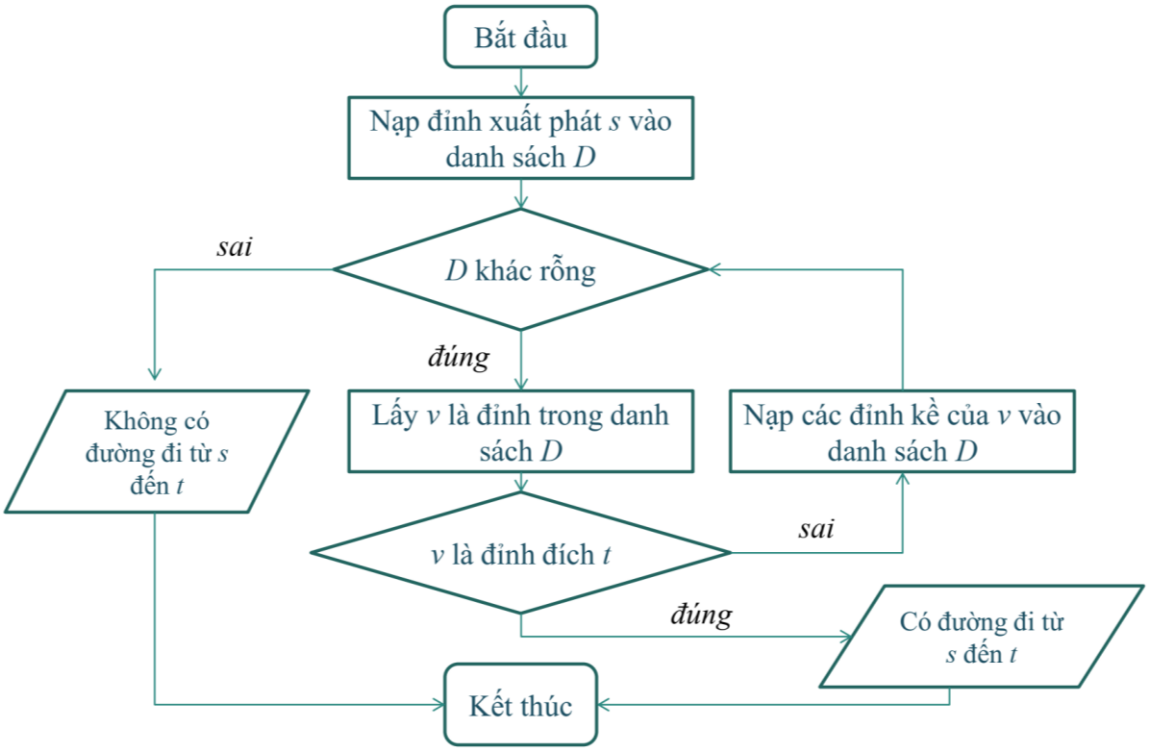
CHƯƠNG 3: TÌM KIẾM TRÊN ĐỒ THỊ

Contents

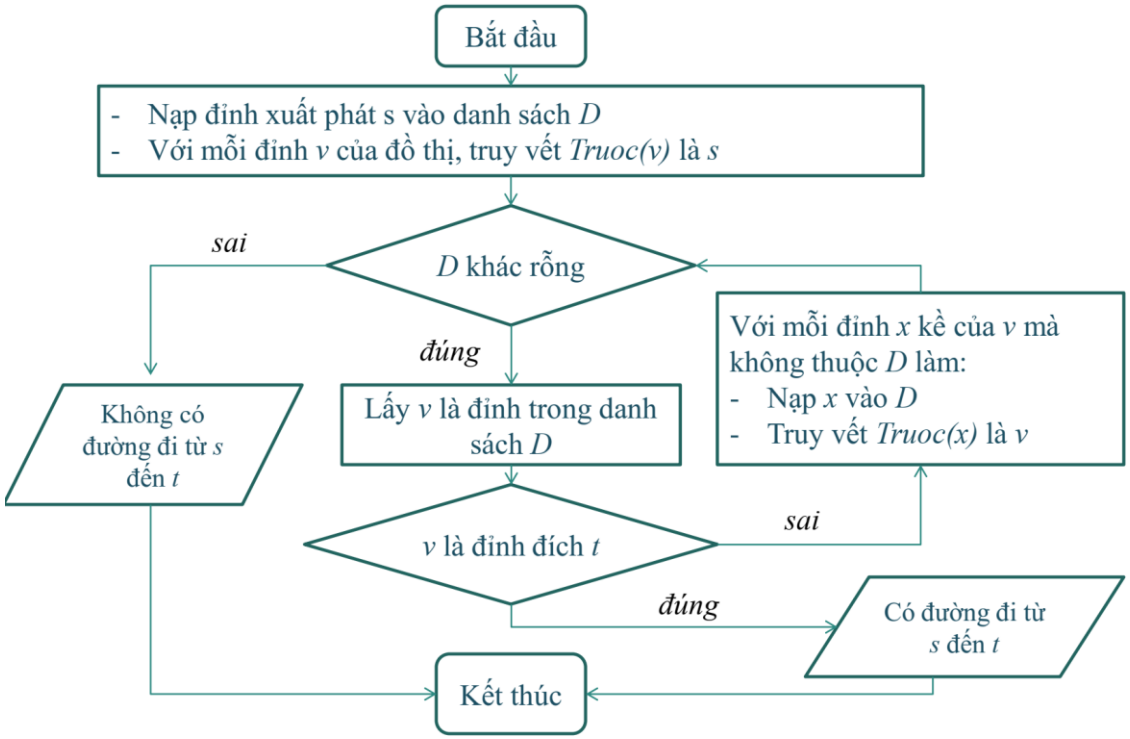
3.1 Bài toán tìm kiếm trên đồ thị

- Tìm đường đi từ một đỉnh đến một đỉnh khác
    - Kiểm tra tính liên thông
    - Kiểm tra tính liên thông mạnh
    - Xác định các thành phần liên thông của đồ thị
    - Kiểm tra đồ thị có là đồ thị hai phía hay không?
- Duyệt qua các đỉnh của đồ thị: Để cập nhật, xử lý dữ liệu tại các đỉnh của đồ thị
- ***Tìm đường đi:*** Từ đỉnh xuất phát s Đến đỉnh đích t

    + *Yêu cầu 1:* Tồn tại đường đi hay không tồn tại đường đi



    + *Yêu cầu 2:* Nếu tồn tại đường đi từ s → t thì đi như thế nào?



