ĐỒ THỊ CƠ BẢN

Ngày 19 tháng 7 năm 2024

Mục lục

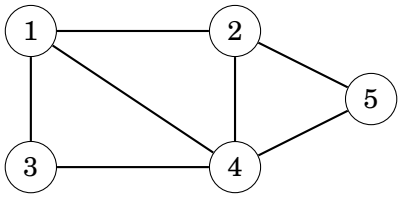
1. Lý thuyết

Đầu tiên, chúng ta sẽ tóm tắt lại một số khái niệm về đồ thị và biểu diễn đồ thị.

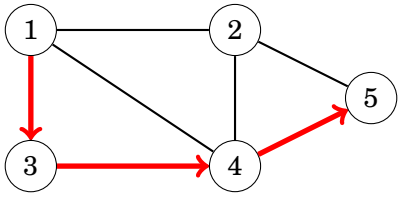
1.1 Các thuật ngữ đồ thị

Một **đồ thị** gồm **các đỉnh** và**các cạnh**. Trong các bài toán, thường sẽ là n đỉnh, m cạnh. Các đỉnh được đánh số lần lượt là $1, 2, \ldots, n$.

Ví dụ, đồ thị dưới đây gồm 5 đỉnh, 7 cạnh:



Một **đường đi** từ đỉnh a đến đỉnh b thông qua một số cạnh trên đồ thị. Độ dài của đường đi là số cạnh trên đường đi đó. Ví dụ, đồ thị dưới đây có một đường đi là từ đỉnh 1 đến đỉnh 5 là: $1 \to 3 \to 4 \to 5$ với độ dài là 3.

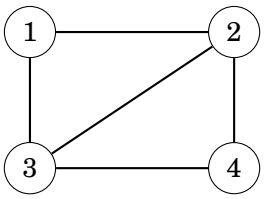


Một đường đi gọi là là **chu trình** nếu đỉnh đầu tiên và đỉnh cuối cùng trùng nhau. Ví dụ, đồ thị trên có một chu trình: $1 \to 3 \to 4 \to 1$. Một đường đi là **đường đi đơn** nếu mỗi nút xuất hiện nhiều nhất một lần trong đường đi.

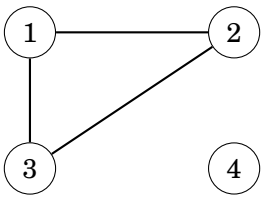
Tính liên thông

Một đồ thị là **liên thông** nếu luôn có một đường đi giữa hai đỉnh bất kỳ trong đồ thị.

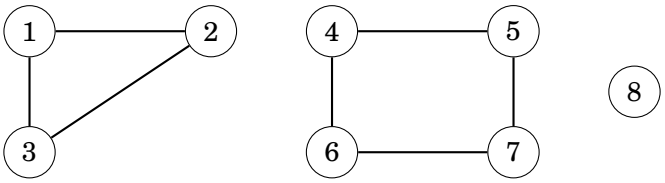
Ví dụ: đồ thị dưới đây là liên thông:



Đồ thị dưới đây không liên thông vì không có đỉnh nào được kết nối với đỉnh 4 của đồ thị:

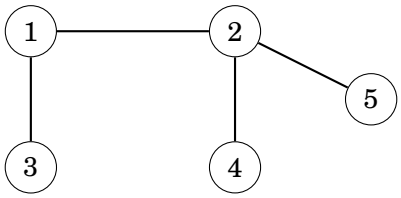


Mỗi phần liên thông trong đồ thị được gọi là **thành phần liên thông**. Ví dụ, đồ thị dưới đây gồm ba thành phần liên thông: $\{1, 2, 3\}$, $\{4, 5, 6, 7\}$ và $\{8\}$.



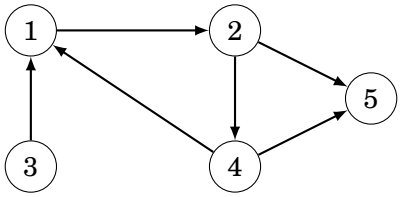
Một **cây đồ thị** là một đồ thị liên thông gồm n đỉnh, $n-1$ cạnh. Giữa hai đỉnh bất kỳ của cây chỉ có một đường đi duy nhất.

Ví dụ, đồ thị dưới đây là một cây:



Cạnh có hướng

Một đồ thị gọi là **có hướng** nếu các cạnh của nó chỉ được duyệt theo một hướng chỉ định. Ví dụ, đồ thị dưới đây là một đồ thị có hướng:

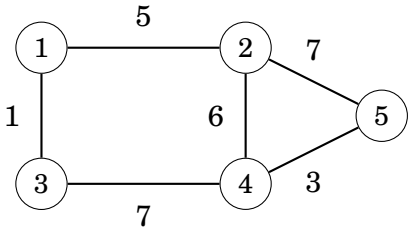


Đồ thị ở trên tồn tại đường đi từ đỉnh 3 tới đỉnh 5: $3 \to 1 \to 2 \to 5$ nhưng không có đường đi nào từ đỉnh 5 tới đỉnh 3.

Cạnh có trọng số

Trong một đồ thị **có trọng số** mỗi cạnh sẽ được gán một giá trị gọi là **trọng số**.Các trọng số thường được hiểu là độ dài cạnh.

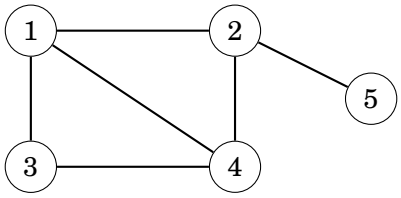
Ví dụ: đồ thị dưới đây là đồ thị có trọng số:



Độ dài của một đường đi trong đồ thị có trọng số là tổng trọng số tất cả các cạnh trên đường đi đó. Ví dụ với đồ thị ở trên, đường đi $1 \to 2 \to 5$ có độ dài là 12, còn độ dài của đường đi $1 \to 3 \to 4 \to 5$ là 11, đây cũng là đường đi **ngắn nhất** từ đỉnh 1 đến đỉnh 5.

Đỉnh kề và bậc của đỉnh

Hai đỉnh gọi là **kề nhau** nếu tồn tại một cạnh nối trực tiếp giữa chúng. **Bậc** của một đỉnh là số lượng đỉnh kề với đỉnh đó. Ví dụ, ở đồ thị dưới đây, đỉnh 2 có 3 đỉnh kề là 1, 4 và 5 do đó nó có bậc là 3.

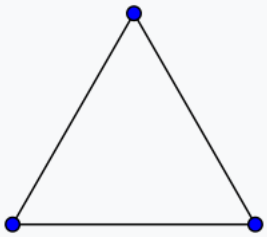
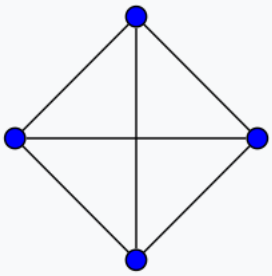
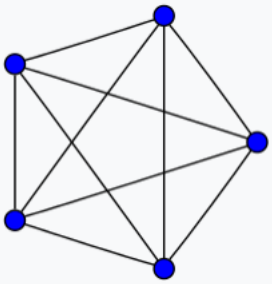
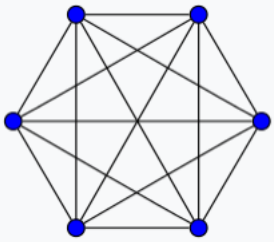
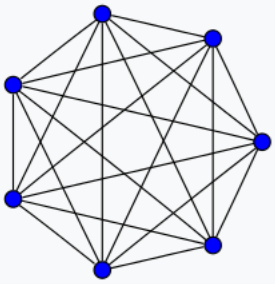
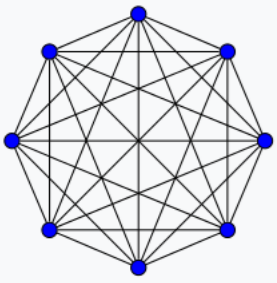
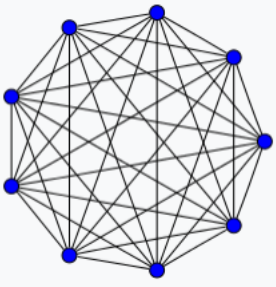
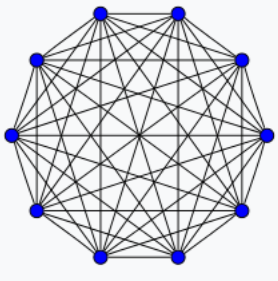
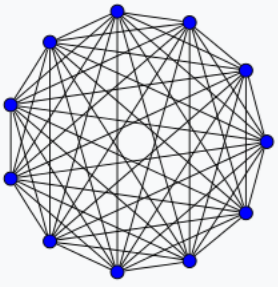
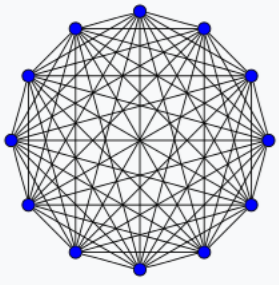


Tổng bậc của tất cả các đỉnh trong đồ thị luôn là $2 \times m$, với m là số cạnh của đồ thị, bởi vì mỗi cạnh tăng bậc hai đỉnh của nó mỗi nút thêm một. Do đó, tổng bậc của đồ thị luôn chẵn.

