SẮP XẾP TOPO

NỘI DUNG BÀI HỌC

01 Sắp xếp Topo

02 Thành phần liên thông mạnh

03 Ứng dụng của Tarjan cho bài toán đỉnh trụ, cạnh cầu

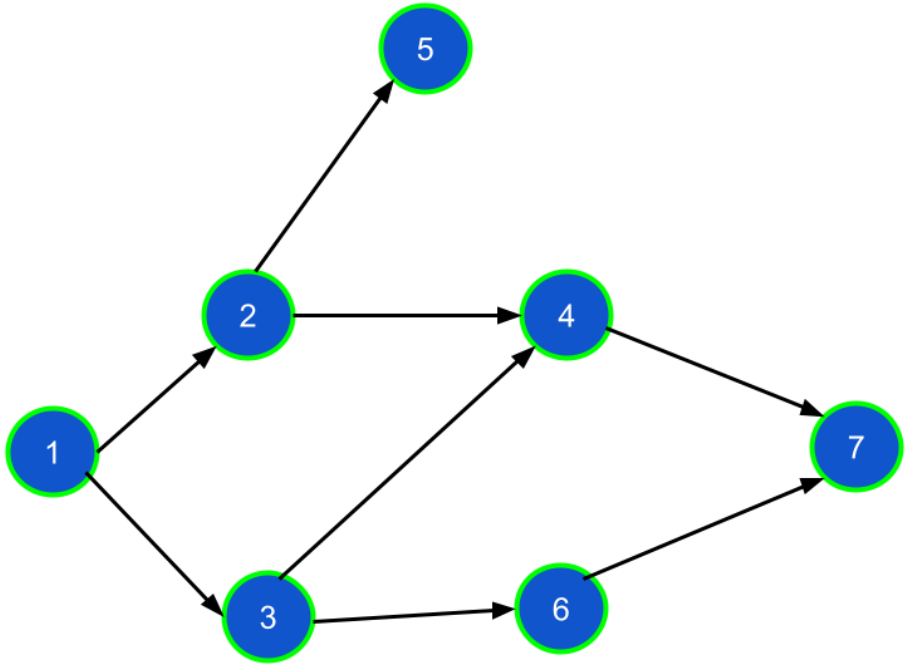
1. Sắp xếp Topo (Topological Sorting):

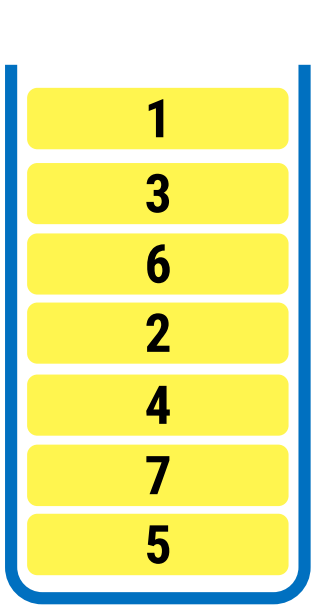
Sắp xếp topo áp dụng cho đồ thị có hướng không có chu trình (DAG - Directed Acyclic Graph) là một thứ tự của các đỉnh sao cho với mọi cạnh u -> v thì u sẽ xuất hiện trước u trong thứ tự topo. Một đồ thị có thể tồn tại nhiều thứ tự topo khác nhau.

1. Sắp xếp Topo (Topological Sorting):

a. Sắp xếp Topo với DFS:

Sắp xếp topo sử dụng DFS dựa trên thứ tự khi duyệt xong một đỉnh của đồ thị, khi một đỉnh được duyệt xong ta đẩy đỉnh này vào trong ngăn xếp. Thứ tự sắp xếp topo chính là các đỉnh nằm trong ngăn xếp tính từ đỉnh.