3.2 Thuật toán tìm kiếm theo chiều sâu (DFS) (Depth First Search)

- Từ đỉnh (nút) gốc ban đầu, thuật toán *duyệt đi xa nhất theo từng nhánh*, khi nhánh đã duyệt hết, *lùi về từng đỉnh để tìm và duyệt những nhánh tiếp theo* (trở về đỉnh ngay trước đỉnh mà không thể đi tiếp để tìm qua nhánh khác) (*Ưu tiên duyệt xuống nhất có thể trước khi quay lại*).

Quá trình duyệt chỉ dừng lại khi tìm thấy đỉnh cần tìm hoặc tất cả đỉnh đều đã được duyệt qua.

- Idea is similar to **pre-order** traversal (*visit children first*)
- Trong quá trình tìm kiếm DFS tổ chức lưu trữ danh sách các đỉnh theo kiểu LIFO - Last In First Out.
- Thuật toán có thể được cài đặt bằng phương pháp ***đệ qui*** hay phương pháp lặp sử dụng ***cấu trúc stack*** để khử đệ qui.

Input: G (V, E) và đỉnh s ∈ V
Output: Dãy đỉnh được viếng thăm

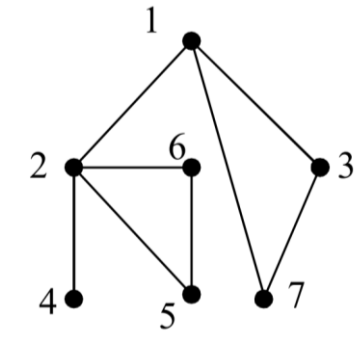
- ***Bước 1:*** Viếng thăm đỉnh s (In đỉnh s ra màn hình)
- ***Bước 2:*** Tìm đỉnh u: *u kề với s và u chưa được viếng thăm*
  - + Nếu có đỉnh u thì đặt s = u và quay về *Bước 1*
  - + Nếu không có đỉnh u thì quay về đỉnh trước đỉnh s, và tìm hướng đi khác.
  - + Thuật toán dừng khi không thể quay về đỉnh trước.

3.2.1 Cài đặt bằng Đệ quy

```
void DFS (int s) {
      if (visited[s] == true) return;
      // Bước 1
      visited[s] = true;

      // process node s :...

      // Bước 2
      foreach (int u in v[s])
            DFS (u);
}
```

```
void DFS(int s){
      <Tham Dinh s>;
      visited[s] = true;
      // Danh dau la u da duoc tham

      //Duyet cac dinh ke voi dinh u
      for(s : adj[u]){
            // Neu s chua duoc tham
            if(!visited[s]){
                  DFS(v);
      }
}

/*Để duyệt tất cả các thành phần liên thông ta
thực hiện như sau:*/

void main() {
/* Nhập đồ thị */
      for (v ∈ V) visited[v] = 0;
      /* Khởi tạo cờ cho đỉnh */ for (v ∈ V)

      if ( visited[v] == false ) DFS(v);
}
```

Ví dụ 1: Xuất phát từ đỉnh 1, dùng thuật toán DFS duyệt qua tất cả các đỉnh của đồ thị được cho như sau:



- Xuất phát từ đỉnh 1 chưa thăm, *thăm đỉnh 1*, tìm các đỉnh kề đỉnh 1 là 2,3,7
- Chọn đỉnh 2 chưa thăm kề với 1, *thăm đỉnh 2*, tìm các đỉnh kề với 2 là **1**,4,5,6
- Chọn đỉnh 4 chưa thăm kề với 2, *thăm đỉnh 4*, tìm các đỉnh kề với 4 là **2**
- Không có đỉnh kề nào của 4 mà chưa thăm nên quay trở lại đỉnh trước của đỉnh 4 là 2 và chọn đỉnh kề chưa thăm khác của đỉnh 2.
- Chọn đỉnh 5 chưa thăm kề với 2, *thăm đỉnh 5*, tìm các đỉnh kề với 5 là **2**,6
- Chọn đỉnh 6 chưa thăm kề với 5, *thăm đỉnh 6*, các đỉnh kề với 6 là **2**,**5**
- Không có đỉnh kề nào của 6 mà chưa thăm nên quay trở lại đỉnh trước của đỉnh 6 là 5
- và chọn đỉnh kề chưa thăm khác của đỉnh 5.
- Không có đỉnh kề nào của 5 mà chưa thăm nên quay trở lại đỉnh trước của đỉnh 5 là 2
- và chọn đỉnh kề chưa thăm khác của đỉnh 2.
- Không có đỉnh kề nào của 2 mà chưa thăm nên quay trở lại đỉnh trước của đỉnh 2 là 1
- và chọn đỉnh kề chưa thăm khác của đỉnh 1.
- Chọn đỉnh 3 chưa thăm kề với 1, *thăm đỉnh 3*, tìm các đỉnh kề của 3 là **1**,7
- Chọn đỉnh 7 chưa thăm kề với 3, *thăm đỉnh 7*, tìm các đỉnh kề của 7 là **1**,3
- Không có đỉnh kề nào của 7 mà chưa thăm nên quay trở lại đỉnh trước của đỉnh 7 là 3 và chọn đỉnh kề chưa thăm khác của đỉnh 3.
- Không có đỉnh kề nào của 3 mà chưa thăm nên quay trở lại đỉnh trước của đỉnh 3 là 1 và chọn đỉnh kề chưa thăm khác của đỉnh 1.
- Không có đỉnh kề nào của 1 mà chưa thăm nên thuật toán dừng.
- Theo thứ tự thăm của các đỉnh, kết quả duyệt bằng thuật toán DFS là: **1, 2, 4, 5, 6, 3, 7**

✍**Lưu ý:** Trong quá trình thực hiện thuật toán, nếu *một đỉnh có nhiều đỉnh kề chưa thăm thì ưu tiên chọn đỉnh kề được đánh số nhỏ hơn.*

```cpp
#include <iostream>

using namespace std;

const int MAX_N = 100; // Số lượng tối đa của đỉnh trong đồ thị

bool visited[MAX_N];
int adj[MAX_N][MAX_N]; // Ma trận kề

void DFS(int s, int n) {
    if (visited[s]) return; // Bước 1
    visited[s] = true;

    // Xử lý đỉnh s ở đây
    cout << "Dang xu ly " << s << endl;

    // Bước 2: Duyệt qua tất cả các đỉnh kề của đỉnh s
    for (int u = 1; u <= n; ++u) {
        if (adj[s][u])
            DFS(u, n);
    }
}

int main() {
    int n, m; // Số đỉnh và số cạnh
    cout << "Nhap so dinh va so canh: ";
    cin >> n >> m;
```