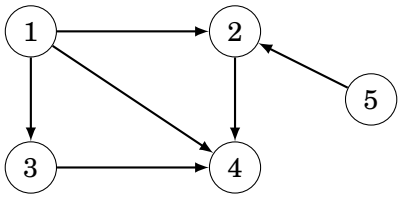
Một đồ thị gọi là **chính quy** nếu các đỉnh đều có cùng bậc. Một đồ thị chính quy với đỉnh có bậc k được gọi là **đồ thị chính quy bậc** k. Đồ thị chính quy bậc 0 gồm tác đỉnh không có cạnh chung. Đồ thị chính quy bậc 1 gồm tập các cạnh không nối với nhau, đồ thị chính quy bậc 2 gồm các chu trình không nối với nhau.

Một đồ thị gọi là **đầy đủ** mỗi đỉnh đều có bậc là $n-1$. Trong đồ thị đầy đủ, luôn có cạnh nối trực tiếp hai đỉnh bất kỳ. Đồ thị đầy đủ là đồ thị đơn có nhiều cạnh nhất và là đồ thị chính quy bậc $n-1$.

Dưới đây là hình minh hoạ một số đồ thị đầy đủ với số đỉnh từ 1 đến 12:

$K_1 : 0$	$K_2 : 1$	$K_3 : 3$	$K_4 : 6$
			
$K_5 : 10$	$K_6 : 15$	$K_7 : 21$	$K_8 : 28$
			
$K_9 : 36$	$K_{10} : 45$	$K_{11} : 55$	$K_{12} : 66$
			

Trong đồ thị có hướng, **bậc trong** của một đỉnh là số cạnh kết thúc tại đỉnh đó và **bậc ngoài** của một đỉnh là số cạnh bắt đầu tại đỉnh đó. Ví dụ, trong biểu đồ sau, bậc trong của nút 2 là 2 và bậc ngoài của nút 2 là 1.

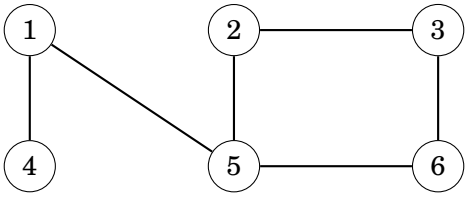


Tô màu

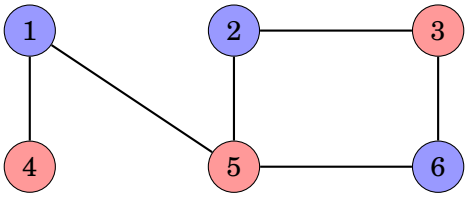
Khi **tô màu** cho một đồ thị,mỗi đỉnh sẽ được gán cho một màu sao cho không có hai đỉnh kề nhau nào được tô cùng một màu.

Một đồ thị gọi là **đồ thị hai phía** nếu có thể tô màu nó bằng hai màu. Như vậy, một đồ thị hai phía sẽ không có chu trình với số cạnh lẻ.

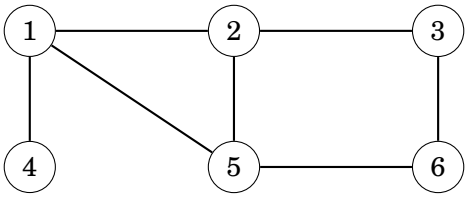
Ví dụ, đồ thị



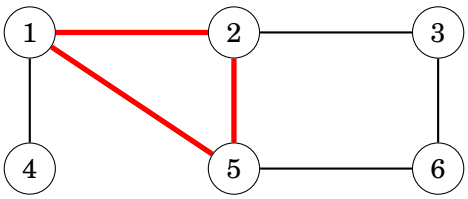
là đồ thị hai phía vì có thể tô nó bằng hai màu như sau:



Tuy nhiên, đồ thị

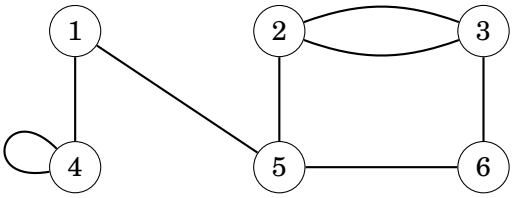


không phải là đồ thị hai phía vì không thể tô màu cho chu trình ba đỉnh bằng cách sử dụng hai màu:



Đồ thị đơn giản

Một đồ thị gọi là **đơn giản** nếu không có cạnh nào xuất phát và kết thúc tại cùng một đỉnh và không có nhiều cạnh nối giữa hai đỉnh. Trong các bài toán, ta thường giả sử đồ thị là đơn giản. Ví dụ đồ thị dưới đây không phải là đồ thị đơn giản:



1.2 Biểu diễn đồ thị

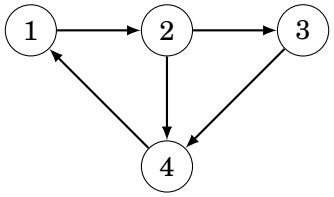
Có một số cách để biểu diễn đồ thị trong thuật toán. Việc lựa chọn cấu trúc dữ liệu phụ thuộc vào kích thước của đồ thị và cách xử lý đồ thị của thuật toán.

Biểu diễn bằng danh sách kề

Trong biểu diễn danh sách kề, mỗi nút x trong đồ thị được gán một **danh sách kề** bao gồm các nút có cạnh nối từ đỉnh x. Danh sách kề là cách phổ biến nhất để biểu diễn đồ thị và hầu hết các thuật toán có thể được triển khai hiệu quả bằng cách sử dụng nó. Một cách thuận tiện để lưu trữ danh sách kề là khai báo một mảng vectơ như sau:

```
vector<int> adj[N];
```

Với hằng số N là số đỉnh tối đa của đồ thị. Ví dụ, đồ thị



có thể được lưu trữ như sau:

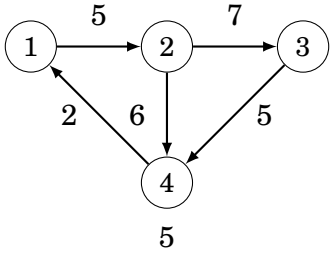
```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

Nếu đồ thị vô hướng thì vẫn lưu trữ tương tự có điều phải đưa vào danh sách kề của cả hai đỉnh (lúc này có thể coi như là hai hướng).

Với đồ thị có trọng số thì có thể lưu như sau:

```
vector<pair<int,int>> adj[N];
```

Khi đó, danh sách các cạnh kề với đỉnh a sẽ chứa các cặp (b,w) ứng với một cạnh nối từ a đến b có trọng số là w. Ví dụ, đồ thị



Có thể lưu trữ như sau:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

Với việc sử dụng cách biểu diễn bằng danh sách kề, ta có thể tìm các đỉnh hiệu quả trong quá trình duyệt đường đi trên đồ thị. Ví dụ, vòng lặp sau sẽ duyệt qua tất cả các đỉnh kề với đỉnh s.

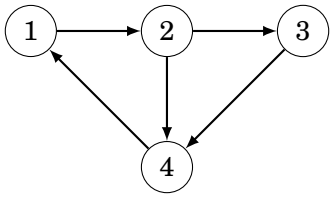
```
for (auto u : adj[s]) {
    // xu ly dinh u ke voi s
}
```

Biểu diễn đồ thị bằng ma trận kề

ma trận kề là một mảng hai chiều cho biết đồ thị chứa các cạnh nào. Cách biểu diễn này cho phép tìm cạnh nối giữa hai đỉnh một cách hiệu quả. Ma trận có thể được lưu trữ dưới dạng một mảng

```
int adj[N][N];
```

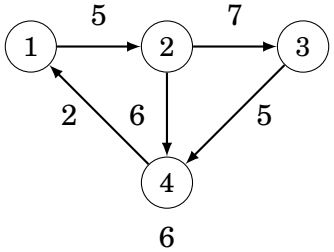
trong đó mỗi giá trị `adj[a][b]` cho biết đồ thị có tồn tại cạnh nối hai đỉnh a và b hay không. Nếu tồn tại thì `adj[a][b]=1`, còn nếu không thì `adj[a][b]=0`. Ví dụ, đồ thị



có thể được biểu diễn như sau:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Nếu đồ thị có trọng số, giá trị của `adj[a][b]` được gán bằng trọng số của cạnh nối hai đỉnh a và b. Ví dụ, đồ thị



tương ứng với ma trận sau:

	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

Nhược điểm của phương pháp biểu diễn đồ thị bằng ma trận kề là ma trận gồm 2^n phần tử trong đó hầu hết đều có giá trị bằng 0. Do đó, cách biểu diễn này thường không được sử dụng trong các đồ thị có kích thước lớn.

Biểu diễn đồ thị bằng danh sách cạnh

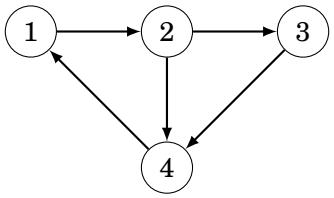
Một **danh sách cạnh** chứa tất cả các cạnh của đồ thị. Cách biểu diễn này thuận tiện trong trường hợp thuật toán chỉ xử lý các cạnh của đồ thị mà không cần xác định cạnh bắt đầu từ một nút nào đó.

Danh sách kề có thể lưu trữ dưới dạng một `vector`

