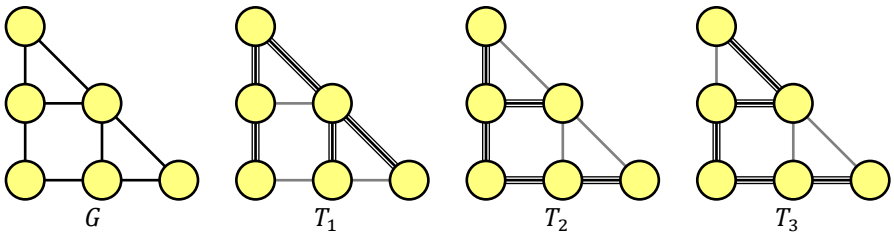
Chương 1. Một số ứng dụng của DFS và BFS

1.1. 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.

Xét đồ thị $G = (V, E)$ và $T = (V, E_T)$ là một đồ thị con của đồ thị G ($E_T \subseteq E$), nếu T là một cây thì ta gọi T là *cây khung* hay *cây bao trùm* (*spanning tree*) của đồ thị G. Điều kiện cần và đủ để một đồ thị vô hướng có cây khung là đồ thị đó phải liên thông. Một đồ thị vô hướng liên thông có thể có nhiều cây khung.



Hình 1-1. Đồ thị G và các cây khung T_1, T_2, T_3 của nó.

Định lý 1-1

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:

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 $n > 1$, gọi $P = \langle v_1, v_2, \dots, v_k \rangle$ là đường đi dài nhất (qua nhiều cạnh nhất) trong T. Đỉnh v_1 không thể kề với đỉnh nào trong số các đỉnh $v_3, v_4, \dots, v_k$, bởi nếu có cạnh (v_1, v_p) $(3 \le p \le k)$, ta sẽ thiết lập được chu trình đơn $\langle v_1, v_2, \dots, v_p, v_1 \rangle$. Mặt khác, đỉnh v_1 cũng không thể kề với đỉnh nào khác ngoài các đỉnh trên đường đi P trên bởi nếu có cạnh $(v_0, v_1) \in E$, $v_0 \notin P$ thì ta thiết lập được đường đi $\langle v_0, v_1, v_2, \dots, v_k \rangle$ dài hơn P. Vậy đỉnh v_1 chỉ có đúng một cạnh nối với v_2, nói cách khác, 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:

Giả sử T có k thành phần liên thông $T_1, T_2, \dots, T_k$. Vì T không chứa chu trình đơn nên các thành phần liên thông của T cũng không chứa chu trình đơn, tức là các $T_1, T_2, \dots, T_k$ đều là cây. Gọi $n_1, n_2, \dots, n_k$ lần lượt là số đỉnh của $T_1, T_2, \dots, T_k$ 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 $n-k$ 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, kết hợp với giả thiết T không có chu trình nên nếu bỏ đi một cạnh bất kỳ thì đồ thị mới vẫn không chứa chu trình. Đồ thị mới này không thể liên thông vì nếu không nó sẽ phải là một cây và theo chứng mình trên, đồ thị mới sẽ có $n-1$ cạnh, tức là T có n cạnh. Mâu thuẫn này chứng tỏ tất cả các cạnh của T đều là cầu.

3⇒4:

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 chỉ ra việc bỏ đi cạnh (u,v) không ảnh hưởng tới việc đi lại được giữa hai đỉnh bất kỳ. Mâu thuẫn với giả thiết (u,v) là cầu.

4⇒5:

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:

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:

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.

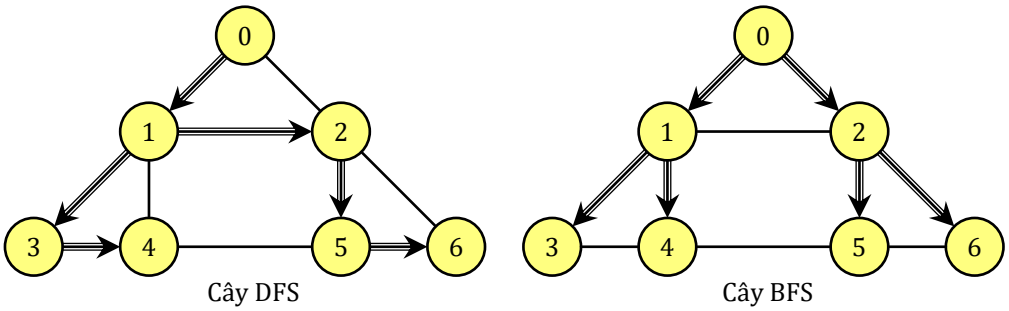
Ta sẽ khảo sát hai thuật toán tìm cây khung trên đồ thị vô hướng liên thông $G = (V, E)$.

1.1.1. Xây dựng cây khung bằng thuật toán hợp nhất

Trước hết, đặt $T = (V, \emptyset)$; T không chứa cạnh nào thì có thể coi T gồm $|V|$ 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 $|V| - 1$ cạnh vào T thì ta được T là cây khung của đồ thị. Trong việc xây dựng cây khung bằng thuật toán hợp nhất, một cấu trúc dữ liệu biểu diễn các tập rời nhau (NEEDREF) thường được sử dụng để tăng tốc phép hợp nhất hai cây cũng như phép kiểm tra hai đỉnh có thuộc hai cây khác nhau không.