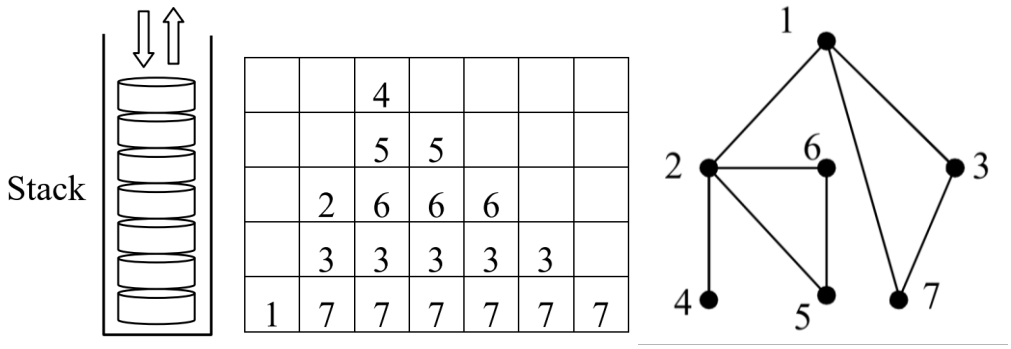
```
    cout << "Nhap " << m << " canh:" << endl;
    for (int i = 1; i <= m; ++i) {
        int u, v;
        cin >> u >> v;
        adj[u][v] = adj[v][u] = 1; // Đánh dấu cạnh u-v tồn tại
    }

    int start_node;
    cout << "Nhap dinh bat dau: ";
    cin >> start_node;

    cout << "Duyet DFS tu dinh " << start_node << ":" << endl;
    DFS(start_node, n);

    return 0;
}
```

3.2.2 Cài đặt bằng Stack

```
void DFS (int s)
{
      1. Đánh dấu s đã viếng thăm
      2. Đưa s vào stack
      3. while (stack chưa rỗng)
      {
            <Lấy 1 đỉnh u từ đỉnh stack & xóa đỉnh u>
            <Tìm các đỉnh v kề u và chưa được viếng thăm>
            {
                  Đánh dấu v đã viếng thăm
                  Đẩy v vào stack
            }
      }
}
```

```
void DFS (int s)
{
      visited[s] = true;
      push(stack, s);
      while (stack != ∅)
      {
            int u = stack.top();
            stack.pop (); //Xóa phần tử đỉnh stack
            <In đỉnh u ra màn hình>
            for (v:adj[u]){
                  if(!visted[v]){
                        visited[v] = true;
                        push(stack, v);
                  }
            }
      }
}
```

Ví dụ 2: Dùng thuật toán DFS không đệ quy duyệt qua tất cả các đỉnh của đồ thị ở ví dụ 1.



Khởi tạo: stack = ∅, đỉnh xuất phát là 1, ***bỏ đỉnh 1 vào stack***

Bước lặp: điều kiện lặp stack != ∅

- **Lấy đỉnh 1 ra khỏi stack:** pop(stack), đánh dấu đỉnh 1 đã thăm: thăm[1]=1, tìm đỉnh kề chưa thăm của đỉnh 1 mà không có trong stack: **2, 3, 7** và bỏ tất cả các đỉnh kề này vào stack theo thứ tự từ lớn đến nhỏ.
- **Lấy đỉnh 2 ra khỏi stack,** đánh dấu đỉnh 2 đã thăm: thăm[2]=1, tìm các đỉnh kề chưa thăm của 2 mà không chứa trong stack: **4, 5, 6** và bỏ tất cả các đỉnh kề này vào stack theo thứ tự từ lớn đến nhỏ.
- **Lấy đỉnh 4 ra khỏi stack,** đánh dấu đỉnh 4 đã thăm: thăm[4]=1, *đỉnh 4 không còn đỉnh kề nào chưa thăm* nên không có đỉnh nào bỏ vào stack.
- **Lấy đỉnh 5 ra khỏi stack,** đánh dấu đỉnh 5 đã thăm: thăm[5]=1, *đỉnh 5 có đỉnh kề 6 chưa thăm nhưng 6 đã có trong stack* nên cũng không có đỉnh nào bỏ vào stack.
- **Lấy đỉnh 6 ra khỏi stack,** đánh dấu đỉnh 6 đã thăm: thăm[6]=1, đỉnh này cũng không có đỉnh kề nào chưa thăm.
- **Lấy đỉnh 3 ra khỏi stack,** đánh dấu đỉnh 3 đã thăm: thăm[3]=1, đỉnh 3 có đỉnh kề 7 chưa thăm nhưng đã có trong stack nên không có đỉnh nào thêm vào stack.
- **Lấy đỉnh 7 ra khỏi stack,** đánh dấu đỉnh 7 đã thăm: thăm[7]=1, đỉnh này không có đỉnh kề nào chưa thăm.
- stack = ∅, thoát khỏi vòng lặp và kết thúc thuật toán. khi đó ta thu được dãy các đỉnh đã thăm qua như sau: **1, 2, 4, 5, 6, 3, 7.**

```
#include <iostream>
#include <stack>

using namespace std;

const int MAX_N = 100; // Số lượng tối đa của đỉnh trong đồ thị

bool visited[MAX_N];
int adj[MAX_N][MAX_N]; // Ma trận kề

void DFS(int s, int n) {
    stack<int> st;

    visited[s] = true;
    st.push(s); // Bước 2

    while (!st.empty()) {
        int u = st.top(); // Lấy 1 đỉnh u từ stack
        st.pop(); //Xóa phần tử trên đỉnh stack

        cout << "Dang xu ly dinh " << u << endl;

        // Tìm 1 đỉnh v kề u và chưa được viếng thăm
```

```
        for (int v = 1; v <= n; ++v) {
            if (adj[u][v] && !visited[v]) {
                visited[v] = true; // Đánh dấu v đã viếng thăm
                st.push(v); // Đẩy v vào stack
            }
        }
    }
}

int main() {
    int n, m; // Số đỉnh và số cạnh
    cout << "Nhap so dinh va so canh: ";
    cin >> n >> m;

    cout << "Nhap " << m << " canh:" << endl;
    for (int i = 1; i <= m; ++i) {
        int u, v;
        cin >> u >> v;
        adj[u][v] = adj[v][u] = 1; // Đánh dấu cạnh u-v tồn tại
    }

    int start_node;
    cout << "Nhap dinh bat dau: ";
    cin >> start_node;

    cout << "Duyet DFS tu dinh " << start_node << ":" << endl;

    DFS(start_node, n);

    return 0;
}
```

- Độ phức tạp tính toán của DFS tùy thuộc vào đồ thị được biểu diễn dưới dạng nào, trong trường hợp xấu nhất ta có bảng so sánh sau: *(với n: số cạnh, m: số đỉnh)*