```
vector<pair<int,int>> edges;
```

Với mỗi cặp (a,b) xác định một cạnh nối từ đỉnh a đến đỉnh b.

Theo đó, đồ thị



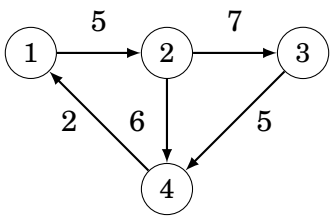
có thể biểu diễn như sau:

```
edges.push_back({1,2});
edges.push_back({2,3});
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

Nếu đồ thị có trọng số thì có thể được lưu như sau:

```
vector<tuple<int,int,int>> edges;
```

Mỗi phần tử trong danh sách có dạng (a,b,w), mô tả một cạnh nối từ đỉnh a đến đỉnh b có trọng số là w. Ví dụ, đồ thị



có thể biểu diễn như sau [1]:

[1] trong một số trình biên dịch cũ, hàm `make_tuple` phải được sử dụng thay vì dùng cặp ngoặc nhọn (ví dụ dùng `make_tuple(1,2,5)` thay vì dùng `{1,2,5}`).

```
edges.push_back({1,2,5});
edges.push_back({2,3,7});
edges.push_back({2,4,6});
edges.push_back({3,4,5});
edges.push_back({4,1,2});
```

2. Duyệt đồ thị

Tiếp theo, ta sẽ thảo luận về hai thuật toán đồ thị cơ bản: tìm kiếm theo chiều sâu và tìm kiếm theo chiều rộng. Cả hai thuật toán đều bắt đầu từ một đỉnh trong đồ thị và chúng truy cập tất cả các đỉnh có thể đến được từ đỉnh đó. Sự khác biệt trong hai thuật toán này là thứ tự chúng truy cập các đỉnh.

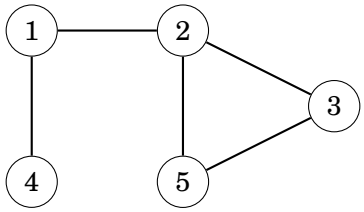
2.1 Tìm kiếm theo chiều sâu - DFS

Tìm kiếm theo chiều sâu DFS là một kỹ thuật duyệt đồ thị đơn giản. Thuật toán bắt từ một đỉnh và duyệt đến tất cả các đỉnh khác có thể truy cập được từ đỉnh này bằng cách sử dụng các cạnh trong đồ thị.

DFS luôn duyệt theo một đường đi duy nhất trong đồ thị miễn là nó tìm thấy các đỉnh mới. Sau đó, nó quay trở lại các đỉnh trước đó và bắt đầu duyệt tiếp các phần khác của đồ thị. Thuật toán theo dõi các đỉnh đã truy cập để nó chỉ xử lý mỗi đỉnh một lần.

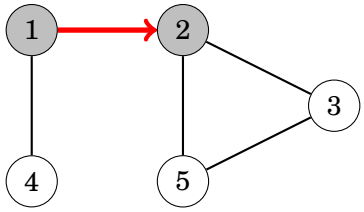
Ví dụ

Hãy quan sát cách mà **DFS** hoạt động trên đồ thị sau:

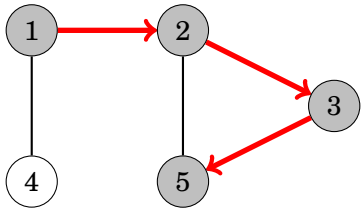


Ta có thể bắt đầu duyệt từ bất kỳ đỉnh nào của đồ thị; giả sử bắt đầu tại đỉnh 1.

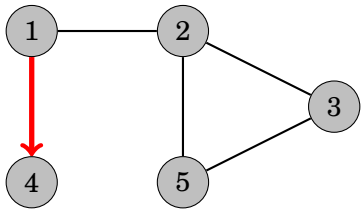
Đầu tiên sẽ duyệt đến đỉnh 2:



Sau đó duyệt đến đỉnh 3 rồi đến đỉnh 5:



Các đỉnh kề của 5 là 2 và 3, nhưng vì cả hai đỉnh này đều đã được duyệt nên lúc này thuật toán sẽ quay lại các đỉnh trước đó. Ngoài ra, các đỉnh kề của 3 và 2 cũng đều duyệt rồi nên tiếp theo sẽ đi từ đỉnh 1 sang đỉnh 4.



Sau đó, quá trình tìm kiếm kết thúc vì tất cả các đỉnh đã được duyệt.

Độ phức tạp về thời gian của tìm kiếm theo chiều sâu là $O(n+m)$ trong đó n là số đỉnh còn m là số cạnh, bởi vì thuật toán xử lý mỗi đỉnh và mỗi cạnh cạnh một lần.

Cài đặt thuật toán

Tìm kiếm theo chiều sâu có thể cài đặt thuận tiện bằng cách sử dụng hàm đệ quy. Hàm `dfs` sau bắt đầu tìm kiếm theo chiều sâu tại một đỉnh nhất định. Hàm này giả định rằng đồ thị được lưu trữ dưới dạng danh sách kề

```
vector<int> adj[N];
```

và đồng thời duy trì một mảng

```
bool visited[N];
```

để theo dõi xem các đỉnh đã được truy cập hay chưa. Ban đầu, tất cả các giá trị của mảng này bằng `false`, và khi duyệt đến đỉnh s, thì gán `visited`[s] bằng `true`.

Hàm này có thể triển khai như sau:

```
void dfs(int s) {
    if (visited[s]) return;
    //neu dinh s da tham thi khong tham nua
    visited[s] = true; //danh dau tham dinh s
    // xu ly dinh s
    for (auto u: adj[s]) { //duyet tat ca cac dinh ke voi s
        dfs(u); //goi de quy tham dinh u
    }
}
```

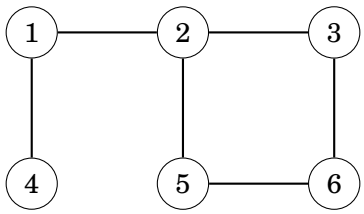
2.2 Tìm kiếm theo chiều rộng

Tìm kiếm theo chiều rộng duyệt các đỉnh theo thứ tự tăng dần của khoảng cách tới đỉnh xuất phát. Do đó, ta có thể tính toán khoảng cách từ đỉnh xuất phát đến tất cả các đỉnh khác bằng cách sử dụng **BFS**. Tuy nhiên, **BFS** khó cài đặt hơn so với **DFS**.

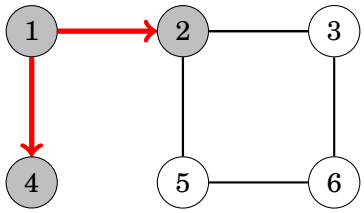
Tìm kiếm theo chiều rộng đi qua các đỉnh từ cấp này đến cấp khác. Đầu tiên duyệt các đỉnh có khoảng cách từ đỉnh xuất phát là 1, sau đó là các đỉnh có khoảng cách là 2, v.v. Quá trình này tiếp tục cho đến khi tất cả các đỉnh đều được duyệt.

Ví dụ

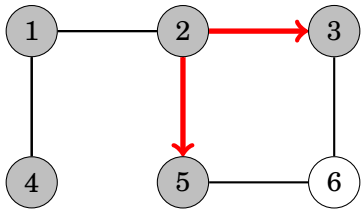
Xét cách mà **BFS** hoạt động trên đồ thị sau:



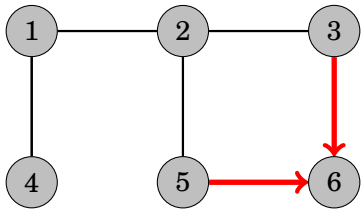
Giả sử tìm kiếm bắt đầu từ đỉnh 1. Đầu tiên, ta sẽ xử lý tất cả các đỉnh có thể đến được từ đỉnh 1 mà chỉ duyệt qua đúng 1 cạnh:



Sau đó, ta xử lý đến đỉnh 3 và đỉnh 5:



Cuối cùng, thăm nốt đỉnh 6:



Bây giờ ta tính toán khoảng cách từ đỉnh xuất phát đến tất cả các đỉnh của đồ thị:

Đỉnh	Khoảng cách
1	0
2	1
3	2
4	1
5	2
6	3

Giống như tìm kiếm theo chiều sâu, độ phức tạp của thuật toán **BFS** cũng là $O(n+m)$, với n là số đỉnh còn m là số cạnh của đồ thị.

Cài đặt thuật toán

Tìm kiếm theo chiều rộng khó cài đặt hơn so với tìm kiếm theo chiều sâu, bởi vì thuật toán truy cập các đỉnh thuộc các phần khác nhau của đồ thị. Mã cài đặt điển hình dựa trên cấu trúc hàng đợi có chứa các đỉnh. Tại mỗi bước, đỉnh tiếp theo trong hàng đợi sẽ được xử lý.