1.1.2. 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 nào đó, 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 $|V| - 1$ 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 1-2 là ví dụ về cây khung DFS và cây khung BFS của cùng một đồ thị (danh sách kề của mỗi đỉnh được sắp xếp theo thứ tự tăng dần).



Hình 1-2. Cây khung DFS và cây khung BFS của cùng một đồ thị

1.2. Tập các chu trình cơ sở của đồ thị

Xét một đồ thị vô hướng liên thông $G = (V, E)$; gọi $T = (V, E_T)$ 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 cây.

Nếu thêm một cạnh ngoài $e \in E - E_T$ vào cây khung T, thì ta được đúng một chu trình đơn trong T, ký hiệu chu trình này là C_e. Chu trình C_e chỉ chứa duy nhất một cạnh ngoài cây còn các cạnh còn lại đều là cạnh trong cây T

Tập các chu trình:

$$\Psi = \{C_e | e \in E - E_T\}$$

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:

- 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.
- Cây khung của đồ thị liên thông $G = (V, E)$ luôn chứa $|V| - 1$ cạnh, còn lại $|E| - |V| + 1$ 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 $|E| - |V| + 1$.
- 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. Điều này có thể chứng minh được bằng cách lấy trong đồ thị liên thông một tập gồm k 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 $|E| - k$ cạnh. Đồ thị liên thông thì không thể có ít hơn $|V| - 1$ cạnh nên ta có $|E| - k \geq |V| - 1$ hay $k \leq |E| - |V| + 1$.
- Mọi cạnh trong một chu trình đơn bất kỳ đều phải thuộc ít nhất 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 $|E| - |V| + 1$ cạnh ngoài của các chu trình cơ sở thì đồ thị vẫn liên thông và còn lại $|V| - 2$ cạnh. Điều này vô lý.

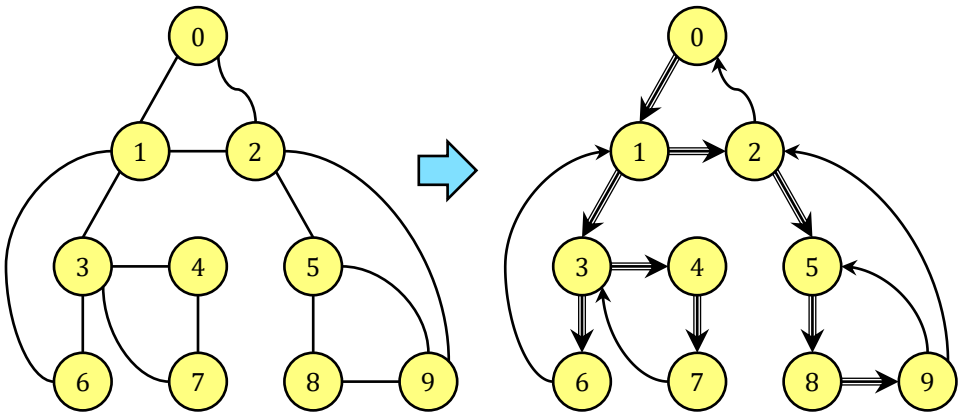
Đối với đồ thị $G = (V, E)$ có k thành phần liên thông, ta có thể xét các thành phần liên thông và xét rừng các cây khung của các thành phần đó. Khi đó có thể mở rộng khái niệm tập các chu trình cơ sở cho đồ thị vô hướng tổng quát: Mỗi khi thêm một cạnh $\in E$ nhưng không thuộc các cây khung vào rừng, 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 $|E| - |V| + k$.

1.3. Bài toán định chiều đồ thị

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.

Có thể tổng quát hoá bài toán định chiều đồ thị: Với đồ thị vô hướng $G = (V, E)$ hãy tìm cách thay mỗi cạnh của đồ thị bằng một cung định hướng để được đồ thị mới có ít thành phần liên thông mạnh nhất. Dưới đây ta xét một tính chất hữu ích của thuật toán thuật toán tìm kiếm theo chiều sâu để giải quyết bài toán định chiều đồ thị

Xét mô hình duyệt đồ thị bằng thuật toán tìm kiếm theo chiều sâu, tuy nhiên trong quá trình duyệt, mỗi khi xét qua cạnh (u, v) thì ta định chiều luôn cạnh đó thành cung (u, v). Nếu coi một cạnh của đồ thị tương đương với hai cung có hướng ngược chiều nhau thì việc định chiều cạnh (u, v) thành cung (u, v) tương đương với việc loại bỏ cung (v, u) của đồ thị. Ta có một phép định chiều gọi là phép định chiều DFS.