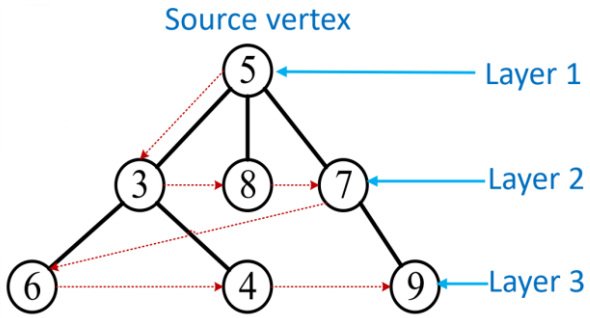
Ma trận kề	Danh sách kề	Danh sách cạnh
$O(n^2)$	O(n + m)	O(n x m)

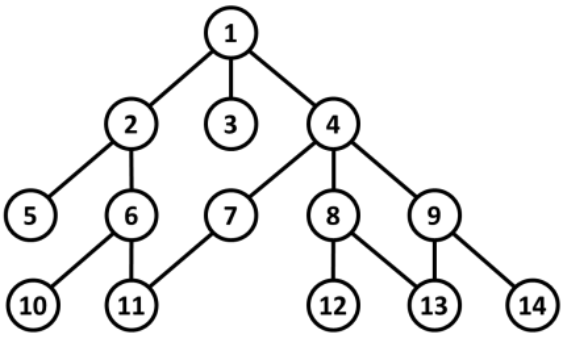
3.2.3 Ý nghĩa

- Kiểm tra đường đi giữa 2 đỉnh
- Chia đồ thị thành các thành phần liên thông
- Xây dựng cây khung của đồ thị
- Kiểm tra xem đồ thị có chu trình hay không

3.3 Thuật toán tìm kiếm theo chiều rộng (BFS) (Breadth First Search)

- BFS là một thuật toán duyệt trong đó bạn nên bắt đầu duyệt từ một đỉnh đã chọn (nút nguồn hoặc nút bắt đầu) và duyệt đồ thị theo từng lớp (layer) để khám phá các nút lân cận (các nút được kết nối trực tiếp với nút nguồn). Sau đó bạn phải di chuyển tới các nút lân cận ở mức (level) tiếp theo.
- Những đỉnh nào gần đỉnh xuất phát hơn sẽ được duyệt trước.
- Như tên gọi duyệt theo chiều rộng, việc duyệt đồ thị theo chiều rộng được thực hiện như sau:
    + Đầu tiên di chuyển theo chiều ngang và duyệt qua tất cả các đỉnh tại layer hiện tại.
    + Tiếp đến mới di chuyển đến layer tiếp theo.





Thứ tự thăm các đỉnh của BFS

- Trong quá trình tìm kiếm BFS tổ chức lưu trữ danh sách các đỉnh theo kiểu FIFO – First In First Out.
- Thuật toán có độ phức tạp là O(m+n).

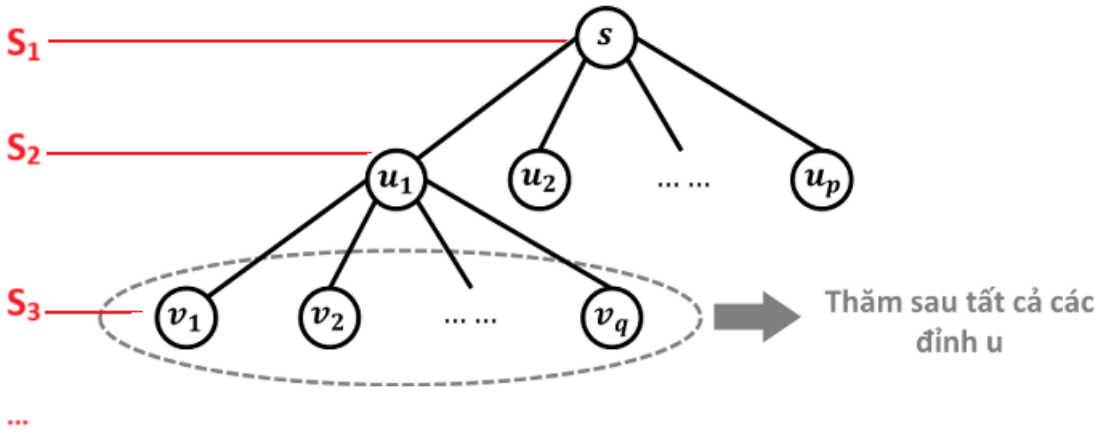
3.3.1 Ý tưởng

- Với đồ thị không trọng số và đỉnh nguồn s. Đồ thị này có thể là đồ thị có hướng hoặc vô hướng, điều đó **không quan trọng** đối với thuật toán.
- Có thể hiểu thuật toán như một ngọn lửa lan rộng trên đồ thị:
  - Ở bước thứ 0, chỉ có đỉnh nguồn s đang cháy.
  - Ở mỗi bước tiếp theo, ngọn lửa đang cháy ở mỗi đỉnh lại lan sang tất cả các đỉnh kề với nó.

Trong mỗi lần lặp của thuật toán, "vòng lửa" lại lan rộng ra theo chiều rộng. Những đỉnh nào gần s hơn sẽ bùng cháy trước.

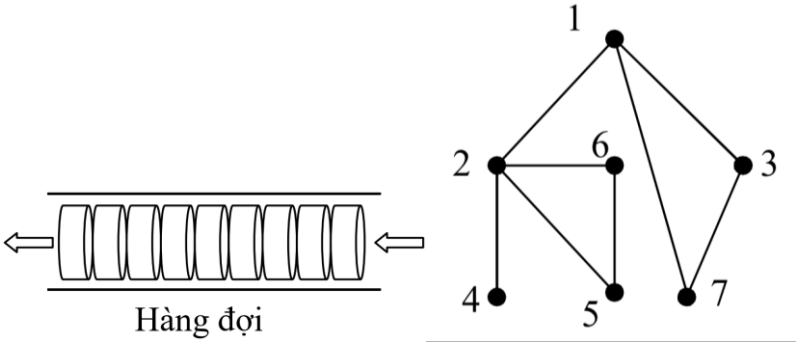
Input: G(V, E) và đỉnh s ∈ V
Output: Dãy đỉnh được viếng thăm

· Bước 1: Gọi $S_1 = \{s\}$, viếng thăm các đỉnh trong S_1
· Bước 2: Tìm các đỉnh (gọi là S_2) *kề với một trong những đỉnh S_1 (xét lần lượt các đỉnh S_1)*, và *chưa được viếng thăm*, rồi viếng thăm các đỉnh trong S_2
· Bước 3: Tìm các đỉnh (gọi là S_3) *kề với một trong những đỉnh S_2 (xét lần lượt các đỉnh S_2)* và *chưa được viếng thăm*, rồi viếng thăm các đỉnh trong S_3
· ... Cho đến khi không thể tìm thêm các đỉnh để viếng thăm.