Dưới đây là mã cài đặt giả sử đồ thị được lưu trữ dưới dạng danh sách kề và duy trì các thông tin sau:

```
queue<int> q;
bool visited[N];
int distance[N];
```

Hàng đợi `q` chứa các đỉnh được xử lý theo thứ tự tăng dần về khoảng cách so với đỉnh xuất phát. Các đỉnh mới luôn được thêm vào cuối hàng đợi và đỉnh ở đầu hàng đợi là đỉnh được xử lý tiếp theo. Mảng `visited` cho biết một đỉnh đã được truy cập hay chưa còn mảng `distance` sẽ chứa khoảng cách từ đỉnh xuất phát đến tất cả các đỉnh của đồ thị.

Việc tìm kiếm xuất phát từ đỉnh x có thể được cài đặt như sau:

```
visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) { //xu ly tat ca cac dinh trong hang doi
    int s = q.front(); q.pop(); //lay dinh s o dau hang doi
    // xu ly dinh s
    for (auto u : adj[s]) { //duyet tat ca cac dinh ke voi dinh s
        if (visited[u]) continue;
        //neu dinh dang xet da duoc truy cap thi bo qua
        visited[u] = true;
        distance[u] = distance[s]+1;
        q.push(u); //dua dinh ke voi s nay vao cuoi hang doi
    }
}
```

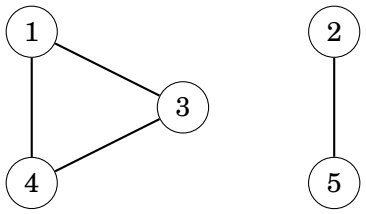
2.3 Ứng dụng của các thuật toán duyệt đồ thị

Sử dụng thuật toán duyệt đồ thị, chúng ta có thể kiểm tra nhiều tính chất của đồ thị. Thông thường, cả tìm kiếm theo chiều sâu và tìm kiếm theo chiều rộng đều có thể được sử dụng, nhưng trong thực tế, tìm kiếm theo chiều sâu là lựa chọn tốt hơn vì nó dễ cài đặt hơn. Trong các ứng dụng sau đây, chúng ta sẽ giả sử rằng đồ thị là vô hướng.

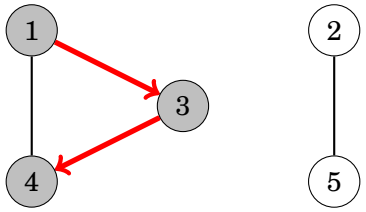
Kiểm tra tính liên thông

Một đồ thị là liên thông nếu luôn có một đường đi giữa hai đỉnh bất kỳ của đồ thị. Do đó, chúng ta có thể kiểm tra xem một đồ thị có liên thông hay không bằng cách bắt đầu tại một đỉnh tùy ý và tìm hiểu xem liệu có thể đến được tất cả các đỉnh khác hay không.

Ví dụ, trong đồ thị



tìm kiếm theo chiều sâu xuất phát từ đỉnh 1 sẽ thăm các đỉnh như sau:

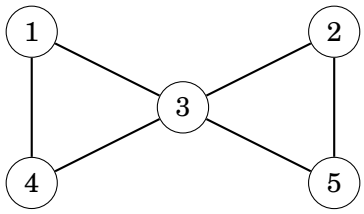


Vì sau khi tìm kiếm không truy cập hết tất cả các đỉnh, ta có thể kết luận rằng đồ thị không liên thông. Theo cách tương tự, chúng ta cũng có thể tìm thấy tất cả các thành phần liên thông của đồ thị bằng cách **DFS** qua tất cả các đỉnh chưa thuộc bất kỳ một thành phần liên thông nào.

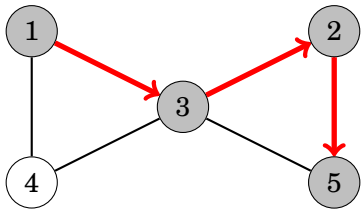
Tìm kiếm chu trình

Một đồ thị chứa một chu trình nếu trong quá trình duyệt qua đồ thị, ta tìm thấy một đỉnh có đỉnh kề (không phải là đỉnh vừa duyệt trước đó trong đường đi hiện tại) đã được thăm.

Ví dụ, đồ thị



có hai chu trình và ta có thể tìm được một chu trình trong đó như sau:



Sau khi di chuyển từ đỉnh 2 đến đỉnh 5, ta thấy rằng đỉnh kề 3 của đỉnh 5 đã được thăm. Như vậy, đồ thị chứa một chu trình đi qua đỉnh 3, chẳng hạn $3 \to 2 \to 5 \to 3$.

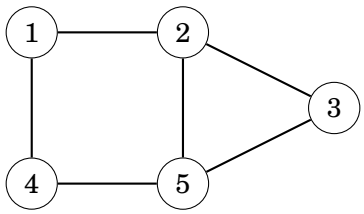
Một cách khác để biết liệu một đồ thị có chứa chu trình hay không là chỉ cần tính số đỉnh và cạnh trong mỗi thành phần liên thông. Nếu một thành phần liên thông có c đỉnh và và không có chu trình, thì nó phải chứa chính xác $c-1$ cạnh (vì vậy nó phải là một cây). Nếu có c hoặc nhiều cạnh hơn, thành phần đó chắc chắn chứa một chu trình.

Kiểm tra đồ thị hai phía

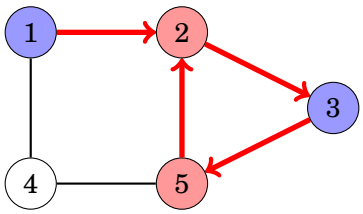
Một đồ thị là đồ thị hai phía nếu các đỉnh của nó có thể được tô bằng hai màu sao cho không có hai đỉnh liền kề nào có cùng màu. Có thể dễ dàng kiểm tra xem một đồ thị có phải là hai phía hay không bằng cách sử dụng thuật toán duyệt đồ thị.

Ý tưởng là tô màu đỉnh bắt đầu màu xanh, tất cả nút kề nó màu đỏ, tất cả nút kề tiếp theo màu xanh, v.v. Nếu trong quá trình tìm kiếm mà nhận thấy hai đỉnh liền kề cùng màu, có nghĩa là đồ thị không phải là đồ thị hai phía. Ngược lại, đồ thị là hai phía.

Ví dụ, đồ thị



Không phải là đồ thị hai phía vì khi xuất phát từ đỉnh 1, quá trình tô màu như sau:



Có thể thấy rằng màu của cả hai đỉnh 2 và 5 đều là màu đỏ, và hai đỉnh này là liền kề. Do đó, đồ thị không phải là đồ thị hai phía.

Thuật toán này luôn hoạt động, bởi vì khi chỉ có hai màu, màu của đỉnh xuất phát trong một thành phần liên thông sẽ xác định màu của tất cả các nút khác trong thành phần liên thông. Đỉnh xuất phát màu đỏ hay xanh không làm thay đổi kết quả.

Lưu ý rằng trong trường hợp chung, rất khó để biết rằng liệu các đỉnh trong đồ thị có thể được tô màu bằng cách sử dụng k màu sao cho không có đỉnh liền kề nào có cùng màu hay không. Ngay cả khi $k = 3$, không có thuật toán hiệu quả nào được làm được điều đó nhưng đây là vấn đề **NP-hard**.

3. Ví dụ minh họa

3.1 Biểu diễn đồ thị [GRAPH1]

Có hai cách tiêu chuẩn để biểu diễn một đồ thị, trong đó $G = (V, E)$ là tập hợp các đỉnh và E là tập hợp các cạnh; biểu diễn danh sách kề và biểu diễn ma trận kề.

Biểu diễn danh sách kề bao gồm một mảng $Adj[|V|]$ các danh sách, mỗi danh sách cho mỗi đỉnh trong $|V|$. Đối với mỗi u, danh sách kề $Adj[u]$ chứa tất cả các đỉnh v sao cho có một cạnh $(u, v) \in E$. Đó là, $Adj[u]$ bao gồm tất cả các đỉnh kề với u trong G.

Biểu diễn ma trận kề bao gồm ma trận $|V| \times |V|$ sao cho $a_{ij} = 1$ nếu $(i, j) \in E$, $a_{ij} = 0$ nếu không.

Viết một chương trình đọc một đồ thị có hướng G được biểu diễn bởi danh sách kề, và in ra biểu diễn ma trận kề của nó. G bao gồm n $(= |V|)$ đỉnh được xác định bằng chỉ số $1, 2, \ldots, n$ tương ứng.

Dữ liệu

- Dòng 1: chứa một số nguyên n;
- Tiếp theo là n dòng, dòng thứ i mô tả một danh sách các đỉnh kề với đỉnh i có dạng: $u\ k\ v_1\ v_2\ \ldots\ v_k$ biểu thị danh sách gồm k đỉnh kề của đỉnh có chỉ số là u.

Kết quả

- Ghi ra biểu diễn ma trận kề của đồ thị đã cho.