Hình 1-3. Phép định chiều DFS

Thuật toán thực hiện phép định chiều DFS có thể viết như sau:

```
void DFSVisit(u ∈ V)
{
    «Đánh dấu u đã thăm»;
    for (∀v: (u, v)∈ E)
    {
        «Định chiều cạnh (u, v) thành cung (u, v) ⇔ xóa cung (v, u)»;
        if («v chưa thăm»)
            DFSVisit(v);
    }
}

«Đánh dấu mọi đỉnh đều chưa thăm»;
for ∀v ∈ V do
    if («v chưa thăm») DFSVisit(v);
```

Thuật toán DFS sẽ cho một rừng các cây DFS và các cung ngoài cây. Ta có các tính chất sau:

Bổ đề 1-2

Sau quá trình duyệt DFS và định chiều, đồ thị sẽ chỉ còn cung DFS và cung ngược.

Chứng minh

Xét một cạnh (u, v) bất kỳ, không giảm tính tổng quát, giả sử rằng u được thăm đến trước v. Theo Định lý đường đi trắng, ta có v là hậu duệ của u. Nhìn vào mô hình cài đặt thuật toán, có nhận xét rằng việc định chiều cạnh (u, v) chỉ có thể được thực hiện trong hàm DFSVisit(u) hoặc trong hàm DFSVisit(v).

- Nếu cạnh (u, v) được định chiều trước khi đỉnh v được thăm đến, nghĩa là việc định chiều được thực hiện trong hàm DFSVisit(u), và ngay sau khi cạnh (u, v) được định chiều thành cung (u, v) thì đỉnh v sẽ được thăm. Điều đó chỉ ra rằng cung (u, v) là cung DFS.
- Nếu cạnh (u, v) được định chiều sau khi đỉnh v được thăm đến, nghĩa là khi hàm DFSVisit(v) được gọi thì cạnh (u, v) chưa định chiều. Vòng lặp bên trong hàm DFSVisit(v) chắc chắn sẽ quét vào cạnh này và định chiều thành cung ngược (v, u).

Trong đồ thị vô hướng ban đầu, cạnh bị định hướng thành cung ngược chính là cạnh ngoài của cây DFS. Chính vì vậy, mọi chu trình cơ sở của cây DFS 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 1-3

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 phép định chiều đó ta sẽ thu được đồ thị liên thông mạnh G'. Với một cạnh (u, v) 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.

Lấy một cạnh (u, v) của G, vì (u, v) nằm trong một chu trình đơn, mà mọi cạnh của một chu trình đơn đều phải thuộc ít nhất một chu trình cơ sở của cây DFS, nên sẽ có một chu trình cơ sở chứa cạnh (u, v). Có thể nhận thấy rằng chu trình cơ sở của cây DFS qua phép định chiều DFS vẫn là chu trình trong G' nên theo các cung đã định hướng của chu trình đó ta có thể đi từ u tới v và ngược lại.

Lấy x và y là hai đỉnh bất kỳ của G, do G liên thông, tồn tại một đường đi

$$\langle x = v_0, v_1, \dots, v_k = y \rangle$$

Vì (v_i, v_{i+1}) là cạnh của G nên theo chứng minh trên, từ v_i có thể đi đến được v_{i+1} trên G', $\forall i: 1 \leq i < k$, tức là từ x vẫn có thể đi đến y bằng các cung định hướng của G'. Suy ra G' là đồ thị liên thông mạnh

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.

1.4. Liệt kê các khớp và cầu của đồ thị

1.4.1. Thuật toán

Việc kiểm tra tính chất của khớp/cầu có thể thực hiện trực tiếp bằng định nghĩa (Đếm số thành phần liền thông, thử xóa đỉnh/cạnh rồi đếm lại số thành phần liên thông). Tuy vậy nếu áp dụng cách thức này để liệt kê tất cả các khớp cũng như cầu thì thuật toán có độ phức tạp khá lớn. Trong phần này ta sẽ trình bày thuật toán $O(|V| + |E|)$ liệt kê các khớp/cầu của đồ thị vô hướng dựa trên mô hình DFS.

Xét phép duyệt đồ thị bằng DFS, ta đánh số các đỉnh theo thứ tự thăm đến và gọi $num[u]$ là số thứ tự của đỉnh u theo cách đánh số đó. Định nghĩa thêm $low[u]$ là giá trị $num[.]$ nhỏ nhất của những đỉnh đến được từ nhánh DFS gốc u bằng một

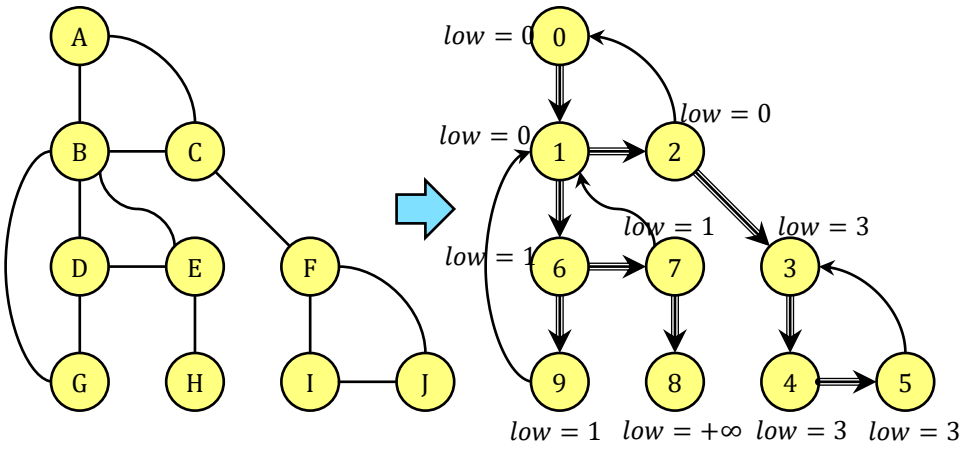
cung ngược (tức là nếu nhánh DFS gốc u có nhiều cung ngược hướng lên phía gốc thì ta ghi nhận lại cung ngược hướng lên cao nhất). Nếu nhánh DFS gốc u không chứa cung ngược thì ta cho $low[u] = +\infty$. Cách tính các giá trị $num[.]$ và $low[.]$ tương tự như thuật toán Tarjan tìm thành phần liên thông mạnh: Trong hàm DFSVisit(u), trước hết $num[u]$ được gán bằng số thứ tự thăm của đỉnh u và $low[u]$ được khởi tạo bằng $+\infty$. Sau đó với từng đỉnh v kề u, thuật toán định chiều cạnh (u, v) thành cung (u, v). Có hai khả năng xảy ra:

- Nếu v chưa thăm thì ta gọi $DFSVisit(v)$ để thăm v. Khi hàm DFSVisit(v) thoát, nhánh DFS gốc v đã xây dựng xong và là con của u. Vì những cung ngược đi từ nhánh DFS gốc v cũng là cung ngược đi từ nhánh DFS gốc u, vì vậy $low[u]$ được cực tiểu hóa theo $low[v]$:

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

- Nếu v đã thăm thì (u, v) là một cung ngược đi từ nhánh DFS gốc u, trong trường hợp này $low[u]$ được cực tiểu hóa theo $num[v]$:

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



Hình 1-4. Cách đánh số và ghi nhận cung ngược lên cao nhất

Hãy để ý một cung DFS (u, v) (u là nút cha của nút v trên cây DFS)

- Nếu từ nhánh DFS gốc v không có cung nào ngược lên phía trên v có nghĩa là từ một đỉnh thuộc nhánh DFS gốc v đi theo các cung định hướng chỉ đi được tới những đỉnh nội bộ trong nhánh DFS gốc v mà thôi chứ không thể tới được u, suy ra (u, v) là một cầu. Cũng dễ dàng chứng minh được điều ngược lại. Vậy (u, v) là cầu nếu và chỉ nếu $low[v] \geq num[v]$. Như ví dụ ở Hình 1-4, ta có (C, F) và (E, H) là cầu.
- Nếu từ nhánh DFS gốc v không có cung nào ngược lên phía trên u, tức là nếu bỏ u đi thì từ v không có cách nào lên được các tiền bối của u. Điều này chỉ ra rằng nếu u không phải là nút gốc của một cây DFS thì u là khớp. Cũng không khó khăn để chứng minh điều ngược lại. Vậy nếu u không là gốc của một cây DFS thì u là khớp nếu và chỉ nếu tồn tại một đỉnh v con của u mà $low[v] \geq num[u]$. Như ví dụ ở Hình 1-4, ta có B, C, E và F là khớp.

- Gốc của một cây DFS thì là khớp nếu và chỉ nếu nó có từ hai 2 nhánh con trở lên. Như ví dụ ở Hình 1-4, gốc A không là khớp vì nó chỉ có một nhánh con trên cây DFS.

Đến đây ta đã có đủ điều kiện để giải bài toán liệt kê các khớp và cầu của đồ thị: đơn giản là dùng phép định chiều DFS đánh số các đỉnh theo thứ tự thăm và ghi nhận cung ngược lên trên cao nhất xuất phát từ một nhánh cây DFS, sau đó dùng ba nhận xét kể trên để liệt kê ra tất cả các cầu và khớp của đồ thị.

1.4.2. Cài đặt

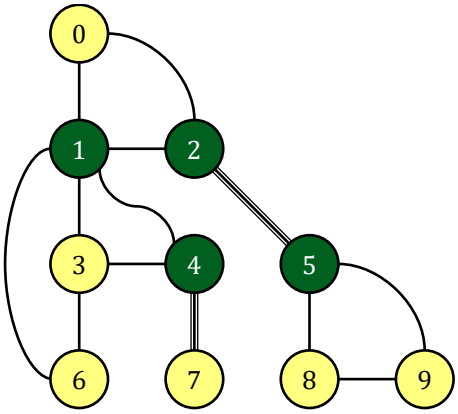
Input

- Dòng 1: Chứa hai số nguyên dương $n, m \leq 10^6$ lần lượt là số đỉnh và số cạnh của đồ thị vô hướng G.
- m dòng tiếp theo, mỗi dòng chứa hai số u, v tương ứng với một cạnh (u, v) của G

Output

Các khớp và cầu của G

Sample Input	Sample Output
10 13 0 1 0 2 1 2 1 3 1 4 1 6 2 5 3 4 3 6 4 7 5 8 5 9 8 9	Bridges: 6: (2, 5) 9: (4, 7) Articulations: 1 2 4 5



Đồ thị được biểu diễn bằng một danh sách cạnh e, mỗi cạnh là một bản ghi (struct) có hai trường x, y là chỉ số hai đỉnh đầu mút của cạnh. Mỗi đỉnh u được gắn với một danh sách liên thuộc $adj[u]$ chứa chỉ số* các cạnh liên thuộc với u. Với một cạnh $e[i]$ liên thuộc với u, để chỉ ra đỉnh v kề với u qua cạnh e có thể dùng công thức $v = e[i].x + e[i].y - u$.

