

Chương 1. Tính liên thông của đồ thị

1.1. Định nghĩa

1.1.1. Tính liên thông trên đồ thị vô hướng

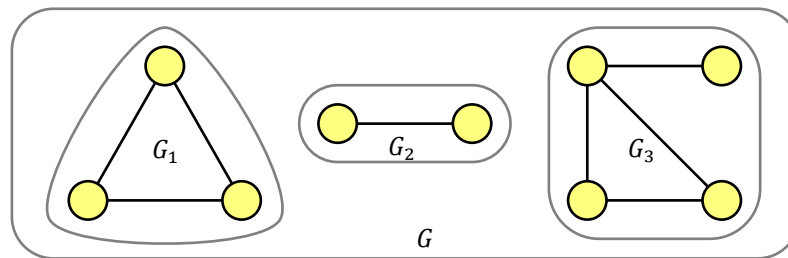
Đồ thị vô hướng $G = (V, E)$ được gọi là *liên thông (connected)* nếu giữa mọi cặp đỉnh của G luôn tồn tại đường đi. Đồ thị chỉ gồm một đỉnh duy nhất cũng được coi là đồ thị liên thông.

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

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

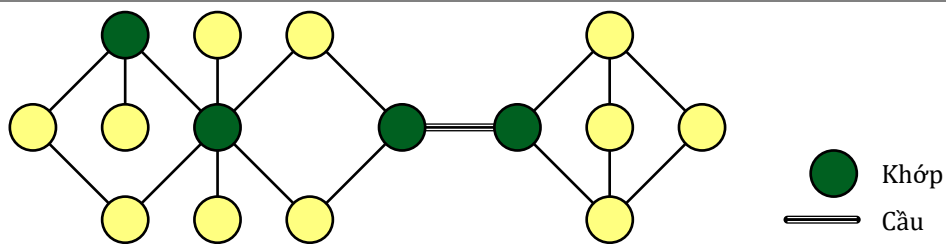
(Ta cũng đồng nhất khái niệm thành phần liên thông U với thành phần liên thông $G_U = (U, E_U)$).

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



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

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

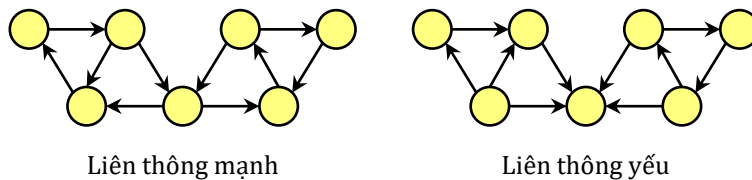


Hình 1-2. Khớp và cầu

1.1.2. Tính liên thông trên đồ thị có hướng

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

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



Hình 1-3. Liên thông mạnh và liên thông yếu

1.2. Bài toán xác định các thành phần liên thông

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

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

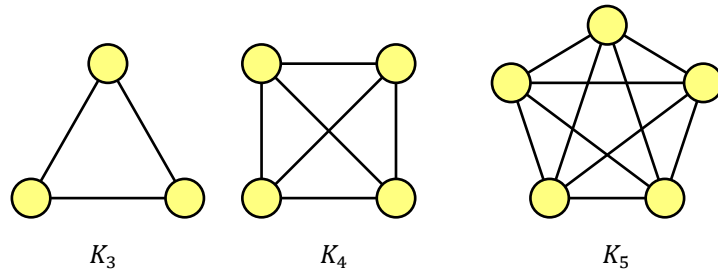
```
1 void Scan(u ∈ V)
2 {
3     «Dùng DFS hoặc BFS liệt kê và đánh dấu "đã thăm"
4     những đỉnh có thể đến được từ u»;
5 }
6
7 for (∀u ∈ V)
8     «Đánh dấu v chưa thăm»;
9 for (∀u ∈ V)
10    if («u chưa thăm»)
11    {
12        Output ← «Các đỉnh của một thành phần liên thông:»;
13        Scan(u);
14    }
```

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

1.3. Bao đóng của đồ thị vô hướng

1.3.1. Định nghĩa

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



Hình 1-4. Đồ thị đầy đủ

1.3.2. Bao đóng đồ thị

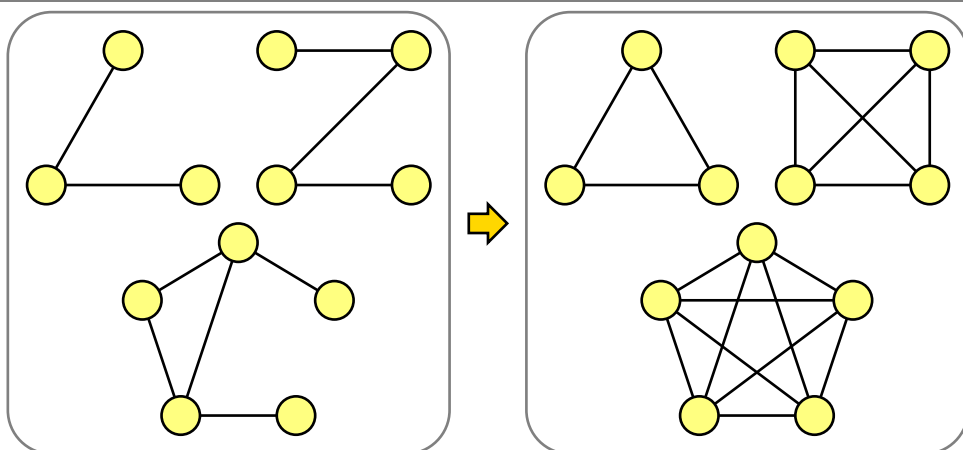
Với đồ thị $G = (V, E)$, người ta xây dựng đồ thị $\overline{G} = (V, \overline{E})$ cũng gồm những đỉnh của G còn các cạnh xây dựng như sau:

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

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

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

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



Hình 1-5. Đồ thị vô hướng và bao đóng của nó

Bởi việc kiểm tra một đơn đồ thị có phải đồ thị đầy đủ hay không có thể thực hiện khá dễ dàng 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.