Ví dụ

input	output
4 1 2 2 4 2 1 4 3 0 4 1 3	0 1 0 1 0 0 0 1 0 0 0 0 0 0 1 0

Hướng dẫn giải - biểu diễn đồ thị [GRAPH1]

Đây là một bài toán ví dụ đơn giản nhất về biểu diễn đồ thị. Bài toán này được thiết kế dành cho học sinh mới học về biểu diễn đồ thị.

Mã tham khảo C++ - sử dụng DFS

```cpp
#include <bits/stdc++.h>
using namespace std;

// Khai bao ma tran ke voi kich thuoc toi da 101 x 101
int adj[101][101];

int main()
{
   int n;     // Khai bao so luong dinh
   cin >> n; // Nhap so luong dinh

   // Duyet qua tung dinh
   for (int i = 0; i < n; i++)
   {
      int m, r;       // Khai bao dinh hien tai va so luong dinh ke
      cin >> m >> r; // Nhap dinh hien tai va so luong dinh ke

      // Duyet qua tung dinh ke cua dinh hien tai
      for (int j = 0; j < r; j++)
      {
         int q;          // Khai bao dinh ke
         cin >> q;       // Nhap dinh ke
         adj[m][q] = 1; // Danh dau co canh tu dinh m den dinh q
      }
   }

   // In ra ma tran ke
   for (int i = 1; i <= n; i++)
   {
      for (int z = 1; z <= n; z++)
      cout << adj[i][z] << (z == n ? '\n' : ' '); // In gia tri va them dau cach
   }
}
```

3.2 Kiểm tra cây [TREE]

Cho đồ thị vô hướng không trọng số gồm n đỉnh, m cạnh.

Hãy cho biết đồ thị đã cho có phải là một cây hay không.

Dữ liệu

- Dòng 1: ghi hai số nguyên n, m ($0 \le n \le 10^5, 0 \le m \le 2 \times 10^5$);
- Tiếp theo mà m dòng, mỗi dòng ghi hai số nguyên u, v ($1 \le u, v \le n$) mô tả một cạnh nối giữa hai đỉnh u, v.

Kết quả

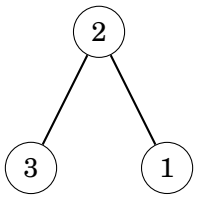
- Ghi "YES" nếu đồ thị đã cho là một cây. Ngược lại ghi "NO".

Ví dụ

input	output
3 2 1 2 2 3	YES

Giải thích ví dụ

- Hình mô tả đồ thị trong ví dụ:



Hướng dẫn giải - Kiểm tra cây

- Theo định nghĩa, một đồ thị là một cây nếu nó thoả mãn các điều kiện sau:
  1. Số đỉnh là n thì số cạnh là $n-1$;
  2. Nếu đã thoả mãn điều kiện ở trên thì cần thêm yêu cầu nữa là trong cây không có chu trình (nếu có chu trình thì sẽ có ít nhất một đỉnh lẻ dẫn đến đồ thị không liên thông)
- Điều kiện thứ nhất dễ dàng kiểm tra bằng lệnh `if`;
- Điều kiện thứ hai có nhiều cách để kiểm tra như `dfs`, `bfs`, `topo sort`, ... Dưới đây là mã triển khai sử dụng `dfs`, ý tưởng là duyệt một lần `dfs`, nếu sau khi duyệt mà còn đỉnh chưa được thăm nghĩa là trong đồ thị có chu trình.

Mã tham khảo C++

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5 + 1; // Dinh nghia so luong dinh toi da
int n, m;              // n la so dinh, m la so canh
vector<int> adj[N];    // Danh sach ke de luu cac canh
bool vis[N];           // Mang danh dau cac dinh da duoc tham

// Ham DFS de duyet do thi
void dfs(int u)
{
   if (vis[u])
   return;    // Neu dinh u da duoc tham, thoat ra
   vis[u] = true; // Danh dau dinh u da duoc tham
   for (auto v : adj[u])
   {           // Duyet tat ca cac dinh ke cua u
      dfs(v); // Goi de quy de tham dinh v
   }
}

int main()
{
   cin.tie(0)->sync_with_stdio(0); // Toi uu hoa nhap/xuat
   cin >> n >> m;                  // Nhap so dinh va so canh

   // Kiem tra dieu kien so canh phai bang n-1
   if (m != n - 1)
   {
      cout << "NO\n"; // Neu khong thoa man, in NO va ket thuc
      return 0;
   }

   // Nhap cac canh va tao danh sach ke
   for (int i = 0; i < m; ++i)
   {
      int u, v;
```

```
37        cin >> u >> v;
38        adj[u].push_back(v); // Them dinh v vao danh sach ke cua dinh u
39        adj[v].push_back(u); // Them dinh u vao danh sach ke cua dinh v
40    }
41
42    // Thuc hien duyet do thi bat dau tu dinh 1
43    dfs(1);
44
45    // Kiem tra tat ca cac dinh co duoc tham hay khong
46    for (int i = 1; i <= n; ++i)
47    {
48        if (!vis[i])
49        {                    // Neu co dinh nao chua duoc tham
50            cout << "NO\n"; // In NO va ket thuc
51            return 0;
52        }
53    }
54
55    // Neu tat ca cac dinh deu duoc tham, in YES
56    cout << "YES\n";
57    return 0;
58 }
```

3.3 Du lịch [TOUR]

Quốc gia X có N thành phố được đánh số từ 1 đến N và M con đường được đánh số từ 1 đến M. Con đường thứ i cho phép đi từ thành phố A_i đến thành phố B_i, nhưng không cho phép đi từ thành phố B_i đến thành phố A_i.

Tèo đang lên kế hoạch cho chuyến du lịch trên quốc gia này, bắt đầu từ một thành phố nào đó, di chuyển qua không hoặc nhiều con đường, và kết thúc tại một thành phố nào đó. Hỏi có bao nhiêu cặp thành phố có thể là điểm bắt đầu và điểm kết thúc của chuyến du lịch của Tèo? Các cặp thành phố có thứ tự khác nhau cũng được coi là phân biệt.

Dữ liệu

- Dòng 1: ghi hai số nguyên N và M ($2 \le N \le 2000, 0 \le M \le \min(2000, N(N-1))$)
- Tiếp theo là M dòng, mỗi dòng ghi hai số nguyên A_i, B_i ($1 \le A_i, B_i \le N; A_i \ne B_i$) các cặp (A_i, B_i) đôi một phân biệt.

Kết quả

- Ghi một số nguyên duy nhất là số cặp thành phố đếm được.

Ví dụ

input	output
3 3 1 2 2 3 3 2	7

Giải thích ví dụ

- Chúng ta có bảy cặp thành phố có thể là điểm bắt đầu và điểm kết thúc: (1,1), (1,2), (1,3), (2,2), (2,3), (3,2), (3,3).

Hướng dẫn giải - du lịch [TOUR]

- Thử bắt đầu từ thành phố i ($\forall i = 1 \ldots N$). Duyệt DFS để đếm xem có thể kết thúc được tại bao nhiêu thành phố.
- Độ phức tạp: $O(N(N+M))$

Mã tham khảo C++

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_N = 2000;

vector<vector<int>> G; // Danh sach ke cua do thi
bool vis[MAX_N];       // Mang danh dau cac dinh da duoc tham

void dfs(int u)
{
   if (vis[u])
   return;          // Neu dinh u da duoc tham, thoat ra
   vis[u] = true;       // Danh dau dinh u da duoc tham
   for (auto v : G[u]) // Duyet tat ca cac dinh ke cua u
   dfs(v);          // Goi de quy de tham dinh v
}

int main()
{
   int N, M;
   cin >> N >> M;
   G.resize(N); // Khoi tao danh sach ke voi N dinh
   for (int i = 0; i < M; i++)
   {
      int a, b;
      cin >> a >> b;
      G[a - 1].push_back(b - 1); // Them canh tu a den b (chuyen ve chi so 0)
   }

   int ans = 0; // Bien dem so cap thanh pho co the di tu thanh pho i den j
   for (int i = 0; i < N; i++)
   {
      fill(vis, vis + N, 0); // Dat lai mang danh dau cho moi lan duyet moi
      dfs(i);                // Goi DFS tu dinh i
      for (int j = 0; j < N; j++)
      if (vis[j])
      ans++; // Neu dinh j duoc tham, tang bien dem
   }

   cout << ans; // In ra ket qua
   return 0;
}
```

3.4 Đếm đường đi đơn [PATH]

Cho đồ thị vô hướng không trọng số gồm n đỉnh, m cạnh. Các đỉnh và các cạnh đều được đánh số từ 1. Mỗi đỉnh của đồ thị có bậc không lớn hơn 10.

Gọi K là số lượng đường đi đơn xuất phát từ đỉnh 1 (đường đi không có cạnh lặp lại). Hãy in ra giá trị: $min(K, 10^6)$.

Dữ liệu

- Dòng 1: ghi hai số nguyên n và m ($1 \le n \le 2 \times 10^5, 0 \le m \le \min(2 \times 10^5, \frac{n(n-1)}{2})$ tương ứng với số đỉnh và số cạnh của đồ thị.