Việc định chiều đồ thị chẳng qua là thao tác "ép" quá trình DFS chỉ được duyệt qua mỗi cạnh một lần theo một chiều. Cơ chế này được thực hiện trong chương trình

---

* Có thể tối ưu hóa cách cài đặt bằng cách dùng $adj[u]$ chứa các con trỏ tới các cạnh liên thuộc với u

như sau: khi duyệt qua mỗi cạnh (u, v), ta sẽ đánh dấu vô hiệu hóa luôn cạnh đó để nếu quá trình DFS về sau duyệt vào cạnh này (theo chiều ngược lại (v, u)) sẽ bỏ qua luôn.

CUTVE.cpp Liệt kê các khớp và cầu của đồ thị

```cpp
#include <iostream>
#include <vector>
using namespace std;
const int maxN = 1e6;
const int maxM = 1e6;
int n, m;

struct TEdge //Cấu trúc cạnh
{
    int x, y; //Hai đỉnh đầu mút
    bool invalid; //Cạnh đã đã vô hiệu hóa chưa
} e[maxM];

vector<int> adj[maxN]; //Các danh sách liên thuộc
int num[maxN], low[maxN];
bool isCut[maxN], isBridge[maxM]; //Đánh dấu đỉnh/cạnh có phải khớp/cầu không

void ReadInput() //Đọc dữ liệu
{
    cin >> n >> m;
    for (int i = 0; i < m; ++i)
    {
        cin >> e[i].x >> e[i].y; //Đọc cạnh thứ i
        e[i].invalid = false; //Cạnh không bị vô hiệu hóa
        adj[e[i].x].push_back(i); //Thêm i vào danh sách liên thuộc của e[i].x
        adj[e[i].y].push_back(i); //Thêm i vào danh sách liên thuộc của e[i].y
    }
}

inline void Minimize(int& Target, int Value) //Target = min(Target, Value)
{
    if (Value < Target) Target = Value;
}

```

```cpp
void DFSVisit(int u) //Thuật toán DFS bắt đầu từ u
{
    bool isRoot = num[u] == -2; //isRoot: u có phải gốc một cây DFS không
    static int Time = 0; //Biến đếm số thứ tự thăm
    num[u] = Time++; //Đánh số u theo thứ tự thăm
    low[u] = maxN; //low[u] khởi tạo bằng +∞
    int nBranches = 0; //Biến đếm số nhánh con của u trên cây DFS
    for (int i: adj[u])
    {   //Xét cạnh e[i] liên thuộc u
        if (e[i].invalid) continue; //e[i] đã bị vô hiệu hóa, bỏ qua
        e[i].invalid = true; //vô hiệu hóa luôn cạnh e[i] vừa duyệt qua
        int v = e[i].x + e[i].y - u; //Xét đỉnh v kề u qua e[i]
        if (num[v] == -1) //v chưa thăm
        {
            DFSVisit(v); //Thăm v
            Minimize(low[u], low[v]); //Cực tiểu hóa low[u] theo low[v]
            isBridge[i] = low[v] >= num[v]; //low[v] ≥ num[v] → e[i] là cầu
            isCut[u] |= low[v] >= num[u]; //low[v] ≥ num[u] → u là khớp
            ++nBranches; //Đếm số nhánh con của u trên cây DFS
        }
        else //v đã thăm
            Minimize(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
    }
    if (isRoot && nBranches < 2) //Nếu u là gốc và có ít hơn 2 nhánh con
        isCut[u] = false; //→ u đã bị đánh dấu nhầm là khớp nên phải đặt lại
}

void Solve()
{
    fill(num, num + n, -1); //Các đỉnh đều chưa thăm, được đánh số bởi giá trị âm
    fill(isCut, isCut + n, false);
    fill(isBridge, isBridge + m, false);
    for (int u = 0; u < n; ++u)
        if (num[u] == -1)
        {
            num[u] = -2; //Cho biết u là gốc cây DFS trước khi thăm
            DFSVisit(u); //Duyệt DFS từ gốc u
        }
    cout << "Bridges:\n"; //In các cầu
    for (int i = 0; i < m; ++i)
        if (isBridge[i])
            cout << i << ": (" << e[i].x << ', ' << e[i].y << ")\n";
    cout << "Articulations:\n"; //In các khớp
    for (int u = 0; u < n; ++u)
        if (isCut[u]) cout << u << '\n';
}

int main()
{
    ReadInput();
    Solve();
}
```

Việc cài đặt thuật toán liệt kê khớp/cầu dễ gặp sai sót, nhất là trong trường hợp đồ thị có khuyên, đỉnh cô lập hay cạnh song song. Vì vậy cần có thiết kế và kiểm thử cẩn thận trước và sau khi viết chương trình. Một lưu ý nữa là nếu chỉ liệt kê khớp,

ta không cần định chiều đồ thị, chỉ đơn giản là coi mỗi cạnh vô hướng tương ứng với hai cung có hướng ngược chiều nhau.