1.3.3. Thuật toán Warshall

Thuật toán Warshall (Warshall, 1962) – gọi theo tên tác giả Stephen Warshall, đôi khi còn được gọi là thuật toán Roy-Warshall vì Bernard Roy cũng đã mô tả thuật toán này vào năm 1959. Thuật toán đó có thể mô tả rất gọn:

Giả sử đơn đồ thị vô hướng $G = (V, E)$ có n đỉnh đánh số từ 0 tới $n - 1$, thuật toán Warshall xét tất cả các đỉnh $k \in V$, với mỗi đỉnh k được xét, thuật toán lại xét tiếp tất cả các cặp đỉnh (i, j) : nếu đồ thị có cạnh (i, k) và cạnh (k, j) thì ta tự nối thêm cạnh (i, j) nếu nó chưa có. Tư tưởng này dựa trên một quan sát đơn giản như sau: Nếu từ i có đường đi tới k và từ k lại có đường đi tới j thì chắc chắn từ i sẽ có đường đi tới j .

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

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

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

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

✧ Tính đúng đắn của thuật toán

Tính đúng đắn của thuật toán được chứng minh qua bất biến lặp:

- ✧ Trước một lượt lặp với một giá trị k , a_{ij} cho biết sự tồn tại đường đi $i \rightsquigarrow j$ mà chỉ qua các đỉnh trong thuộc tập $\{0, 1, \dots, k - 1\}$
- ✧ Sau một lượt lặp với một giá trị k , a_{ij} cho biết sự tồn tại một đường đi $i \rightsquigarrow j$ mà chỉ qua các đỉnh trong thuộc tập $\{0, 1, \dots, k\}$

Nhắc lại rằng đỉnh trong của một đường đi là những đỉnh thuộc đường đi ngoại trừ hai đỉnh đầu cuối, cũng có thể gọi chúng là những đỉnh trung chuyển hay đỉnh trung gian trên đường đi.

Tính bất biến đúng trước lượt lặp đầu tiên

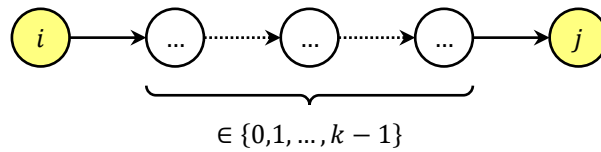
Trước khi vào vòng lặp thì a_{ij} = sự tồn tại của cạnh (i, j) , ngoài ra thì $a_{ii} = \text{true}$ với $\forall i$. Tức là a_{ij} = sự tồn tại đường đi $i \rightsquigarrow j$ mà không cần đi qua bất cứ một đỉnh trong nào cả (tập đỉnh trong bằng \emptyset).

Tính bất biến được duy trì trong quá trình lặp:

Giả sử trước một lượt lặp với giá trị k , a_{ij} = sự tồn tại đường đi $i \rightsquigarrow j$ mà chỉ qua các đỉnh trong thuộc tập $\{0, 1, \dots, k - 1\}$.

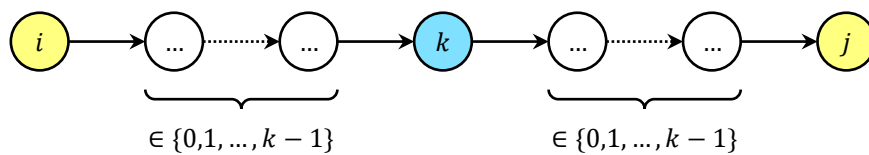
Đường đi $i \rightsquigarrow j$ mà chỉ qua các đỉnh trong $\in \{0, 1, \dots, k\}$ nếu tồn tại sẽ thuộc một trong hai loại:

Loại 1: Đường đi chỉ qua các đỉnh trong $\in \{0, 1, \dots, k - 1\}$:



Trong trường hợp này a_{ij} đã bằng true trước lượt lặp với giá trị k và nó sẽ duy trì bằng true sau lượt lặp với giá trị k .

Loại 2: Đường đi có qua đỉnh trong bằng k , trong số những đường đi như vậy, chắc chắn sẽ có một đường đi đơn có đỉnh trong bằng k



Xét đoạn đường từ i tới k và đoạn đường từ k tới j , vì chúng là đường đi đơn nên chỉ đi qua các đỉnh trong $\in \{0, 1, \dots, k - 1\}$, các giá trị a_{ik} và a_{kj} không hề bị thay đổi trong lượt lặp với giá trị k , vì vậy nếu cả a_{ik} và a_{kj} đều bằng true, ta có thể đặt lại $a_{ij} = \text{true}$.

Phân tích hai trường hợp trên cho thấy phép cập nhật:

$$a[i][j] = a[i][k] \ \&\& \ a[k][j];$$

sẽ duy trì tính bất biến sau mỗi lượt lặp với giá trị k .