- Tiếp theo là m dòng, dòng thứ i ghi hai số nguyên u_i, v_i $(1 \le i \le n)$ mô tả cạnh thứ i nối hai đỉnh u_i và v_i.

Kết quả

- Ghi một số nguyên duy nhất là kết quả tìm được.

Ví dụ

input	output
4 2 1 2 2 3	3

Hướng dẫn giải - đếm đường đi đơn [PATH]

- Thử bắt đầu từ thành phố i $(\forall i = 1 \ldots N)$. Duyệt DFS đề đếm xem có thể kết thúc được tại bao nhiêu thành phố.
- Độ phức tạp: $O(N(N+M))$

Mã tham khảo C++

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_N = 2000;

vector<vector<int>> G;
bool vis[MAX_N];
void dfs(int u)
{
   if (vis[u])
   return;
   vis[u] = true;
   for (auto v : G[u])
   dfs(v);
}

int main()
{
   int N, M;
   cin >> N >> M;
   G.resize(N);
   for (int i = 0; i < M; i++)
   {
      int a, b;
      cin >> a >> b;
      G[a - 1].push_back(b - 1);
   }

   int ans = 0;
   for (int i = 0; i < N; i++)
   {
      fill(vis, vis + N, 0);
      dfs(i);
      for (int j = 0; j < N; j++)
      if (vis[j])
      ans++;
   }

   cout << ans;
}
```

3.5 Đếm thành phần liên thông [COMPONENT]

Cho đồ thị vô hướng không trọng số gồm n đỉnh, m cạnh. Các đỉnh và các cạnh đều được đánh số từ 1.

Hãy đếm xem trong đồ thị có bao nhiêu thành phần liên thông.

Dữ liệu

- Dòng 1: ghi hai số nguyên n và m ($1 \le n \le 100, 0 \le m \le \frac{n(n-1)}{2}$) tương ứng với số đỉnh và số cạnh của đồ thị.
- Tiếp theo là m dòng, dòng thứ i ghi hai số nguyên u_i, v_i ($1 \le i \le n$) mô tả cạnh thứ i nối hai đỉnh u_i và v_i.

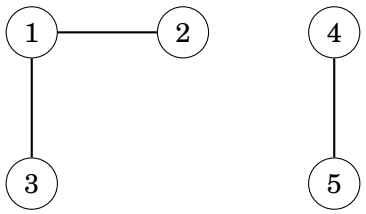
Kết quả

- Ghi một số nguyên duy nhất là số lượng thành phần liên thông của đồ thị.

Ví dụ

input	output
5 3 1 2 1 3 4 5	2
5 0	5

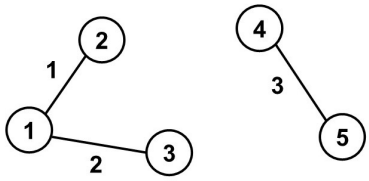
Giải thích ví dụ 1



Hướng dẫn giải - đếm thành phần liên thông [COMPONENT]

Khi một đỉnh v được chọn, tập hợp các đỉnh chứa trong thành phần liên thông chứa v có thể được tìm thấy bằng thuật toán tìm kiếm theo chiều sâu (DFS) hoặc Tìm kiếm theo chiều rộng (BFS). Do đó, bạn có thể kiểm tra từng thành phần liên thông một lần bằng cách thực hiện BFS hoặc DFS như sau:

- Đầu tiên, khởi tạo một mảng N phần tử *visited* với giá trị ban đầu là `false`, được sử dụng để quản lý các đỉnh đã được thăm. Ngoài ra, khởi tạo biến *ans* với giá trị 0, được sử dụng để quản lý giá trị của câu trả lời.
- Với mỗi $i = 1, 2, \ldots, N$, thực hiện các bước sau:
  - Nếu *visited*[i] là `true`, không làm gì cả (vì chúng ta đã thăm đỉnh i).
  - Nếu *visited*[i] là `false`, tìm "thành phần liên thông chứa đỉnh i" bằng DFS hoặc BFS, và đặt *visited*[v] là `true` cho mọi đỉnh v nằm trong thành phần liên thông. Sau đó, tăng 1 vào *ans*.



Ví dụ, trong ví dụ 1 (hình trên), chúng ta kiểm tra các thành phần liên thông như sau:

- Chúng ta chưa thăm đỉnh 1, vì vậy chúng ta kiểm tra thành phần liên thông chứa đỉnh 1. Tập hợp các đỉnh của thành phần liên thông là $\{1,2,3\}$.
- Bỏ qua đỉnh 2 vì chúng ta đã thăm rồi.
- Bỏ qua đỉnh 3 vì chúng ta đã thăm rồi.
- Chúng ta chưa thăm đỉnh 4, vì vậy chúng ta kiểm tra thành phần liên thông chứa đỉnh 4. Tập hợp các đỉnh của thành phần liên thông là $\{4,5\}$.
- Bỏ qua đỉnh 5 vì chúng ta đã thăm rồi.

Thời gian thực hiện là $O(N+M)$, đủ nhanh để giải quyết bài toán này.

Mã tham khảo C++ - sử dụng DFS

```cpp
#include <iostream>
#include <vector>
using namespace std;

int N, M;
vector<vector<int>> g;
vector<int> vis;

void dfs(int c) {
   vis[c] = true;
   for (auto& d : g[c]) {
      if (vis[d]) continue;
      dfs(d);
   }
}

int main() {
   cin >> N >> M;
   g.resize(N);
   for (int i = 0; i < M; i++) {
      int u, v;
      cin >> u >> v;
      --u, --v;
      g[u].push_back(v);
      g[v].push_back(u);
   }
   int ans = 0;
   vis.resize(N);
   for (int i = 0; i < N; i++) {
      if (!vis[i]) ans++, dfs(i);
   }
   cout << ans << "\n";
}
```

Mã tham khảo C++ - sử dụng BFS

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int main() {
   int N, M;
   cin >> N >> M;
   vector<vector<int>> g(N);
   for (int i = 0; i < M; i++) {
      int u, v;
      cin >> u >> v;
      --u, --v;
      g[u].push_back(v);
      g[v].push_back(u);
   }
   int ans = 0;
   vector<int> vis(N);
   queue<int> Q;
   for (int i = 0; i < N; i++) {
      if (vis[i]) continue;
      ans++, vis[i] = true;
      Q.push(i);
      while (!Q.empty()) {
         int c = Q.front();
         Q.pop();
         for (auto& d : g[c]) {
            if (vis[d]) continue;
            vis[d] = true, Q.push(d);
         }
      }
   }
   cout << ans << "\n";
}
```

4. Một số bài tập khác

4.1 Xây dựng đường [BUILDROAD]

Quốc gia X có n thành phố và m con đường nối giữa chúng. Bạn cần phải xây dựng thêm một số con đường mới sao cho luôn có đường đi giữa mọi cặp thành phố.

Nhiệm vụ của bạn là tìm ra số lượng tối thiểu các con đường cần thiết, và xác định các con đường nào cần được xây dựng.

Dữ liệu

- Dòng đầu tiên chứa hai số nguyên n và m: số lượng thành phố và con đường. Các thành phố được đánh số từ $1, 2, \ldots, n$.
- Tiếp theo là m dòng, mỗi dòng mô tả một con đường. Mỗi dòng chứa hai số nguyên a và b: có một con đường nối giữa hai thành phố a và b.
- Một con đường luôn nối hai thành phố khác nhau, và có tối đa một con đường giữa hai thành phố bất kỳ.

Kết quả

- Đầu tiên in ra một số nguyên k: số lượng con đường cần xây thêm.
- Sau đó, in ra k dòng, mỗi dòng mô tả một con đường mới. Nếu có nhiều giải pháp, bạn cần in ra giải pháp mà chỉ số thành phố được đưa ra theo thứ tự tăng dần.

Ràng buộc

- $1 \le n \le 10^5$
- $1 \le m \le 2 \cdot 10^5$
- $1 \le a, b \le n$

Ví dụ

Input	Output
4 2 1 2 3 4	1 2 3

Hướng tiếp cận

Để kết nối tất cả các thành phố, ta cần đảm bảo rằng đồ thị là liên thông, tức là có một đường đi giữa mọi cặp thành phố. Nếu đồ thị không liên thông, ta cần thêm các con đường để nối các thành phần liên thông rời rạc lại với nhau.

Giải thuật

1. Duyệt qua tất cả các đỉnh và tìm các thành phần liên thông sử dụng thuật toán DFS hoặc BFS.
2. Số lượng thành phần liên thông rời rạc là c.
3. Để kết nối tất cả các thành phố, ta cần ít nhất $c-1$ con đường để nối các thành phần liên thông lại với nhau.
4. Chọn một đỉnh từ mỗi thành phần liên thông và thêm các con đường giữa các đỉnh này.

Độ phức tạp thời gian của thuật toán là $O(n+m)$, do chúng ta chỉ cần duyệt qua tất cả các đỉnh và cạnh một lần để tìm các thành phần liên thông.