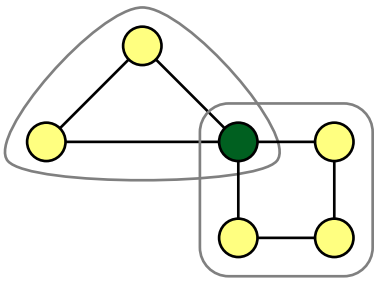
1.5. Các thành phần song liên thông

1.5.1. Các khái niệm và thuật toán

Đồ thị vô hướng liên thông được gọi là đồ thị song liên thông nếu nó không có khớp, tức là việc bỏ đi một đỉnh bất kỳ của đồ thị không ảnh hưởng tới tính liên thông của các đỉnh còn lại. Ta quy ước rằng đồ thị chỉ gồm một đỉnh và không có cạnh nào cũng là một đồ thị song liên thông.

Cho đồ thị vô hướng $G = (V, E)$, xét một tập con $V' \subset V$. Gọi G' là đồ thị G hạn chế trên V'. Đồ thị G' được gọi là một thành phần song liên thông của đồ thị G nếu G' song liên thông và không tồn tại đồ thị con song liên thông nào khác của G nhận G' làm đồ thị con. Ta cũng đồng nhất khái niệm G' là thành phần song liên thông với khái niệm V' là thành phần song liên thông.

Cần phân biệt hai khái niệm đồ thị định chiều được (không có cầu) và đồ thị song liên thông (không có khớp). Nếu như đồ thị G không định chiều được thì *tập đỉnh* của G có thể phân hoạch thành các tập con rời nhau để đồ thị G hạn chế trên các tập con đó là các đồ thị định chiều được. Còn nếu đồ thị G không phải đồ thị song liên thông thì *tập cạnh* của G có thể phân hoạch thành các tập con rời nhau để trên mỗi tập con, các cạnh và các đỉnh đầu mút của chúng trở thành một đồ thị song liên thông. Hai thành phần song liên thông có thể có chung một đỉnh khớp nhưng không có cạnh nào chung*.



Hình 1-5. Đồ thị và hai thành phần liên thông có chung khớp

Coi mỗi cạnh vô hướng tương ứng với hai cung có hướng ngược chiều nhau. Xét mô hình đánh số đỉnh theo thứ tự thăm đến và ghi nhận cung ngược lên cao nhất...

* Một số tài liệu định nghĩa một thành phần song liên thông là một tập tối đại các **cạnh** sao cho các cạnh này và các đầu mút của chúng tạo ra một đồ thị song liên thông, định nghĩa này tương đồng với định nghĩa trong bài ngoại trừ việc không chấp nhận thành phần song liên thông chỉ gồm một đỉnh cô lập.

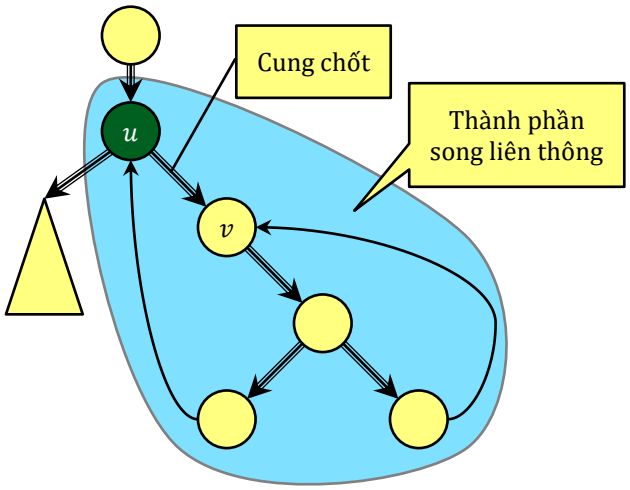
```
void DFSVisit(u ∈ V)
{
    «Đánh dấu u đã thăm»;
    num[u] = «Số thứ tự thăm đỉnh u»;
    low[u] = +∞;
    for (∀v ∈ adj[u]) //Xét mọi đỉnh v kề u
        if («v chưa thăm»)
        {
            DFSVisit(v); //Đi thăm v
            low[u] = min(low[u], low[v]); //Cực tiểu hoá low[u] theo low[v]
        }
        else //v đã thăm
            low[u] = min(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
}

«Đánh dấu mọi đỉnh chưa thăm»;
for (∀u ∈ V)
    if («u chưa thăm») DFSVisit(v);
```

Trong sơ đồ cài đặt này, $num[u]$ là số hiệu của đỉnh u theo thứ tự thăm đến và $low[u]$ là giá trị $num[.]$ nhỏ nhất của một đỉnh kề với nhánh DFS gốc u (đỉnh cao nhất trên cây DFS kề với một đỉnh thuộc nhánh DFS gốc u).