Tính bất biến suy ra ĐPCM

Thuật toán sẽ kết thúc khi duyệt qua mọi đỉnh k từ 0 tới $n - 1$. Khi vòng lặp ngoài cùng kết thúc, a_{ij} chỉ ra sự tồn tại của đường đi $i \rightsquigarrow j$ mà chỉ đi qua các đỉnh trong $\in \{0, 1, \dots, n - 1\}$, tức là a_{ij} chỉ ra sự tồn tại của đường đi $i \rightsquigarrow j$.

Có ba điểm cần lưu ý:

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

* Cài đặt

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

Việc cài đặt thuật toán sẽ qua những bước sau:

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

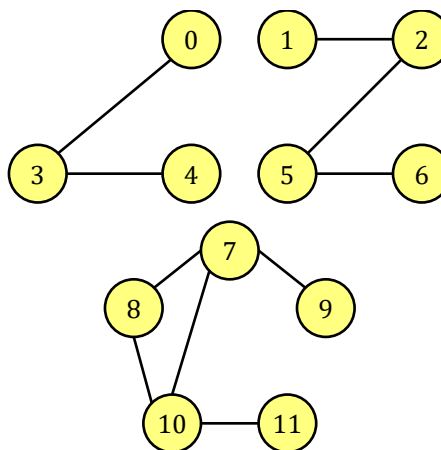
Input

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

Output

Liệt kê các thành phần liên thông của đồ thị

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



WARSHALL.cpp Thuật toán Warshall

```
1 #include <iostream>
2 using namespace std;
3 const int maxN = 200;
4 int n, m;
5 bool a[maxN][maxN]; //Ma trận kề
6 bool avail[maxN]; //Đánh dấu đỉnh chưa được liệt kê
7
```

```

8 void ReadInput() //Nhập dữ liệu
9 {
10     cin >> n >> m;
11     for (int i = 0; i < n; ++i)
12         for (int j = 0; j < n; ++j)
13             a[i][j] = i == j; //a[i][i] = true, ∀i
14     while (m-- > 0)
15     {
16         int u, v;
17         cin >> u >> v;
18         a[u][v] = a[v][u] = true; //Đồ thị vô hướng
19     }
20 }
21
22 void ComputeTransitiveClosure() //Thuật toán Warshall
23 {
24     for (int k = 0; k < n; ++k)
25         for (int i = 0; i < n; ++i)
26             for (int j = 0; j < n; ++j)
27                 a[i][j] |= a[i][k] && a[k][j];
28 }
29
30 void Print()
31 {
32     fill(avail, avail + n, true); //avail[.] = true: đỉnh chưa được liệt kê
33     for (int u = 0; u < n; ++u) //Xét mọi đỉnh
34         if (avail[u]) //Gặp đỉnh chưa được liệt kê vào thành phần liên thông nào
35         { //Liệt kê thành phần liên thông chứa u
36             cout << "Connected Component: \n";
37             for (int v = 0; v < n; ++v)
38                 if (a[u][v]) //Những đỉnh v kề với u trên bao đóng
39                 {
40                     cout << v << ", ";
41                     avail[v] = false; //Liệt kê đỉnh nào đánh dấu đỉnh đó
42                 }
43             cout << "\n";
44         }
45 }
46
47 int main()
48 {
49     ReadInput();
50     ComputeTransitiveClosure();
51     Print();
52 }

```

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

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

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

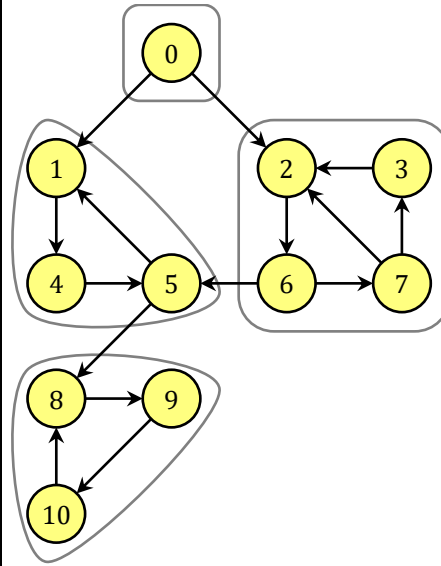
Input

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

Output

Các thành phần liên thông mạnh.

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



1.4.1. Phân tích

Xét thuật toán tìm kiếm theo chiều sâu:

```

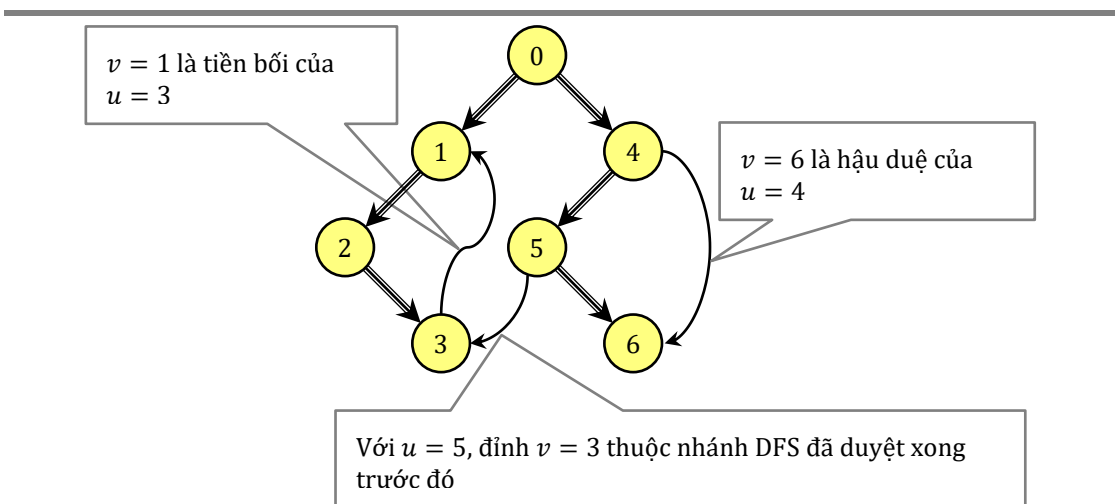
1 void DFSVisit(u ∈ V)
2 {
3     num[u] = ++Time; //Ghi nhận thời điểm thăm u, cũng là đánh dấu ≠ 0
4     for (∀v: (u, v) ∈ E)
5         if (num[v] == 0) //v chưa thăm
6             DFSVisit(v);
7 }
8
9 for (∀u ∈ V) //num[u] = thời điểm thăm u, bằng 0 nếu chưa thăm
10     num[u] = 0; //Các đỉnh đều chưa thăm
11 Time = 0;
12 for (∀u ∈ V)
13     if (num[u] == 0)
14         DFSVisit(u);