1. Sắp xếp Topo (Topological Sorting):

a. Sắp xếp Topo với DFS:

Code

INPUT

```
7 9
1 2
1 3
2 5
2 4
3 4
3 6
4 7
6 7
```

```cpp
int n, m;
vector<int> adj[1005];
bool visited[1005];
stack<int> st;

void nhap(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        int x, y; cin >> x >> y;
        adj[x].push_back(y);
    }
    memset(visited, false, sizeof(visited));
}
```

OUTPUT

```
1 3 6 2 4 7 5
```

```cpp
void DFS(int u){
    visited[u] = true;
    for(int v : adj[u]){
        if(!visited[v]) DFS(v);
    }
    st.push(u);
}
int main(){
    nhap();
    for(int i = 1; i <= n; i++){
        if(!visited[i]) DFS(i);
    }
    while(!st.empty()){
        cout << st.top() << ' ';
        st.pop();
    }
}
```

1. Sắp xếp Topo (Topological Sorting):

b. Sắp xếp Topo với BFS:

Sắp xếp topo với BFS hay thuật toán Kahn, thuật toán xóa dần đỉnh.

Thuật toán:

Bước 1: Tính bán bậc vào của mọi đỉnh trên đồ thị.

Bước 2: Đưa các đỉnh có bán bậc vào bằng 0 vào hàng đợi.

Bước 3: Duyệt và xóa đỉnh khỏi đầu hàng đợi, duyệt các đỉnh kề với đỉnh này và giảm bán bậc vào của các đỉnh kề, nếu đỉnh nào sau khi giảm có bán bậc vào bằng 0 thì tiếp tục đẩy vào hàng đợi.

Bước 4: Lặp lại bước 3 cho tới khi hàng đợi còn phần tử

Bước 5: Nếu số lượng đỉnh được duyệt bằng với số đỉnh của đồ thị thì đó chính là thứ tự topo, ngược lại ta có thể suy ra đồ thị tồn tại chu trình.

1. Sắp xếp Topo (Topological Sorting):

b. Sắp xếp Topo với BFS:

Code

```
int n, m;
vector<int> adj[1005];
int degree[1005];

void nhap(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        int x, y; cin >> x >> y;
        adj[x].push_back(y);
        degree[y]++;
    }
}

void Kahn(){
    queue<int> q;
    for(int i = 1; i <= n; i++){
        if(degree[i] == 0) q.push(i);
    }
    vector<int> topo;
    while(!q.empty()){
        int u = q.front(); q.pop();
        topo.push_back(u);
        for(int v : adj[u]){
            degree[v]--;
            if(degree[v] == 0) q.push(v);
        }
    }
    if (topo.size() < n) cout << "Do thi co chu trinh !\n";
    else{
        for(int x : topo)
            cout << x << ' ';
    }
}

int main(){
    nhap();
    Kahn();
}
```