Xét hàm DFSVisit(u), mỗi khi xét các đỉnh v kề u chưa được thăm, thuật toán sẽ gọi DFSVisit(v) để đi thăm v sau đó cực tiểu hóa $low[u]$ theo $low[v]$. Tại thời điểm này, nếu $low[v] \geq num[u]$ thì hoặc u là khớp hoặc u là gốc của một cây DFS. Để tiện trình bày, trong trường hợp này ta gọi cung (u, v) là *cung chốt* của thành phần song liên thông.

Thuật toán tìm kiếm theo chiều sâu không chỉ duyệt qua các đỉnh mà còn duyệt và định chiều các cung nữa. Ta sẽ quan tâm tới cả thời điểm một cạnh được thăm đến, duyệt xong, cũng như thứ tự tiền bối–hậu duệ của các cung DFS: Cung DFS (u, v) được coi là tiền bối thực sự của cung DFS (u', v') (hay cung (u', v') là hậu duệ thực sự của cung (u, v)) nếu cung (u', v') nằm trong nhánh DFS gốc v. Xét về vị trí trên cây, cung (u', v') nằm dưới cung (u, v).

Có thể thấy rằng nếu (u, v) là một cung chốt thỏa mãn: Khi DFSVisit(u) gọi DFSVisit(v) và quá trình tìm kiếm theo chiều sâu tiếp tục từ v không thăm tiếp bất cứ một cung chốt nào (tức là nhánh DFS gốc v không chứa cung chốt nào) thì cung (u, v) hợp với tất cả các cung hậu duệ của nó sẽ tạo thành một nhánh cây mà mọi đỉnh thuộc nhánh cây đó là một thành phần song liên thông (Hình 1-6).



Hình 1-6. Cung chốt và thành phần song liên thông

Thuật toán liệt kê các thành phần song liên thông dựa trên chính nhận xét này: Mỗi khi quá trình DFS thăm tới một cung, cung này được đẩy vào một ngăn xếp để mỗi khi quá trình DFS duyệt xong một cung chốt (u, v), ta chỉ việc lấy các cung ra khỏi ngăn xếp cho tới khi lấy được cung (u, v), các đỉnh đầu mút của các cung lấy ra từ ngăn xếp chính là một thành phần song liên thông. Cơ chế thực hiện khá giống với thuật toán Tarjan tìm thành phần liên thông mạnh, chỉ là thay khái niệm "chốt" bởi "cung chốt" và ngăn xếp dùng để chứa các cung DFS thay vì chứa các đỉnh.

Vấn đề duy nhất còn phải xử lý là quy ước một đỉnh cô lập của đồ thị cũng là một thành phần song liên thông. Thuật toán trên chỉ liệt kê được thành phần song liên thông có ít nhất 1 cạnh nên sẽ bỏ sót các đỉnh cô lập. Ta sẽ phải xử lý các đỉnh cô lập như trường hợp riêng khi liệt kê các thành phần song liên thông của đồ thị.

```
void DFSVisit(u ∈ V)
{
    «Đánh dấu u đã thăm»;
    num[u] = «Số thứ tự thăm đỉnh u»;
    low[u] = +∞;
    for (∀v ∈ adj[u]) //Xét mọi đỉnh v kề u
        if («v chưa thăm»)
        {
            «Đẩy cung (u, v) vào ngăn xếp»;
            DFSVisit(v); //Đi thăm v
            low[u] = min(low[u], low[v]); //Cực tiểu hoá low[u] theo low[v]
            if (low[v] ≥ num[u]) //(u, v) là cung chốt
                «lấy các cung khỏi ngăn xếp cho tới khi lấy được cung (u, v)
                 Các đỉnh đầu mút tạo thành một thành phần song liên thông»;
        }
        else //v đã thăm
            low[u] = min(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
}

«Đánh dấu mọi đỉnh chưa thăm»;
for (∀u ∈ V)
    if («u chưa thăm»)
    {
        DFSVisit(u);
        if («u là đỉnh cô lập»)
            «Liệt kê thành phần song liên thông chỉ gồm 1 đỉnh u»;
    }
```

1.5.2. Cài đặt

Có một chi tiết nhỏ nhằm tăng hiệu suất của chương trình ở cách sử dụng ngăn xếp. Để ý rằng ngăn xếp chỉ chứa các cung DFS và khi lấy các cung ra khỏi ngăn xếp ứng với một thành phần song liên thông thì thực chất là ta lấy được cả một nhánh DFS. Để in ra các đỉnh của nhánh này thì ngoài đỉnh gốc nhánh, với mỗi cung DFS (u, v), ta chỉ đưa ra đầu mút v. Vì vậy thay vì lưu trữ một cung (u, v) trong ngăn xếp, ta chỉ cần lưu trữ đầu mút v của cung là đủ.

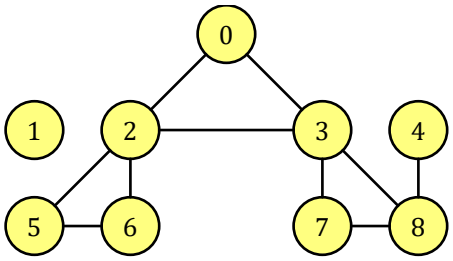
Input

- Dòng 1: Chứa hai số nguyên $n, m \leq 10^6$ lần lượt là số đỉnh và số cạnh của một đồ thị vô hướng
- m dòng tiếp theo, mỗi dòng chứa hai số u, v tương ứng với một cạnh (u, v) của đồ thị.