```

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

Để ý hàm $\text{DFSVisit}(u)$. Hàm này xét tất cả những đỉnh v nối từ u :

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



Hình 1-6. Ba loại cung ngoài cây DFS

Ta nhận thấy một đặc điểm của thuật toán tìm kiếm theo chiều sâu, thuật toán không chỉ duyệt qua các đỉnh, nó còn duyệt qua tất cả những cung nữa. Ngoài những cung nằm trên cây DFS, những cung còn lại có thể chia làm ba loại: cung ngược, cung xuôi, cung chéo. Ở đây ta không xét khuyên là những cung nối một đỉnh với chính nó, trong trường hợp đồ thị có khuyên, có thể coi khuyên là cung xuôi.

Chú ý rằng việc phân chia các loại cung là phụ thuộc vào cấu trúc cây DFS, cấu trúc cây DFS lại phụ thuộc vào thứ tự duyệt danh sách kề của các đỉnh. Hình 1-6 là ví dụ về các loại cung trên cây DFS khi danh sách kề của các đỉnh được sắp xếp theo thứ tự tăng dần của số hiệu đỉnh.

1.4.2. Cây DFS và các thành phần liên thông mạnh

Bổ đề 1-1

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

Chứng minh

Gọi v là một đỉnh trên đường đi $x \rightsquigarrow y$. Vì x và y thuộc cùng một thành phần liên thông mạnh nên cũng tồn tại một đường đi $y \rightsquigarrow x$. Nối tiếp hai đường đi này lại ta sẽ được một chu trình đi từ x tới y rồi quay lại x . Mọi điểm trên chu trình này phải thuộc cùng một thành phần liên thông mạnh, v nằm trên chu trình nên $v \in C$.

Bổ đề 1-2

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

Chứng minh

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

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

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

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

Bổ đề 1-3

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

Chứng minh:

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

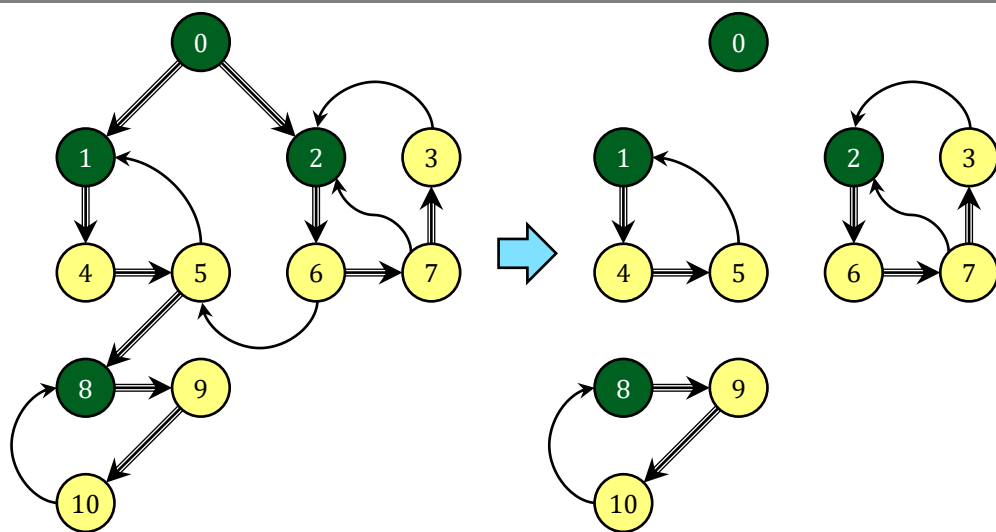
thành phần liên thông mạnh đó. Mỗi thành phần liên thông mạnh có duy nhất một chốt mà r và s đều là chốt nên $r = s$.

Bổ đề 1-2 khẳng định thành phần liên thông mạnh chứa r nằm trong nhánh DFS gốc r , theo chứng minh trên ta lại có: Mọi đỉnh trong nhánh DFS gốc r nằm trong thành phần liên thông mạnh chứa r . Kết hợp lại được: Nhánh DFS gốc r chính là thành phần liên thông mạnh chứa r .