Ví dụ 3: Xét đồ thị như trong hình 3.1, dùng thuật toán BFS để duyệt đồ thị:

6	5	4	7	3	2	1



Khởi tạo: queue = ∅, đỉnh xuất phát là 1, ***bỏ đỉnh 1 vào queue***

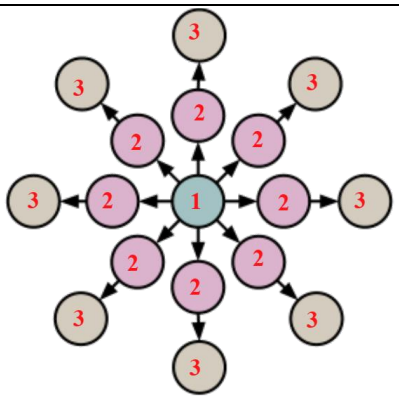
Bước lặp: điều kiện lặp queue != ∅

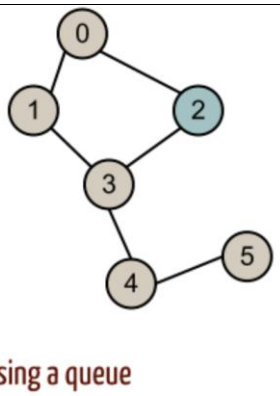
- ***Lấy đỉnh 1 ra khỏi queue:*** pop(queue), đánh dấu đỉnh 1 đã thăm: thăm[1]=1, tìm đỉnh kề chưa thăm của đỉnh 1 mà không có trong queue: **2, 3, 7** và bỏ tất cả các đỉnh kề này vào queue theo ***thứ tự từ nhỏ đến lớn.***
- ***Lấy đỉnh 2 ra khỏi queue,*** đánh dấu đỉnh 2 đã thăm: thăm[2]=1, tìm các đỉnh kề chưa thăm của 2 mà không chứa trong queue: **4, 5, 6** và bỏ tất cả các đỉnh kề này vào queue theo thứ tự từ nhỏ đến lớn.
- ***Lấy đỉnh 3 ra khỏi queue,*** đánh dấu đỉnh 3 đã thăm: thăm[3]=1, đỉnh 3 không còn đỉnh kề nào chưa thăm mà không có trong queue nên không có đỉnh nào bỏ vào queue.
- ***Lấy đỉnh 7 ra khỏi queue,*** đánh dấu đỉnh 7 đã thăm: thăm[7]=1, đỉnh 7 không còn đỉnh kề nào chưa thăm.
- ***Lấy đỉnh 4 ra khỏi queue,*** đánh dấu đỉnh 4 đã thăm: thăm[4]=1, đỉnh này cũng không có đỉnh kề nào chưa thăm.
- ***Lấy đỉnh 5 ra khỏi queue,*** đánh dấu đỉnh 5 đã thăm: thăm[5]=1, đỉnh 5 có đỉnh kề 6 chưa thăm nhưng đã có trong queue nên không có đỉnh nào thêm vào queue.
- ***Lấy đỉnh 6 ra khỏi queue,*** đánh dấu đỉnh 6 đã thăm: thăm[6]=1, đỉnh này không có đỉnh kề nào chưa thăm.
- ***queue*** = ∅, thoát khỏi vòng lặp và kết thúc thuật toán. khi đó ta thu được dãy các đỉnh đã thăm qua như sau: **1, 2, 3, 7, 4, 5, 6.**

3.3.2 Cài đặt bằng hàng đợi (queue)

```
void BFS (int s)
{
	//Bước 1: Khởi tạo
	queue q = ∅;
	visited[s] = true;
	q.push(s) ; //Đẩy đỉnh s vào hàng đợi
	// Process node s

	//Bước 2: Lặp lại khi hàng đợi còn phần tử
	while (q.Count != 0) { //Khi hàng đợi chưa rỗng
		s = q.front();  //Truy cập vào đỉnh ở đầu hàng đợi
		q.pop ();  //Xóa đỉnh ở đầu hàng đợi

		<Tiến hành in đỉnh s ra màn hình>

		//Duyệt đỉnh kề với đỉnh s mà chưa được thăm, rồi đẩy vào queue
		foreach (int u in v[s]) {
```

```
                    if (visited[u] == false){
                        visited[u]=true;
                        q.push(u) ;
                        // Process node u
                    }
                }
            }
}
```






```
1. q = {}
2. q = {2}
3. q = {0, 3}
4. q = {1}
5. q = {4}
6. q = {5}
```

```
#include <iostream>
#include <queue>

using namespace std;

const int MAX_N = 100; // Số lượng tối đa của đỉnh trong đồ thị

bool visited[MAX_N];
int adj[MAX_N][MAX_N]; // Ma trận kề

void BFS(int s, int n) {
    queue<int> q;

    visited[s] = true;
    q.push(s);

    while (!q.empty()) {
        s = q.front();
        q.pop();

        // Xử lý đỉnh s ở đây
        cout << "Dang xu ly dinh " << s << endl;

        // Duyệt qua tất cả các đỉnh kề của đỉnh s
        for (int u = 0; u < n; ++u) {
            if (adj[s][u] && !visited[u]) {
                visited[u] = true;
                q.push(u);
            }
        }
    }
}

int main() {
    int n, m; // Số đỉnh và số cạnh
    cout << "Nhap so dinh va so canh: ";
    cin >> n >> m;

    cout << "Nhap " << m << " canh:" << endl;
```

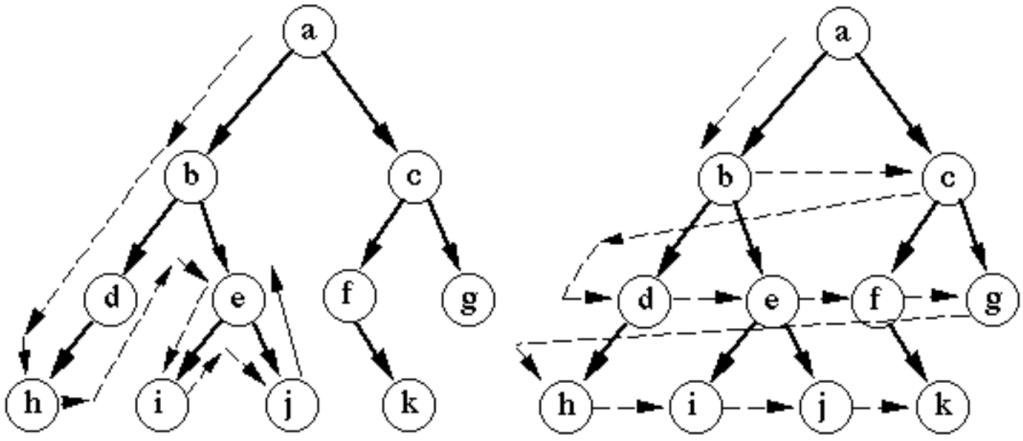
```
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        adj[u][v] = adj[v][u] = 1; // Đánh dấu cạnh u-v tồn tại
    }

    int start_node;
    cout << "Nhap dnh bat dau: ";
    cin >> start_node;

    cout << "Duyet BFS tu dinh " << start_node << ":" << endl;

    BFS(start_node, n);

    return 0;
}
```