INPUT

```
7 9
1 2
1 3
2 5
2 4
3 4
3 6
4 7
6 7
```

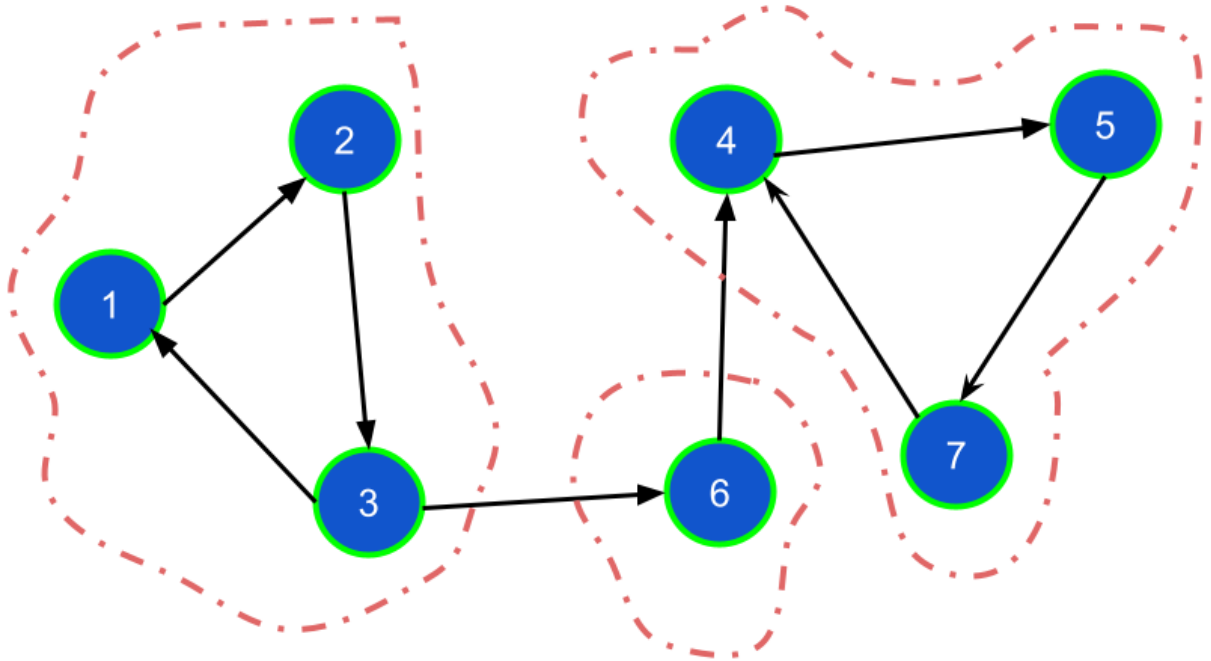
OUTPUT

```
1 2 3 5 4 6 7
```

2. Thành phần liên thông mạnh:

Đồ thị có hướng liên thông mạnh nếu giữa 2 đỉnh bất kì của đồ thị đều có đường đi. Nếu đồ thị không liên thông mạnh nó sẽ chia thành các thành phần liên thông mạnh (Strongly connected component - SCC). **Thành phần liên thông mạnh** là thành phần liên thông lớn nhất mà giữa 2 đỉnh của nó đều có đường đi.

Ví dụ 1: Đồ thị có 3 thành phần liên thông mạnh (1, 2, 3), (6), (4, 5, 7).



2. Thành phần liên thông mạnh:

a. Thuật toán Kosaraju:

Thuật toán Kosaraju có thể dùng để liệt kê các thành phần liên thông mạnh của đồ thị và vì thế cũng có thể sử dụng để kiểm tra đồ thị liên thông mạnh. Độ phức tạp của Kosaraju tương tự như DFS: O(V + E)

2. Thành phần liên thông mạnh:

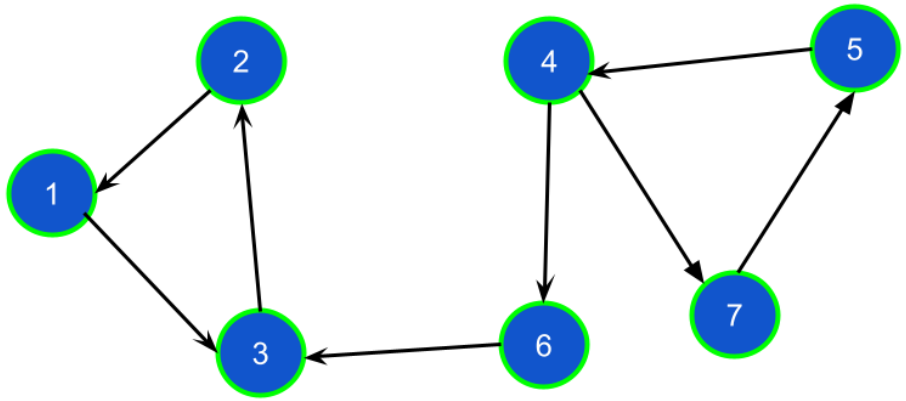
a. Thuật toán Kosaraju:

Thuật toán:

Bước 1: Tạo một ngăn xếp rỗng để lưu thứ tự duyệt xong các đỉnh của đồ thị, đỉnh nào được duyệt xong trước sẽ được đẩy vào ngăn xếp. (Tương tự như topo)

Bước 2: Lật ngược hướng của các cạnh trên đồ thị ban đầu để thu được đồ thị chuyển vị (transpose graph)

Bước 3: Lần lượt lấy các đỉnh trong ngăn xếp và nếu đỉnh này được chưa được duyệt trong các thành phần liên thông mạnh trước đó thì lấy đỉnh này làm đỉnh nguồn và gọi thuật toán DFS (trên đồ thị transpose ở bước 2) để in ra các đỉnh trong thành phần liên thông.