1.4.3. Thuật toán Tarjan

✱ Ý tưởng

Thuật toán Tarjan (Tarjan, 1972) có thể phát biểu như sau: Chọn r là chốt không là tiền bối của một chốt nào khác, chọn lấy thành phần liên thông mạnh thứ nhất là nhánh DFS gốc r . Sau đó loại bỏ nhánh DFS gốc r ra khỏi cây DFS, lại tìm thấy một chốt s khác mà nhánh DFS gốc s không chứa chốt nào khác, lại chọn lấy thành phần liên thông mạnh thứ hai là nhánh DFS gốc s ... Tương tự như vậy cho thành phần liên thông mạnh thứ ba, thứ tư, v.v... Có thể hình dung thuật toán Tarjan “bẻ” cây DFS bằng cách cắt bỏ các cung nối tới chốt để được các nhánh rời rạc, mỗi nhánh là một thành phần liên thông mạnh.



Hình 1-7. Thuật toán Tarjan “bẻ” cây DFS

Mô hình cài đặt của thuật toán Tarjan:

```

1 void DFSVisit(u ∈ V)
2 {
3     num[u] = ++Time; //Ghi nhận thời điểm thăm u, cũng là đánh dấu ≠ 0
4     for (∀v: (u, v) ∈ E) //Xét mọi đỉnh v nối từ u
5         if (num[v] == 0) //v chưa thăm
6             DFSVisit(v); //Thăm v bằng DFS
7     if («u là chốt»)
8         «Liệt kê và loại bỏ các đỉnh ∈ thành phần
9         liên thông mạnh ứng với chốt u»;
10 }
11
12 for (∀u ∈ V) //num[u] = thời điểm thăm u, bằng 0 nếu chưa thăm
13     num[v] = 0; //Các đỉnh đều chưa thăm
14 Time = 0;
15 for (∀u ∈ V)
16     if (num[u] == 0) //Gặp đỉnh u chưa thăm
17         DFSVisit(u); //DFS từ u

```

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

Bổ đề 1-4

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

Chứng minh

Ta nhắc lại các tính chất của 4 loại cung:

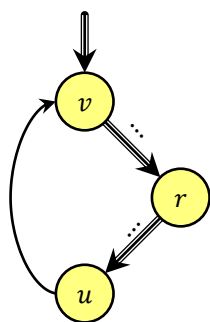
- ✳ Cung DFS và cung xuôi nối từ đỉnh thăm trước đến đỉnh thăm sau, hơn nữa chúng đều là cung nối từ tiền bối tới hậu duệ
- ✳ Cung ngược và cung chéo nối từ đỉnh thăm sau tới đỉnh thăm trước, cung ngược nối từ hậu duệ tới tiền bối còn cung chéo nối hai đỉnh không có quan hệ tiền bối–hậu duệ.

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

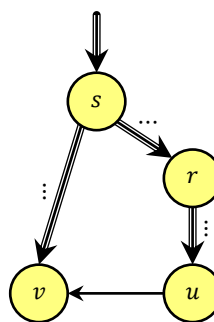
Ngược lại, nếu từ đỉnh u của nhánh DFS gốc r có cung (u, v) tới đỉnh v thăm trước r , ta chứng minh rằng r không thể là chốt.

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

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



TH (u, v) là cung ngược



TH (u, v) là cung chéo

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

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

Suy ra khi hàm $\text{DFSVisit}(u)$ được gọi, hai hàm $\text{DFSVisit}(r)$ và $\text{DFSVisit}(s)$ đều đã được gọi nhưng chưa thoát, tức là chúng nằm trên một dây chuyền đệ quy, hay r và s có quan hệ tiền bối-hậu duệ. Vì s được thăm trước r nên s sẽ là tiền bối của r , ta có chu trình $s \rightsquigarrow r \rightsquigarrow u \rightarrow v \rightsquigarrow s$ nên r và s thuộc cùng một thành phần liên thông mạnh, thành phần này đã có chốt s rồi nên r không thể là chốt nữa.

Từ Bổ đề 1-4, việc sẽ kiểm tra đỉnh r có là chốt hay không có thể thay bằng việc kiểm tra xem có tồn tại cung nối từ một đỉnh thuộc nhánh DFS gốc r tới một đỉnh thăm trước r hay không?

Dưới đây là một cách cài đặt rất thông minh, nội dung của nó là đánh số thứ tự các đỉnh theo thứ tự thăm đến. Định nghĩa $\text{num}[u]$ là số thứ tự của đỉnh u theo cách đánh số đó. Ta tính thêm $\text{low}[u]$ là giá trị $\text{num}[\cdot]$ nhỏ nhất trong các đỉnh có thể đến được từ một đỉnh v nào đó của nhánh DFS gốc u bằng một cung. Cụ thể cách tính $\text{low}[u]$ như sau:

Trong hàm $\text{DFSVisit}(u)$, trước hết ta đánh số thứ tự thăm cho đỉnh u : $\text{num}[u]$ và khởi tạo $\text{low}[u] = +\infty$. Sau đó xét các đỉnh v nối từ u , có hai khả năng:

- ✿ Nếu v đã thăm thì ta cực tiểu hoá $\text{low}[u]$ theo công thức:

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

- ✿ Nếu v chưa thăm thì ta gọi đệ quy $\text{DFSVisit}(v)$ để xây dựng nhánh DFS gốc v (con của u), cũng là để tính giá trị $\text{low}[v]$ (bằng $\text{num}[\cdot]$ nhỏ nhất của các đỉnh có thể đến được từ nhánh DFS gốc v), sau đó cực tiểu hoá $\text{low}[u]$ theo công thức:

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

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

Để công việc dễ dàng hơn nữa, ta sử dụng một ngăn xếp để chứa các đỉnh thuộc một nhánh DFS: Khi thăm đến 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 ngay sau u . Nếu u là chốt, ta chỉ việc lấy các đỉnh ra khỏi ngăn xếp cho tới khi lấy tới đỉnh u là sẽ được nhánh DFS gốc u , đây cũng chính là thành phần liên thông mạnh chứa u .