Mã tham khảo C++

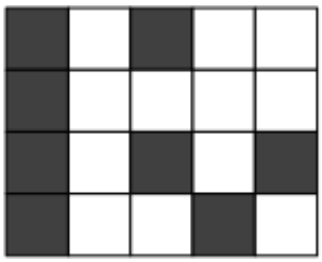
```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> adj[100001];  // Danh sach ke cua cac thanh pho
bool visited[100001];  // Mang danh dau cac thanh pho da duoc tham

// Ham DFS de tim thanh phan lien thong
void dfs(int v, vector<int>& component) {
    visited[v] = true;
    component.push_back(v);
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs(u, component);
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; ++i) {
        int a, b;
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    vector<vector<int>> components;  // Danh sach cac thanh phan lien thong
    for (int i = 1; i <= n; ++i) {
        if (!visited[i]) {
            vector<int> component;
            dfs(i, component);
            components.push_back(component);
        }
    }

    // So luong duong can them la so thanh phan lien thong - 1
    int k = components.size() - 1;
    cout << k << endl;

    // In ra cac duong can them
    for (int i = 0; i < k; ++i) {
        cout << components[i][0] << " " << components[i + 1][0] << endl;
    }

    return 0;
}
```

4.2 Đếm đảo [COUNT]

Bạn được cung cấp một bản đồ khu vực biển là một lưới các hình vuông, mỗi hình vuông đại diện cho một khu vực đất hoặc biển. Hình dưới đây là một ví dụ về một bản đồ với ô màu đen là đất, màu trắng là biển.

Bạn có thể đi từ một khu vực đất vuông này đến một khu vực đất vuông khác nếu chúng nằm kề nhau theo chiều ngang, chiều dọc hoặc đường chéo trên bản đồ. Hai khu vực được coi là cùng một hòn đảo nếu và chỉ nếu bạn có thể đi từ một khu vực này đến khu vực khác có thể thông qua các khu vực đất khác. Khu vực biển trên bản đồ được bao quanh bởi biển và do đó bạn không thể đi ra khỏi khu vực.

Bạn được yêu cầu viết một chương trình đọc bản đồ và đếm số hòn đảo trên đó. Ví dụ, bản đồ trong dưới đây bao gồm ba hòn đảo.



Dữ liệu

Dữ liệu nhập bao gồm một loạt các bộ dữ liệu, mỗi bộ dữ liệu có định dạng như sau:

- Dòng 1: ghi hai số nguyên w h ($1 \le w, h \le 50$);
- Tiếp theo là h dòng, mỗi dòng gồm w số nguyên mô tả quần đảo. Mỗi số nguyên chỉ có thể là `0` hoặc `1`.

Kết thúc dữ liệu nhập được chỉ định bằng một dòng chứa hai số không.

Kết quả

- Đối với mỗi bộ dữ liệu, in ra số lượng các hòn đảo trên một dòng.

Ví dụ

Input	Output
1 1 0 2 2 0 1 1 0 3 2 1 1 1 1 1 1 5 4 1 0 1 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 1 0 5 4 1 1 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 1 1 5 5 1 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0 0	0 1 1 3 1 9

Hướng dẫn giải

1. **Đọc dữ liệu đầu vào**: Đọc các bộ dữ liệu, mỗi bộ dữ liệu bao gồm các giá trị w và h, sau đó là ma trận $w \times h$ gồm các giá trị 0 và 1.

2. **Xác định các hòn đảo**: Sử dụng thuật toán DFS hoặc BFS để duyệt qua ma trận, đếm số lượng hòn đảo. Một hòn đảo được xác định bằng cách kết nối các ô đất theo chiều ngang, chiều dọc hoặc đường chéo.

3. **In kết quả**: Đối với mỗi bộ dữ liệu, in ra số lượng hòn đảo tìm được trên một dòng.

Giải pháp bằng C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int w, h;
vector<vector<int>> grid;
vector<vector<bool>> visited;
int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1};
int dy[] = {-1, 0, 1, -1, 1, -1, 0, 1};

void dfs(int x, int y)
{
   visited[x][y] = true;
   for (int i = 0; i < 8; ++i)
   {
      int nx = x + dx[i];
      int ny = y + dy[i];
```

```
        if (nx >= 0 && nx < h && ny >= 0 && ny < w && grid[nx][ny] == 1 && !visited[nx][
            ny])
        {
            dfs(nx, ny);
        }
    }
}

int main()
{
    while (cin >> w >> h)
    {
        if (w == 0 && h == 0)
        break;

        grid.assign(h, vector<int>(w));
        visited.assign(h, vector<bool>(w, false));

        for (int i = 0; i < h; ++i)
        {
            for (int j = 0; j < w; ++j)
            {
                cin >> grid[i][j];
            }
        }

        int island_count = 0;
        for (int i = 0; i < h; ++i)
        {
            for (int j = 0; j < w; ++j)
            {
                if (grid[i][j] == 1 && !visited[i][j])
                {
                    dfs(i, j);
                    island_count++;
                }
            }
        }

        cout << island_count << endl;
    }
    return 0;
}
```

4.3 Xóa cạnh [EDGE]

Cho đồ thị vô hướng gồm N đỉnh, M ($1 \le N, M \le 3000$) cạnh. Bạn cần xoá lần lượt từng đỉnh của đồ thị theo danh sách cho trước. Mỗi khi xoá một đỉnh thì các cạnh nối tới đỉnh đó sẽ bị xoá theo.

Yêu cầu: mỗi lần xoá xong một đỉnh, bạn cần cho biết lúc này đồ thị có còn liên thông hay không.

Dữ liệu

- Dòng 1: chứa hai số nguyên N và M.
- M dòng tiếp theo mô tả mỗi đường đi giữa các cặp đỉnh (đỉnh được đánh số từ 1 đến N).
- N dòng cuối cùng đưa ra một hoán vị của 1 đến N mô tả thứ tự đỉnh bị bị xoá.

Kết quả

Đầu ra gồm N dòng, mỗi dòng chứa "YES"hoặc "NO". Dòng đầu tiên cho biết đồ thị ban đầu có liên thông hay không, và dòng thứ $i+1$ cho biết đồ thị có còn liên thông không sau khi xoá đỉnh thứ i đi.

Ví Dụ

input	output
4 3 1 2 2 3 3 4 3 4 1 2	YES NO YES YES

Giải thuật

1. Sử dụng thuật toán DFS hoặc BFS để kiểm tra xem đồ thị ban đầu có liên thông hay không.
2. Xoá lần lượt từng đỉnh theo thứ tự cho trước
   - Đánh dấu đỉnh bị xoá và cập nhật danh sách kề.
   - Sử dụng DFS hoặc BFS để kiểm tra xem đồ thị sau khi xoá đỉnh có còn liên thông hay không.

Giải pháp bằng C++

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 3005;
int n, m, cnt = 0;
vector<int> g[N];    // Bieu dien do thi bang danh sach ke
bool vis[N], del[N]; // danh dau tham dinh, danh dau xoa dinh

void dfs(int s)
{
   if (vis[s] || del[s])
   return;
   vis[s] = 1;
   ++cnt;
   for (auto v : g[s])
   if (!vis[v])
   dfs(v);
}

int main()
{
   cin >> n >> m;
   for (int i = 0; i < m; i++)
   {
      int u, v;
      cin >> u >> v;
      g[u].push_back(v);
      g[v].push_back(u);
   }
```

```
    dfs(1);                                  // Truoc het dfs de xem co tham du n dinh
        khong
    cout << (cnt == n ? "YES\n" : "NO\n"); // tham du n dinh thi do thi ban dau lien
        thong
    int x[N];                                // Luu danh sach dinh can xoa theo thu tu
    for (int i = 1; i <= n; i++)
    cin >> x[i];
    for (int i = 1; i < n; i++) // Thuc hien xoa tung dinh theo thu tu
    {
        fill(vis, vis + N, 0);                      // Dat lai trang thai tat ca cac dinh
            chua duoc tham
        cnt = 0;                                    // Reset bien dem xem da tham bao
            nhieu dinh
        del[x[i]] = 1;                              // Danh dau da xoa dinh x[i]
        dfs(x[n]);                                  // Goi DFS dinh cuoi cung duoc xoa de
            dam bao khong goi DFS vao dinh da xoa
        cout << (cnt == n - i ? "YES\n" : "NO\n"); // Xoa i dinh roi thi do thi con lai
            n - i dinh
    }
}
```

4.4 Nhiệm vụ [TASK]

Có N người lính ($1 \le N \le 200$) đứng trên các điểm toạ độ nguyên của trục toạ độ Oxy. Mỗi người có cường độ âm thanh là P, nghĩa là có thể nói cho người cách mình nhiều nhất P mét nghe rõ.

Bây giờ, người chỉ huy muốn thông báo nhiệm vụ bằng cách chọn một người rồi yêu cầu người này thông báo nhiệm vụ tới những người trong phạm vi có thể nghe rõ lời của anh ta, khi một người nghe rõ nhiệm vụ, người đó cũng sẽ lặp lại thao tác tương tự.

Yêu cầu: Hãy cho biết trong trường hợp chỉ huy chọn được một người tối ưu nhất, sẽ có tối đa bao nhiêu người nhận được thông báo nhiệm vụ.