Output

Các thành phần song liên thông của đồ thị

Sample Input	Sample Output
9 10 0 2 0 3 2 3 2 5 2 6 3 7 3 8 4 8 5 6 7 8	Biconnected component: 4 8 Biconnected component: 8 7 3 Biconnected component: 6 5 2 Biconnected component: 3 2 0 Biconnected component: 1



BCC.PAS ✓ Liệt kê các thành phần song liên thông

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
const int maxN = 1e6;
const int maxM = 1e6;
int n, m;

vector<int> adj[maxN]; //Các danh sách kề
int num[maxN], low[maxN];
stack<int> Stack;
int Time; //Biến đếm số thứ tự thăm

void ReadInput() //Đọc dữ liệu
{
    cin >> n >> m;
    for (int i = 0; i < m; ++i)
    {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
}

inline void Minimize(int& Target, int Value) //Target = min(Target, Value)
{
    if (Value < Target) Target = Value;
}

```

```cpp
void DFSVisit(int u) //Thuật toán DFS bắt đầu từ u
{
    num[u] = Time++; //Đánh số u theo thứ tự thăm
    low[u] = maxN; //low[u] khởi tạo bằng +∞
    for (int v: adj[u])
        if (num[v] == -1) //v chưa thăm
        {
            Stack.push(v);
            DFSVisit(v); //Thăm v
            Minimize(low[u], low[v]); //Cực tiểu hóa low[u] theo low[v]
            if (low[v] >= num[u]) //(u, v) là cung chốt
            {
                cout << "Biconnected component:\n";
                int vertex;
                do //Lấy các đỉnh khỏi ngăn xếp cho tới khi v được lấy ra
                {
                    vertex = Stack.top();
                    Stack.pop();
                    cout << vertex << ' ';
                }
                while (vertex != v);
                cout << u << '\n'; //in thêm đỉnh u là gốc nhánh DFS
            }
        }
        else //v đã thăm
            Minimize(low[u], num[v]); //Cực tiểu hóa low[u] theo num[v]
}

void Solve()
{
    fill(num, num + n, -1); //Các đỉnh đều chưa thăm, được đánh số bởi giá trị âm
    Time = 0; //Khởi tạo biến đếm số thứ tự thăm
    for (int u = 0; u < n; ++u)
        if (num[u] == -1) //u chưa thăm
        {
            int OldTime = Time;
            DFSVisit(u); //Duyệt DFS từ u
            if (Time - OldTime == 1) //u là đỉnh cô lập
                cout << "Biconnected component:\n" << u << "\n";
        }
}

int main()
{
    ReadInput();
    Solve();
}
```

Bài tập 1-1

Cho một đồ thị n đỉnh, ban đầu chưa có cạnh nào, người ta lần lượt thêm vào m cạnh vô hướng. Yêu cầu cho biết số cặp đỉnh đi sang được nhau sau mỗi bước thêm cạnh. Yêu cầu thuật toán với độ phức tạp $O\big((m+n)\alpha(n)\big)$.

Bài tập 1-2

Cho một đồ thị hình cây gồm n đỉnh, người ta thực hiện m phép duyệt, mỗi phép duyệt cho bởi cặp đỉnh (s, t) và duyệt qua tất cả các cạnh trên đường đi đơn từ s tới t. Tìm thuật toán $O(n + m)$ xác định các cạnh của cây không được duyệt qua.

Gợi ý: Với mỗi phép duyệt bổ sung thêm cạnh (s, t) vào đồ thị, với đồ thị mới tạo thành, những cầu chắc chắn là những cạnh trên cây ban đầu mà không được duyệt qua

Bài tập 1-3

Cho một bảng hình chữ nhật được chia làm lưới ô vuông đơn vị, gọi k là diện tích bảng. Mỗi ô được tô một màu. Một miền là một tập hợp các ô của bảng sao cho từ một ô của miền có thể đi sang mọi ô khác bằng các phép di chuyển qua các ô kề cạnh.

Tìm thuật toán $O(k)$ xác định miền lớn nhất sao cho các ô của miền cùng màu.

Tìm thuật toán $O(k \log k)$ xác định miền lớn nhất chỉ gồm hai màu.

Bài tập 1-4

Cho một đồ thị vô hướng G gồm n đỉnh và m cạnh, tìm thuật toán $O(n + m)$ cho biết từng mỗi đỉnh u, nếu xóa đỉnh đó cùng những cạnh liên thuộc thì đồ thị còn lại (gồm $n - 1$ đỉnh) có bao nhiêu thành phần liên thông

Bài tập 1-5

Tìm thuật toán đếm số cây khung của đồ thị (Hai cây khung gọi là khác nhau nếu chúng có ít nhất một cạnh khác nhau)

Gợi ý: Tìm đọc thêm về định lý Kirchhoff và ma trận Laplace

Bài tập 1-6

Tìm hiểu các giải pháp xác định khớp/cầu/số thành phần song liên thông với đồ thị động gồm thao tác thêm đỉnh, thêm cạnh và truy vấn khớp, cầu cũng như số thành phần song liên thông.