* Mô hình

Chi tiết thuật toán Tarjan:

```

1 void DFSVisit(u ∈ V)
2 {
3     num[u] = ++Time; //Ghi nhận thời điểm thăm u, cũng là đánh dấu ≠ 0
4     low[u] = +∞;
5     «Đẩy u vào ngăn xếp»;
6     for (∀v: (u, v) ∈ E) //Xét mọi đỉnh v nối từ u
7     {
8         if («v đã bị xóa»)
9             continue; //bỏ qua
10        if (num[v] != 0) //v đã thăm
11            low[u] = min(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
12        else //v chưa thăm
13        {
14            DFSVisit(v); //Đi thăm v, cũng là tính luôn low[v]
15            low[u] = min(low[u], low[v]); //Cực tiểu hóa low[u] theo low[v]
16        }
17    }
18    if (low[u] ≥ num[u]) //u đã duyệt xong, nhận thấy u là chốt
19        «Xuất các đỉnh lấy ra từ ngăn xếp và xóa luôn cho tới khi u bị xóa»;
20 }
21
22 for (∀u ∈ V) //num[u] = thời điểm thăm u, bằng 0 nếu chưa thăm
23     num[v] = 0; //Các đỉnh đều chưa thăm
24 Time = 0;
25 for (∀u ∈ V)
26     if (num[u] == 0) //Gặp đỉnh u chưa thăm
27         DFSVisit(u); //DFS từ u

```

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

✳ Cài đặt

Các nhân $num[.]$ trong thuật toán dùng để lưu thời điểm thăm một đỉnh. Thời điểm này là một giá trị số nguyên trong phạm vi từ 1 tới n , vì vậy ta dùng giá trị $num[.] = 0$ để đánh dấu một đỉnh chưa thăm và giá trị $num[.] = 1$ để đánh dấu một đỉnh đã bị xóa.

TARJAN.cpp Thuật toán Tarjan

```
1  #include <iostream>
2  #include <vector>
3  #include <stack>
4  using namespace std;
5  const int maxN = 1e6;
6  int n, m; //Số đỉnh và số cung
7  vector<int> adj[maxN]; //Các danh sách kề
8  int num[maxN], low[maxN];
9  stack<int> Stack;
10 int Time;
11
12 void ReadInput() //Nhập dữ liệu
13 {
14     cin >> n >> m;
15     while (m-- > 0) //Đọc m cung
16     {
17         int u, v;
18         cin >> u >> v; //Đọc cung (u, v)
19         adj[u].push_back(v); //Đưa v vào danh sách kề của u
20     }
21 }
22
23 void Minimize(int& Target, int Value) //Target = min(Target, Value)
24 {
25     if (Value < Target) Target = Value;
26 }
27
```

```

28 void DFSVisit(int u) //Thuật toán DFS
29 {
30     num[u] = ++Time; //Đánh số u theo thứ tự thăm, cũng là đánh dấu đã thăm ( $\neq 0$ )
31     low[u] = n + 1; //+ $\infty$ 
32     Stack.push(u); //Đẩy u vào ngăn xếp
33     for (int v: adj[u]) //Xét các đỉnh v nối từ u
34     {
35         if (num[v] == -1) continue; //v đã bị xóa, bỏ qua
36         if (num[v] != 0) //v đã thăm
37             Minimize(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
38         else //v chưa thăm
39         {
40             DFSVisit(v); //Thăm v, tính low[v]
41             Minimize(low[u], low[v]); //Cực tiểu hóa low[u] theo low[v]
42         }
43     }
44     if (low[u] >= num[u]) //u đã duyệt xong, phát hiện u là chốt
45     {
46         cout << "Strongly connected component:\n";
47         int v;
48         do //Liệt kê thành phần liên thông mạnh
49         {
50             v = Stack.top(); Stack.pop(); //Lấy một đỉnh v khỏi ngăn xếp
51             cout << v << ", "; //Output v thuộc thành phần liên thông mạnh chốt u
52             num[v] = -1; //Đánh dấu xóa v
53         }
54         while (v != u); //Khi u đã bị lấy ra khỏi ngăn xếp thì dừng
55         cout << "\n";
56     }
57 }
58
59 void Tarjan()
60 {
61     Time = 0;
62     fill(num, num + n, 0); //Các đỉnh đều chưa thăm
63     for (int u = 0; u < n; ++u)
64         if (num[u] == 0)
65             DFSVisit(u);
66 }
67
68 int main()
69 {
70     ReadInput();
71     Tarjan();
72 }

```


1.4.4. Thuật toán Kosaraju-Sharir

* Ý tưởng của thuật toán

Có một thuật toán khác để liệt kê các thành phần liên thông mạnh là thuật toán Kosaraju-Sharir (Sharir, 1981). Thuật toán này thực hiện qua hai bước:

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

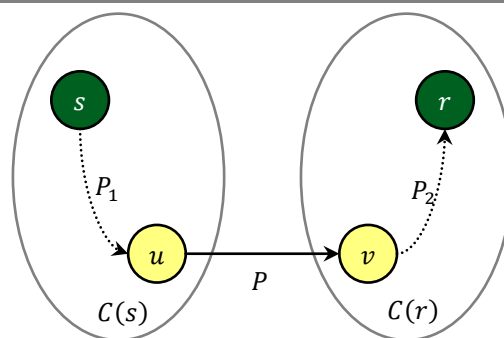
Bổ đề 1-5

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

Chứng minh

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

Gọi s là chốt của một thành phần liên thông mạnh $C(s)$ khác. Ta chứng minh rằng không thể tồn tại cung đi từ $C(s)$ sang $C(r)$, giả sử phản chứng rằng có cung (u, v) trong đó $u \in C(s)$ và $v \in C(r)$. Khi đó tồn tại một đường đi $P_1: s \rightsquigarrow u$ trong nội bộ $C(s)$ và tồn tại một đường đi $P_2: v \rightsquigarrow r$ trong nội bộ $C(r)$. Nối đường đi $P_1: s \rightsquigarrow u$ với cung (u, v) và nối tiếp với đường đi $P_2: v \rightsquigarrow r$ ta được một đường đi $P: s \rightsquigarrow r$ (Hình 1-8)



Hình 1-8

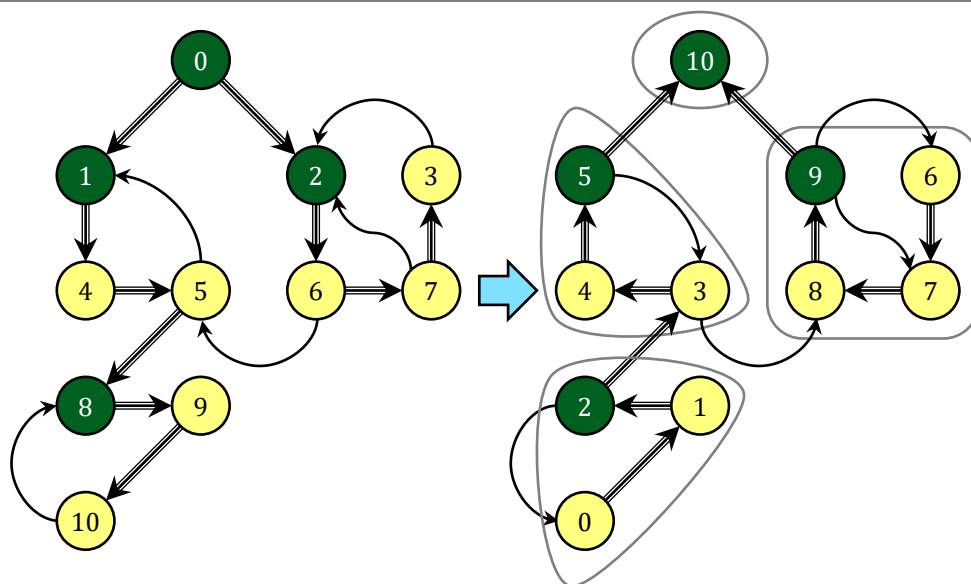
Theo tính chất của chốt là đỉnh thăm trước mọi đỉnh khác thuộc cùng thành phần liên thông mạnh, s sẽ được thăm trước mọi đỉnh $\in P_1$ và r sẽ được thăm trước mọi đỉnh $\in P_2$. Có hai khả năng xảy ra:

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

Bổ đề được chứng minh.

Bổ đề 1-5 chỉ ra tính đúng đắn của thuật toán Kosaraju-Sharir: Đỉnh r duyệt xong sau cùng chắc chắn là chốt của một thành phần liên thông mạnh và thành phần liên thông mạnh này gồm mọi đỉnh đến được r . Việc liệt kê các đỉnh thuộc thành phần liên thông mạnh chốt r được thực hiện trong thuật toán thông qua thao tác đảo chiều các cung của đồ thị rồi liệt kê các đỉnh đến được từ r .

Loại bỏ thành phần liên thông mạnh với chốt r khỏi đồ thị. Cây DFS gốc r lại phân rã thành nhiều cây con. Lập luận tương tự như trên với đỉnh duyệt xong sau cùng (Hình 1-9).



Hình 1-9. Đánh số lại, đảo chiều các cung và thực hiện BFS/DFS với cách chọn đỉnh xuất phát ngược lại với thứ tự duyệt xong (thứ tự 10, 9, 5, 2 ở hình bên phải)

✿ Cài đặt

Để đánh số lại các đỉnh theo thứ tự ngược với thứ tự duyệt xong, khi thực hiện DFS, mỗi khi duyệt xong một đỉnh thì đỉnh đó được đưa vào đầu một danh sách. Sau khi

đảo chiều các cung của đồ thị, chúng ta chỉ cần duyệt danh sách từ đầu đến cuối là sẽ được các đỉnh đúng thứ tự ngược với thứ tự duyệt xong. Có rất nhiều cấu trúc dữ liệu cho phép biểu diễn danh sách dạng này, trong chương trình dưới đây ta dùng hàng đợi hai đầu (Deque) để biểu diễn danh sách.

