillChar(match, SizeOf(match), 0);
end;

procedure InitBFS; {Thủ tục này được gọi để khởi tạo trước khi tìm đường mở xuất phát từ start}
var
  i: Integer;
begin
  {Hàng đợi chỉ gồm một đỉnh đậm start}
  first := 1; last := 1;
  Queue[1] := start;
  FillChar(InQueue, SizeOf(InQueue), False);
  InQueue[start] := True;
  {Các nhãn T được khởi gán = 0}
  FillChar(T, SizeOF(T), 0);
  {Nút cơ sở của outermost blossom chứa i chính là i}
  for i := 1 to n do b[i] := i;
  finish := 0;   {finish = 0 nghĩa là chưa tìm thấy đường mở}
end;

procedure Push(v: Integer);      {Đẩy một đỉnh đậm v vào hàng đợi}
begin
  Inc(last);
  Queue[last] := v;
  InQueue[v] := True;
end;

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

{Khó nhất của phương pháp Lawler là thủ tục này: Thủ tục xử lý khi gặp cạnh nhạt nốt hai đỉnh đậm p, q}
procedure BlossomShrink(p, q: Integer);
var
  i, NewBase: Integer;
  Mark: array[1..max] of Boolean;

  {Thủ tục tìm nút cơ sở bằng cách truy vết ngược theo đường pha từ p và q}
  function FindCommonAncestor(p, q: Integer): Integer;
  var
    InPath: array[1..max] of Boolean;
  begin
    FillChar(InPath, SizeOf(Inpath), False);
    repeat           {Truy vết từ p}
      p := b[p];                  {Nhảy tới nút cơ sở của Blossom chứa p, phép nhảy này để tăng tốc độ truy vết}
      Inpath[p] := True;          {Đánh dấu nút đó}
      if p = start then Break;    {Nếu đã truy về đến nơi xuất phát thì dừng}
      p := T[match[p]];       {Nếu chưa về đến start thì truy lùi tiếp hai bước, theo cạnh đậm rồi theo cạnh nhạt}
    until False;
    repeat           {Truy vết từ q, tương tự như đối với p}
      q := b[q];
      if InPath[q] then Break; {Tuy nhiên nếu chạm vào đường pha của p thì dừng ngay}
      q := T[match[q]];
    until False;
    FindCommonAncestor := q;    {Ghi nhận đỉnh cơ sở mới}
  end;

  procedure ResetTrace(x: Integer); {Gán lại nhãn vết dọc trên đường pha từ start tới x}
  var
    u, v: Integer;
  begin
    v := x;
```

```
    while b[v] <> NewBase do    {Truy vết đường pha từ start tới đỉnh đậm x}
      begin
        u := match[v];
        Mark[b[v]] := True;     {Đánh dấu nhãn blossom của các đỉnh trên đường đi}
        Mark[b[u]] := True;
        v := T[u];
        if b[v] <> NewBase then T[v] := u; {Chỉ đặt lại vết T[v] nếu b[v] không phải nút cơ sở mới}
      end;
  end;

begin      {BlossomShrink}
  FillChar(Mark, SizeOf(Mark), False);   {Tất cả các nhãn b[v] đều chưa bị đánh dấu}
  NewBase := FindCommonAncestor(p, q);   {xác định nút cơ sở}
  {Gán lại nhãn}
  ResetTrace(p); ResetTrace(q);
  if b[p] <> NewBase then T[p] := q;
  if b[q] <> NewBase then T[q] := p;
  {Chập blossom ⇔ gán lại các nhãn b[i] nếu blossom b[i] bị đánh dấu}
  for i := 1 to n do
    if Mark[b[i]] then b[i] := NewBase;
  {Xét những đỉnh đậm i chưa được đưa vào Queue nằm trong Blossom mới, đẩy i và Queue để chờ duyệt tiếp tại các bước sau}
  for i := 1 to n do
    if not InQueue[i] and (b[i] = NewBase) then
      Push(i);
end;

{Thủ tục tìm đường mở}
procedure FindAugmentingPath;
var
  u, v: Integer;
begin
  InitBFS;  {Khởi tạo}
  repeat    {BFS}
    u := Pop;
    {Xét những đỉnh v chưa duyệt, kề với u, không nằm cùng Blossom với u, dĩ nhiên T[v] = 0 thì (u, v) là cạnh nhạt rồi}
    for v := 1 to n do
      if (T[v] = 0) and (a[u, v]) and (b[u] <> b[v]) then
        begin
          if match[v] = 0 then    {Nếu v chưa ghép thì ghi nhận đỉnh kết thúc đường mở và thoát ngay}
            begin
              T[v] := u;
              finish := v;
              Exit;
            end;
          {Nếu v là đỉnh đậm thì gán lại vết, chập Blossom ...}
          if (v = start) or (T[match[v]] <> 0) then
            BlossomShrink(u, v)
          else      {Nếu không thì ghi vết đường đi, thăm v, thăm luôn cả match[v] và đẩy tiếp match[v] vào Queue}
            begin
              T[v] := u;
              Push(match[v]);
            end;
        end;
  until first > last;
end;

procedure Enlarge;     {Nới rộng bộ ghép bởi đường mở bắt đầu từ start, kết thúc ở finish}
var
  v, next: Integer;
begin
  repeat
    v := T[finish];
    next := match[v];
    match[v] := finish;
```

```
    match[finish] := v;
    finish := next;
  until finish = 0;
end;

procedure Solve;    {Thuật toán Edmonds}
var
  u: Integer;
begin
  for u := 1 to n do
    if match[u] = 0 then
      begin
        start := u;           {Với mỗi đỉnh chưa ghép start}
        FindAugmentingPath;   {Tìm đường mở bắt đầu từ start}
        if finish <> 0 then Enlarge;   {Nếu thấy thì nới rộng bộ ghép theo đường mở này}
      end;
end;

procedure Result;        {In bộ ghép tìm được}
var
  u, count: Integer;
begin
  count := 0;
  for u := 1 to n do
    if match[u] > u then   {Vừa tránh sự trùng lặp (u, v) và (v, u), vừa loại những đỉnh không ghép được (match=0)}
      begin
        Inc(count);
        WriteLn(u, ' ', match[u]);
      end;
  WriteLn('Number of matched edges: ', count);
end;

begin
  Assign(Input, 'GMATCH.INP'); Reset(Input);
  Assign(Output, 'GMATCH.OUT'); Rewrite(Output);
  Enter;
  Init;
  Solve;
  Result;
  Close(Input);
  Close(Output);
end.
```

V. ĐỘ PHỨC TẠP TÍNH TOÁN

- Thủ tục BlossomShrink có độ phức tạp O(n).
- Thủ tục FindAugmentingPath cần không quá n lần gọi thủ tục BlossomShrink, cộng thêm chi phí của thuật toán tìm kiếm theo chiều rộng, có độ phức tạp $O(n^2)$
- Phương pháp Lawler cần không quá n lần gọi thủ tục FindAugmentingPath nên có độ phức tạp tính toán là $O(n^3)$