Dữ liệu

- Dòng đầu tiên chứa số N.
- N dòng tiếp theo mỗi dòng chứa ba số nguyên: tọa độ x và y của một người lính (các số nguyên trong phạm vi từ 0 đến 25,000) và P, cường độ âm thanh của người lính này.

Kết quả

Ghi ra một dòng gồm một số nguyên là số lượng người lính tối đa có thể nhận được thông báo nhiệm vụ (tính cả người ban đầu được chỉ huy chọn).

Ví Dụ

input	output
4 1 3 5 5 4 3 7 2 1 6 1 1	3

Trong ví dụ trên, nếu chỉ huy chọn người lính 1 để bắt đầu thông báo thì sẽ có số người nhận được thông báo tối đa bằng 3.

Hướng dẫn giải

1. **Đọc dữ liệu đầu vào**: Nhập số lượng người lính N. Sau đó nhập tọa độ (x, y) và cường độ âm thanh P cho từng người lính.
2. **Tạo danh sách kề**: Tạo danh sách kề cho đồ thị, trong đó mỗi đỉnh là một người lính và mỗi cạnh là khả năng thông báo giữa hai người lính nếu họ nằm trong phạm vi cường độ âm thanh của nhau.
3. **Duyệt tìm số lượng người nhận thông báo tối đa**: Sử dụng thuật toán DFS hoặc BFS để tìm số lượng người nhận được thông báo nhiệm vụ từ mỗi người lính ban đầu và cập nhật kết quả tối đa.

Giải pháp bằng C++

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <cstring>

using namespace std;

struct Soldier
{
   int x, y, p;
};

vector<Soldier> soldiers;
vector<int> adj[201];
bool visited[201];

void dfs(int u, int &count)
{
   visited[u] = true;
   count++;
   for (int v : adj[u])
   {
      if (!visited[v])
      {
         dfs(v, count);
      }
   }
}

int main()
{
   int N;
   cin >> N;
   soldiers.resize(N);

   for (int i = 0; i < N; i++)
   {
      cin >> soldiers[i].x >> soldiers[i].y >> soldiers[i].p;
   }

   /*
   *   Xay dung do thi: moi nguoi linh la mot dinh
   *   Co canh noi tu i den j khi khoang cach hai nguoi linh nay
   *   nho hon hoac bang cuong do am thanh cua nguoi linh i
   */
   for (int i = 0; i < N; i++)
   {
      for (int j = i + 1; j < N; j++)
      {
```

```
            int dx = soldiers[j].x - soldiers[i].x;
            int dy = soldiers[j].y - soldiers[i].y;
            double distance = sqrt(dx * dx + dy * dy);

            if (distance <= soldiers[i].p)
            {
               adj[i].push_back(j);
            }
            if (distance <= soldiers[j].p)
            {
               adj[j].push_back(i);
            }
         }
      }
      /**
      * So luong nguoi duoc thong bao nhieu nhat chinh la so luong
      * dinh cua thanh phan lien thong co kich thuoc lon nhat.
      */
      int maxCount = 0;
      for (int i = 0; i < N; i++)
      {
         memset(visited, false, sizeof(visited));
         int count = 0;
         dfs(i, count);
         if (count > maxCount)
         {
            maxCount = count;
         }
      }

      cout << maxCount << endl;

      return 0;
}
```

4.5 Xây dựng hàng rào [FENCE]

Có N ($2 \le N \le 10^5$) con cừu được đánh số từ 1 đến N, mỗi con cừu được đặt tại một vị trí (x, y) riêng biệt trên bản đồ 2D của trang trại. Có M cặp bò ($1 \le M < 10^5$) là bạn của nhau. Hai con cừu mà là bạn của nhau thì thuộc cùng một mạng lưới bạn bè.

Chủ trang trại muốn xây dựng một hàng rào hình chữ nhật có các cạnh song song với các trục x và y. Ông ấy muốn đảm bảo rằng ít nhất một mạng lưới bàn bè của các con cừu được hoàn toàn bao quanh bởi hàng rào (các con cừu trên biên của hình chữ nhật được tính là đã được bao quanh).

Hãy xác định chu vi nhỏ nhất có thể của một hàng rào thỏa mãn yêu cầu này. Có thể có trường hợp hàng rào này có chiều rộng hoặc chiều cao bằng không.

Dữ liệu

- Dòng đầu tiên của đầu vào chứa N và M.
- N dòng tiếp theo mỗi dòng chứa tọa độ x và y của một con cừu (số nguyên không âm có giá trị không quá 10^8).
- M dòng tiếp theo mỗi dòng chứa hai số nguyên a và b mô tả một mối quan hệ bạn bè giữa hai con cừu a và b. Mỗi con cừu có ít nhất một mối quan hệ bạn bè, và không có mối quan hệ bạn nào được lặp lại trong dữ liệu vào.

Kết quả

- Ghi một số nguyên duy nhất là chu vi nhỏ nhất của một hàng rào thỏa mãn yêu cầu.

Ví Dụ

fenceplan.in	fenceplan.out
7 5 0 5 10 5 5 0 5 10 6 7 8 6 8 4 1 2 2 3 3 4 5 6 7 6	10

Hướng dẫn giải

1. **Đọc dữ liệu đầu vào**: Nhập số lượng cừu N và số lượng cặp bạn bè M. Sau đó nhập tọa độ (x, y) cho từng con cừu và các mối quan hệ bạn bè giữa các cặp bò.
2. **Xây dựng danh sách kề**: Tạo danh sách kề để lưu trữ thông tin về các mối quan hệ bạn bè giữa các con cừu.
3. **Tìm các mạng lưới bạn bè**: Sử dụng thuật toán DFS hoặc BFS để tìm các mạng lưới bạn bè của các con cừu.
4. **Tính toán chu vi nhỏ nhất của hàng rào**: Với mỗi mạng lưới bạn bè, tính toán tọa độ nhỏ nhất và lớn nhất của các con cừu trong mạng lưới đó để xác định kích thước hình chữ nhật bao quanh và chu vi của nó. Lưu lại chu vi nhỏ nhất.
5. **In kết quả**: In ra chu vi nhỏ nhất của hàng rào thỏa mãn yêu cầu.

Mã tham khảo C++

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Cow {
   int x, y;
};

vector<vector<int>> adj;
vector<bool> visited;
vector<Cow> cows;

void dfs(int v, int& minX, int& minY, int& maxX, int& maxY) {
   visited[v] = true;
   minX = min(minX, cows[v].x);
   minY = min(minY, cows[v].y);
   maxX = max(maxX, cows[v].x);
   maxY = max(maxY, cows[v].y);

```

```
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs(u, minX, minY, maxX, maxY);
        }
    }
}

int main() {
    int N, M;
    cin >> N >> M;
    cows.resize(N);
    adj.resize(N);
    visited.resize(N, false);

    for (int i = 0; i < N; ++i) {
        cin >> cows[i].x >> cows[i].y;
    }

    for (int i = 0; i < M; ++i) {
        int a, b;
        cin >> a >> b;
        --a; --b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    int minPerimeter = INT_MAX;

    for (int i = 0; i < N; ++i) {
        if (!visited[i]) {
            int minX = INT_MAX, minY = INT_MAX;
            int maxX = INT_MIN, maxY = INT_MIN;
            dfs(i, minX, minY, maxX, maxY);
            int perimeter = 2 * ((maxX - minX) + (maxY - minY));
            minPerimeter = min(minPerimeter, perimeter);
        }
    }

    cout << minPerimeter << endl;

    return 0;
}
```

5. Bài tập tự luyện độ khó tăng dần

Dưới đây là danh sách 10 bài tập tự luyện trên oj.vnoi.info. Quý thầy cô có thể click vào tên bài để mở trang chấm online.

1. VNOI - Gặm cỏ
2. VNOI - Truyền tin
3. VNOI - Nước lạnh
4. VNOI - Tic tac toe
5. VNOI - Truy vết
6. VNOI - Tìm thành phần liên thông mạnh
7. VNOI - Quảng cáo
8. VNOI - Tìm khớp và cầu cơ bản
9. VNOI - Bảo vệ nông trang
10. VNOI - Bộ ba cao thủ

6. Kết luận

DFS và BFS là hai thuật toán cơ bản nhưng cực kỳ mạnh mẽ trong việc xử lý các bài toán đồ thị. Mỗi thuật toán có ưu điểm riêng và phù hợp với các loại bài toán khác nhau. Hiểu rõ các đặc điểm và ứng dụng của từng thuật toán giúp chúng ta lựa chọn và áp dụng chúng một cách hiệu quả trong thực tế. Các nghiên cứu và phát triển tiếp theo có thể tập trung vào việc tối ưu hóa các thuật toán này và khám phá các ứng dụng mới trong các lĩnh vực đa dạng.

Nghiên cứu về đồ thị, DFS và BFS không chỉ dừng lại ở lý thuyết mà còn mở ra nhiều cơ hội ứng dụng thực tiễn, góp phần giải quyết nhiều vấn đề phức tạp trong cuộc sống và công nghệ.