3.3.3 Ý nghĩa

- Kiểm tra đường đi giữa 2 đỉnh
- Chia đồ thị thành các thành phần liên thông
- Xây dựng cây khung của đồ thị
- Tìm đường đi ngắn nhất từ 1 đỉnh đến các đỉnh còn lại

3.4 Đặc điểm của 2 thuật toán

- BFS cho kết quả là đường đi ngắn nhất (tính theo số cung/cạnh).
- Do đó nếu đích "ở gần" đỉnh xuất phát thì BFS có thể cho kết quả nhanh hơn DFS.
- Ngược lại nếu đích "xa" đỉnh xuất phát, BFS có thể cho kết quả chậm hơn DFS.
- Đối với đồ thị dạng cây và có độ sâu lớn, DFS có thể đi vào "ngõ cụt" do sự bùng nổ tổ hợp.

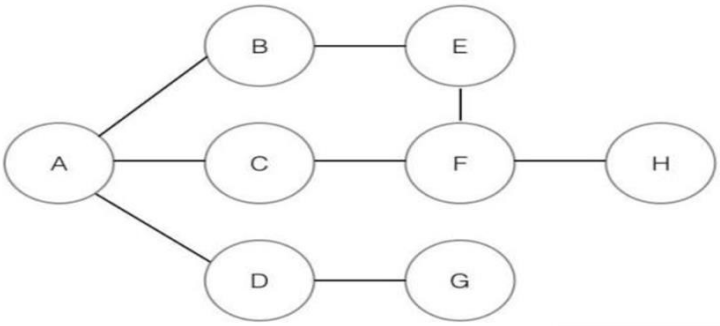


Depth-first search Breadth-first search

3.5 Các ví dụ

Ví dụ 1: DFS & BFS từ đỉnh A

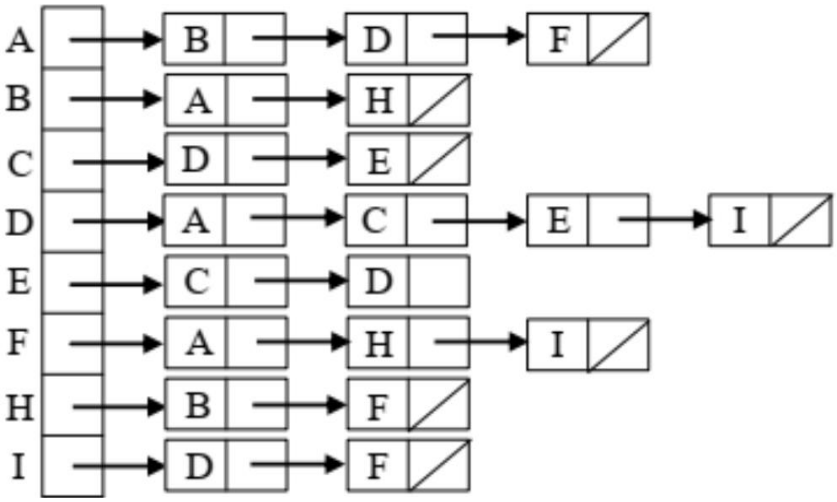
	A	B	C	D	E	F	G	H
A	0	1	1	1	0	0	0	0
B	1	0	0	0	1	0	0	0
C	1	0	0	0	0	1	0	0
D	1	0	0	0	0	0	1	0
E	0	1	0	0	0	1	0	0
F	0	0	1	0	1	0	0	1
G	0	0	0	1	0	0	0	0
H	0	0	0	0	0	1	0	0

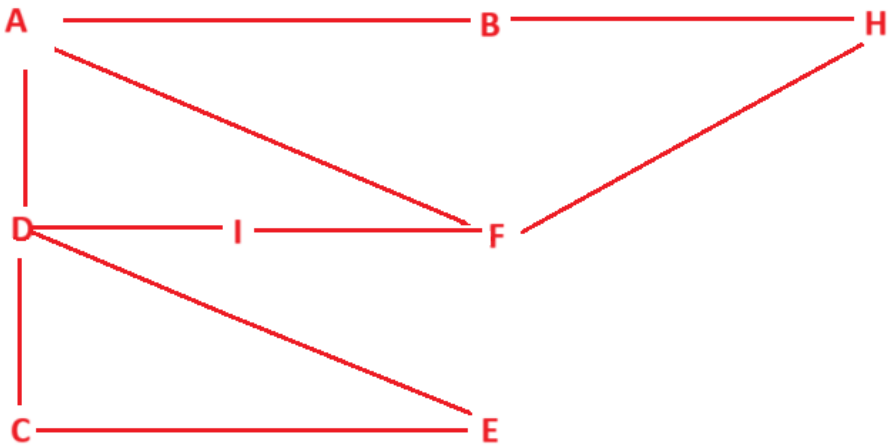


A: B, C, D
B: A, E
C: A, F
D: A, G
E: B, F
F: C, E, H
G: D
H: F

DFS (stack)	BFS (queue)
$S = [A]$	$Q = [A]$
$S = [D, C, B]$	$Q = [B, C, D]$
$S = [G, C, B]$	$Q = [C, D, E]$
$S = [C, B]$	$Q = [D, E, F]$
$S = [F, B]$	$Q = [E, F, G]$
$S = [H, E, B]$	$Q = [F, G]$
$S = [E, B]$	$Q = [G, H]$
$S = [B]$	$Q = [H]$
$S = []$	$Q = []$