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

KOSARAJUSHARIR.cpp Thuật toán Kosaraju-Sharir

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  using namespace std;
5  const int maxN = 1e6;
6  int n, m; //Số đỉnh và số cung
7  vector<int> adjF[maxN], adjR[maxN]; //Danh sách kề forward/reverse star
8  deque<int> Q; //Danh sách các đỉnh ngược lại với thứ tự duyệt xong
9  bool avail[maxN];
10
11 void ReadInput() //Nhập dữ liệu
12 {
13     cin >> n >> m;
14     while (m-- > 0) //Đọc m cung
15     {
16         int u, v;
17         cin >> u >> v; //Đọc 1 cung (u, v)
18         adjF[u].push_back(v); //Đưa v vào danh sách kề forward star của u
19         adjR[v].push_back(u); //Đưa u vào danh sách kề reverse star của v
20     }
21 }
22
23 void Scan(int u) //DFS trên đồ thị nguyên bản
24 {
25     avail[u] = false;
26     for (int v: adjF[u]) //Xét danh sách forward star
27         if (avail[v]) Scan(v); //Gặp v chưa thăm, DFS tiếp từ v
28     Q.push_front(u); //duyet xong u, đưa u vào đầu danh sách Q
29 }
30
31 void Enum(int u) //DFS trên đồ thị đảo chiều
32 {
33     cout << u << " ";
34     avail[u] = false;
35     for (int v: adjR[u]) //Xét danh sách reverse star
36         if (avail[v]) Enum(v); //Gặp v chưa thăm, DFS tiếp từ v
37 }
38
```

```

39 void KosarajuSharir()
40 {
41     fill(avail, avail + n, true); //Đánh dấu mọi đỉnh đều chưa thăm
42     for (int u = 0; u < n; ++u)
43         if (avail[u]) Scan(u); //DFS trên đồ thị nguyên bản
44     fill(avail, avail + n, true); //Đánh dấu lại mọi đỉnh đều chưa thăm
45     for (int u: Q) //Duyệt các đỉnh ngược lại với thứ tự duyệt xong ở bước trước
46         if (avail[u])
47         {
48             cout << "Strongly connected component:\n";
49             Enum(u); //In ra thành phần liên thông mạnh chứa u
50             cout << "\n";
51         }
52 }
53
54 int main()
55 {
56     ReadInput();
57     KosarajuSharir();
58 }

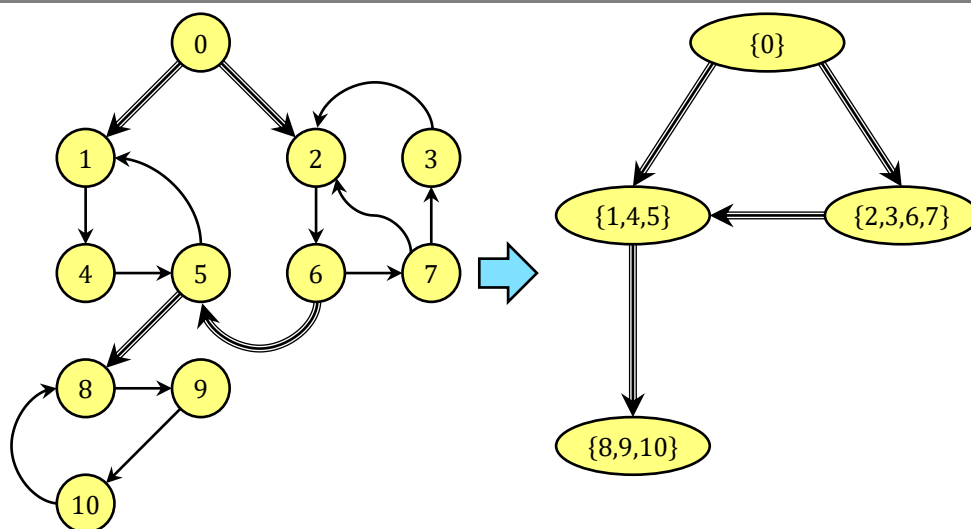
```

Thời gian thực hiện giải thuật có thể tính bằng hai lượt DFS, vậy nên thời gian thực hiện giải thuật sẽ là $\Theta(|V| + |E|)$ trong trường hợp đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, là $\Theta(|V|^2)$ nếu dùng ma trận kề và là $\Theta(|V||E|)$ nếu dùng danh sách cạnh.

1.5. Sắp xếp tô pô

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

Đồ thị G^{SCC} gọi là đồ thị các thành phần liên thông mạnh (Hình 1-10)



Hình 1-10. Đồ thị có hướng và đồ thị các thành phần liên thông mạnh

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

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

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

Bài tập 1-1

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

Bài tập 1-2

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

Bài tập 1-3


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

Bài tập 1-4

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

một số các nhóm thỏa mãn: Mỗi nhóm gồm các hình giống nhau và hai hình bất kỳ thuộc thuộc hai nhóm khác nhau thì không giống nhau:

1	1	1	0	1	1	0	0	1
1	0	0	0	1	0	0	1	1
1	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1
1	0	1	0	0	1	1	0	1
1	1	1	1	1	0	0	1	1



1	1	1	0	2	2	0	0	2
1	0	0	0	2	0	0	2	2
1	1	0	0	0	0	0	0	0
1	0	0	3	0	0	0	0	0
1	0	0	3	0	0	0	0	0
0	0	3	3	0	3	0	0	0
1	0	0	0	0	3	0	0	3
1	0	1	0	0	3	3	0	3
1	1	1	1	1	0	0	3	3

Bài tập 1-5

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

Bài tập 1-6

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

- Chứng minh rằng đồ thị có hướng $G = (V, E)$ là nửa liên thông nếu và chỉ nếu trên G tồn tại đường đi qua tất cả các đỉnh (không nhất thiết phải là đường đi đơn)
- Tìm thuật toán và viết chương trình kiểm tra tính nửa liên thông của đồ thị.