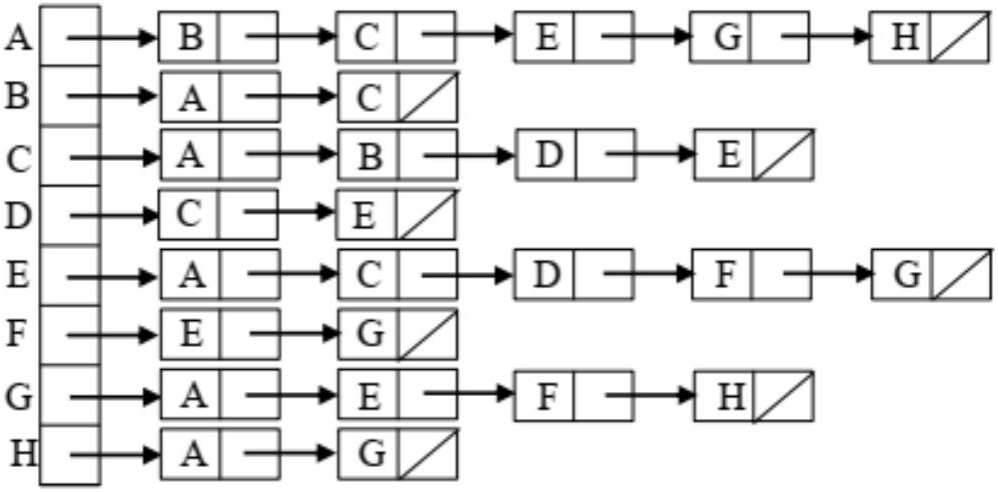
Ví dụ 2: DFS & BFS từ I

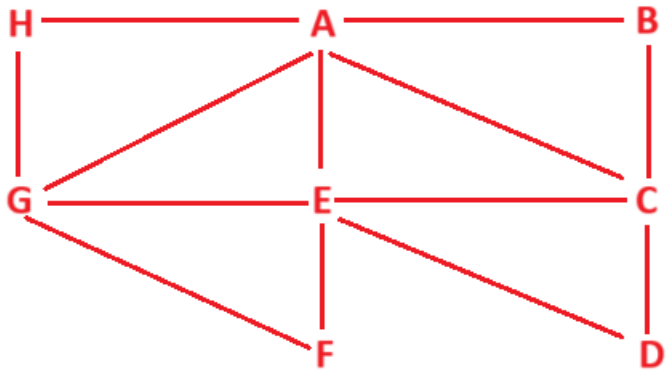




- **DFS:** I, D, A, B, H, C, E, F
- **BFS:** I, F, D, H, A, E, C, B

Ví dụ 3: DFS & BFS từ A





- **DFS:** A, B, C, D, E, F, G, H
- **BFS:** A, B, C, E, G, H, D, FS

3.6 Ứng dụng

3.6.1 Tìm đường đi giữa hai đỉnh

- Bài toán: Cho đồ thị G, s và t là hai đỉnh tùy ý của đồ thị. Hãy tìm đường đi từ s đến t.

- Phương pháp

+ Bắt đầu từ đỉnh s, Sử dụng DFS hoặc BFS để duyệt đồ thị.
  - Tìm thấy visited[t] = 1
  - Không tìm thấy visited[t] = 0

+ Để ghi nhận đường đi từ s đến t ta sử dụng thêm mảng trước[v] để lưu lại đỉnh trước của đỉnh v trong đường đi từ s đến t.

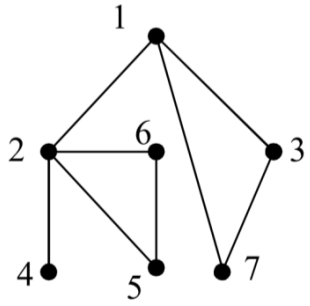
Dùng thuật toán DFS đệ quy:

```
void DFS (int v){
      visited[v] = true;
      for(u : adj[v]){
            if(visited[u] = 0){
                  trước[u]=v; //Lưu lại đỉnh trước của u là v
                  DFS(u); //Gọi đệ quy
            }
      }
}
void main(){
      for (v ∈ V) visited[v] = false; // Khởi tạo cờ cho đỉnh
      DFS(v);
}
```

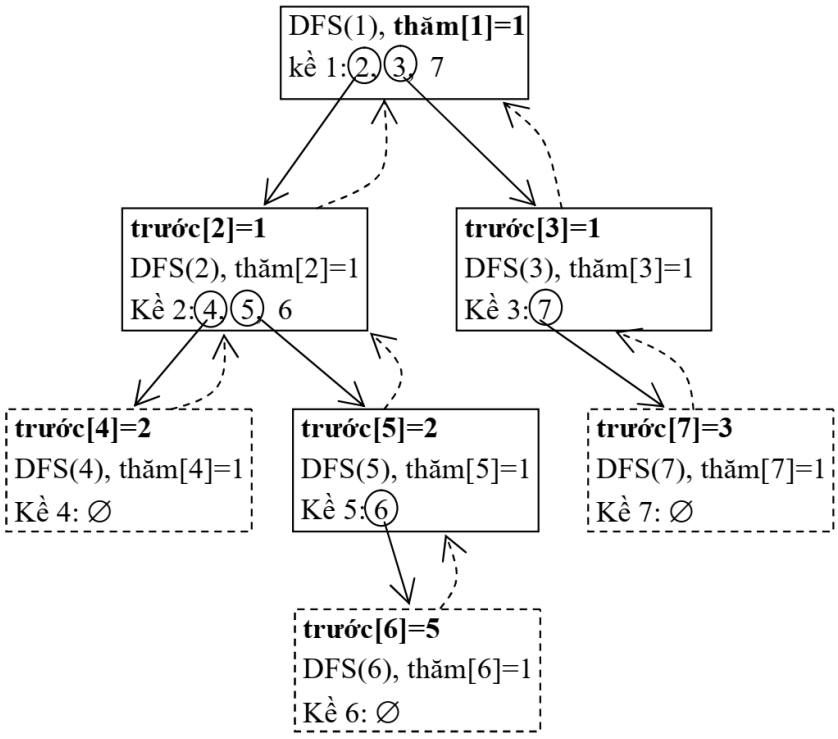
Dùng thuật toán BFS:

```
void BFS (int v){
	queue = ∅; // Khởi tạo hàng đợi
	visited[s] = true;
	push(queue,v) //Bỏ v vào stack

	while(queue != ∅){
		u = queue.top();
		pop(queue); //Xóa đỉnh ở đầu hàng đợi
		for (k : adj[u]){
			if(visited[k] = 0){
				visited[u]=true;
				trước[k]=u; //Lưu lại đỉnh trước của k là u
				push(queue, k);
			}
		}
	}
}
void main(){
	for (v ∈ V) visited[v] = false; // Khởi tạo cờ cho đỉnh
	BFS(v);
}
```

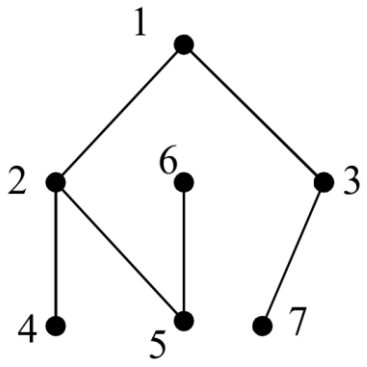
Ví dụ 1: Xét đồ thị G như sau:



- Dùng thuật toán DFS duyệt qua đồ thị có lưu lại đường đi với **đỉnh xuất phát là s=1**. Từng bước của thuật toán được mô tả trong sơ đồ sau:



- Khi kết thúc thuật toán, tất cả các đỉnh đều được thăm (thăm[v]=1, ∀v∈V) nên từ đỉnh xuất phát s=1 có đường đi đến tất cả các đỉnh còn lại.
- Đường đi từ đỉnh xuất phát s=1 đến các đỉnh khác được xác định dựa vào giá trị của mảng trước[v]: trước[2]=1, trước[3]=1, trước[4]=4, trước[5]=2, trước[6]=5, trước[7]=3. đường đi được thể hiện như sau:



3.6.2 Kiểm tra tính liên thông

- ***Bài toán:*** Tính số thành phần liên thông của đồ thị, và xác định những đỉnh thuộc cùng một thành phần liên thông.

- Phương pháp:

+ Sử dụng DFS và BFS

+ Biến inconnect *đếm số thành phần liên thông* của đồ thị.

+ Mảng index[v] *lưu chỉ số của các thành phần liên thông. (Lưu đỉnh v thuộc thành phần liên thông thứ mấy?).*

Dùng thuật toán DFS đệ quy:

```
void DFS(int v); {
	index[v] = inconnect;
	visited[v] = true;

	for ( u : adj[v] ){
		if ( !visited[u])
			DFS(u);
	}
}
void main() {
	for ( v ∈ V ) visited[v] = false; /* Khởi tạo cờ cho đỉnh */
	inconnect = 0;
	for ( v ∈ V ){
		if (visited[v] == false) {
			inconnect ++;
			DFS(v);
		}
	}
}
```

Ví dụ 2: Giả sử G = (V, E) có dữ liệu như sau: *V = {1, 2, 3, 4, 5, 6, 7}*

Adjacency List:

1: {2, 5}

2: {1, 5}

3: {4, 6, 7}

4: {3, 6, 7}

5: {1, 2}
6: {3, 4, 7}
7: {3, 4, 6}

// Khởi tạo visited[v] = false cho tất cả các đỉnh
visited = {false, false, false, false, false, false, false}

// Khởi tạo biến inconnect = 0 để đếm số thành phần liên thông
inconnect = 0

// Bắt đầu với mỗi đỉnh trong tập đỉnh V
1. DFS(1)
- Đỉnh 1 chưa được thăm → Tăng biến inconnect lên 1 (inconnect = 1) → Vào hàm DFS().
- Gán index[1] = 1. Nghĩa là đỉnh 1 thuộc thành phần liên thông (1).
- Đánh dấu đỉnh 1 là đã thăm (visited[1] = true).
- Xét các đỉnh kề với 1: 2, 5.

*** Gọi DFS(2)**
- Gán index[2] = 1. Đỉnh 2 thuộc thành phần liên thông (1).
- Đánh dấu đỉnh 2 là đã thăm (visited[2] = true)
- Xét các đỉnh kề với 2: 1, 5.
- Gọi DFS(1), nhưng đỉnh 1 đã được thăm rồi, nên không làm gì cả

*** Gọi DFS(5)**
- Gán index[5] = 1. Đỉnh 5 thuộc thành phần liên thông (1).
- Đánh dấu đỉnh 5 là đã thăm (visited[5] = true)
- Xét các đỉnh kề với 5: 1, 2.
- DFS(1) và DFS(2) đã được thực hiện, không còn đỉnh kề chưa thăm

2. DFS(3)
- Đỉnh 1 chưa được thăm → Tăng biến inconnect lên 1 (inconnect = 2) → Vào hàm DFS().
- Gán index[3] = 2. Nghĩa là đỉnh 3 thuộc thành phần liên thông (2).
- Đánh dấu đỉnh 3 là đã thăm (visited[3] = true)
- Xét các đỉnh kề với 3: 4, 6, 7.

*** Gọi DFS(4)**
- Gán index[4] = 2. Nghĩa là đỉnh 4 thuộc thành phần liên thông (2).
- Đánh dấu đỉnh 4 là đã thăm (visited[4] = true)
- Xét các đỉnh kề với 4: 3, 6, 7.
- Gọi DFS(3), nhưng đỉnh 3 đã được thăm rồi, nên không làm gì cả

*** Gọi DFS(6)**
- Gán index[4] = 2. Nghĩa là đỉnh 4 thuộc thành phần liên thông (2).
- Đánh dấu đỉnh 6 là đã thăm (visited[6] = true)
- Xét các đỉnh kề với 4: 3, 4, 7.

- Gọi DFS(3), nhưng đỉnh 3 đã được thăm rồi, nên không làm gì cả
- Gọi DFS(4), nhưng đỉnh 4 đã được thăm rồi, nên không làm gì cả

*** Gọi DFS(7)**

- Gán index[7] = 2. Nghĩa là đỉnh 7 thuộc thành phần liên thông (2).
- Đánh dấu đỉnh 7 là đã thăm (visited[7] = true)
- Xét các đỉnh kề với 7: 3, 4, 6.

- Gọi DFS(3), nhưng đỉnh 3 đã được thăm rồi, nên không làm gì cả
- Gọi DFS(4), nhưng đỉnh 4 đã được thăm rồi, nên không làm gì cả
- Gọi DFS(6), nhưng đỉnh 6 đã được thăm rồi, nên không làm gì cả

3. Tất cả visited[v] = true nên dừng vòng lặp và dừng chương trình.

→ Số thành phần liên thông = inconnect = 2

→ Mỗi đỉnh sẽ được gắn nhãn là thuộc thành phần liên thông nào:

1: 1
2: 1
3: 2
4: 2
5: 1
6: 2
7: 2

Dùng thuật toán BFS:

```
void BFS (int v)
{
	queue q = ∅;
	visited[v] = true;
	q.push(v);

	while (q.Count != 0) { //Khi hàng đợi chưa rỗng
		p = q.front();  //Truy cập vào đỉnh ở đầu hàng đợi
		p.pop ();  //Xóa đỉnh ở đầu hàng đợi

		index[p] = inconnect;

		for (u ∈ Ke(p)){
			if (visited[u] == false){
				visited[u] = true;
				q.push(u) ;
			}
		}
	}
}

void main() {
	for ( v ∈ V ) visited[v] = 0;
	inconnect = 0;
	for ( v ∈ V )
		if ( visited[v] == 0 ) {
			inconnect++;
			BFS(v);
	}
}
```

**** Cách hoạt động của BFS gần như tương tự DFS.***