Đồ thị chuyển vị (transpose graph) của đồ thị ban đầu

2. Thành phần liên thông mạnh:

a. Thuật toán Kosaraju:

Code

INPUT

```
7 8
1 2
2 3
3 1
3 6
6 4
4 5
5 7
7 4
```

```cpp
int n, m;
vector<int> adj[1005], t_adj[1005];
bool visited[1005];
stack<int> st;
void nhap(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        int x, y; cin >> x >> y;
        adj[x].push_back(y); // graph
        t_adj[y].push_back(x); // transpose graph
    }
    memset(visited, false, sizeof(visited));
}
void DFS1(int u){
    visited[u] = true;
    for(int v : adj[u]){
        if(!visited[v]) DFS1(v);
    }
    st.push(u);
}
void DFS2(int u){
    visited[u] = true;
    cout << u << ' ';
    for(int v : t_adj[u]){
        if(!visited[v]) DFS2(v);
    }
}
void Kosaraju(){
    for(int i = 1; i <= n; i++){
        if(!visited[i]) DFS1(i);
    }
    memset(visited, false, sizeof(visited));
    int scc = 0;
    while(!st.empty()){
        int u = st.top(); st.pop();
        if(!visited[u]){
            ++scc;
            cout << "SCC " << scc << " : ";
            DFS2(u); cout << endl;
        }
    }
}
int main(){
    nhap();
    Kosaraju();
}
```

OUTPUT

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

2. Thành phần liên thông mạnh:

b. Thuật toán Tarjan:

Tarjan là một thuật toán giúp liệt kê, đếm thành phần liên thông của đồ thị với độ phức tạp là O(V + E), Tarjan chỉ cần 1 lần duyệt DFS là có thể liệt kê được SCC thay vì 2 lần như Kosaraju.

Để hiểu được thuật toán Tarjan trước hết bạn cần hiểu được **ý nghĩa của 2 mảng là disc[] và low[]** được tính toán dựa trên thứ tự duyệt các đỉnh trên đồ thị bằng thuật toán DFS.

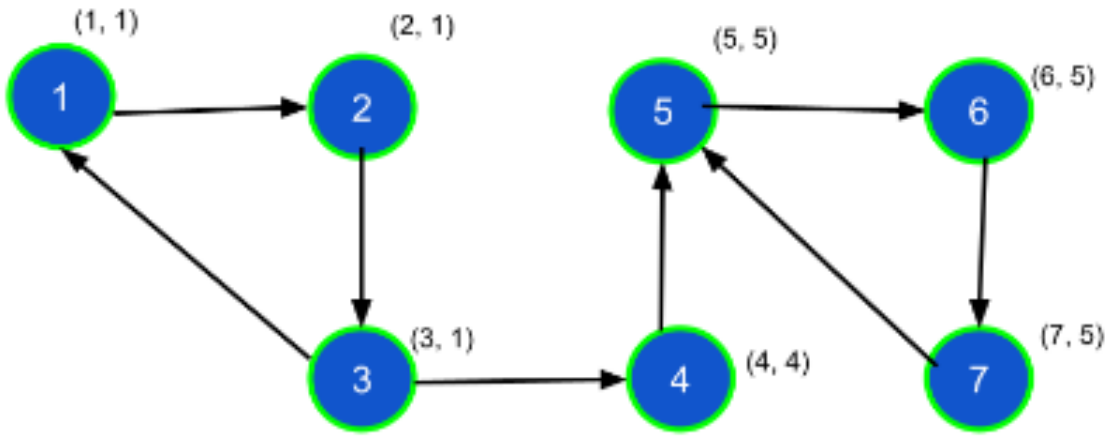
2. Thành phần liên thông mạnh:

b. Thuật toán Tarjan:

Ý nghĩa của hai mảng disc[ ] và low[ ]

Mảng disc[u]
Chỉ ra thời gian bắt đầu thăm đỉnh u theo thuật toán DFS.

Mảng low[u]
Chỉ ra thời gian thăm sớm nhất của một đỉnh có thể đi tới được từ một cây con có gốc là u.



vectice	1	2	3	4	5	6	7
disc	1	2	3	4	5	6	7
low	1	1	1	4	5	5	5

2. Thành phần liên thông mạnh:

b. Thuật toán Tarjan:

Code xây dựng mảng disc và low cho các đỉnh trên đồ thị

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

using ll = long long;

int n, m, timer = 0;
vector<int> adj[1005];
int disc[1005], low[1005];
bool visited[1005];

void nhap(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        int x, y; cin >> x >> y;
        adj[x].push_back(y);
    }
}
```

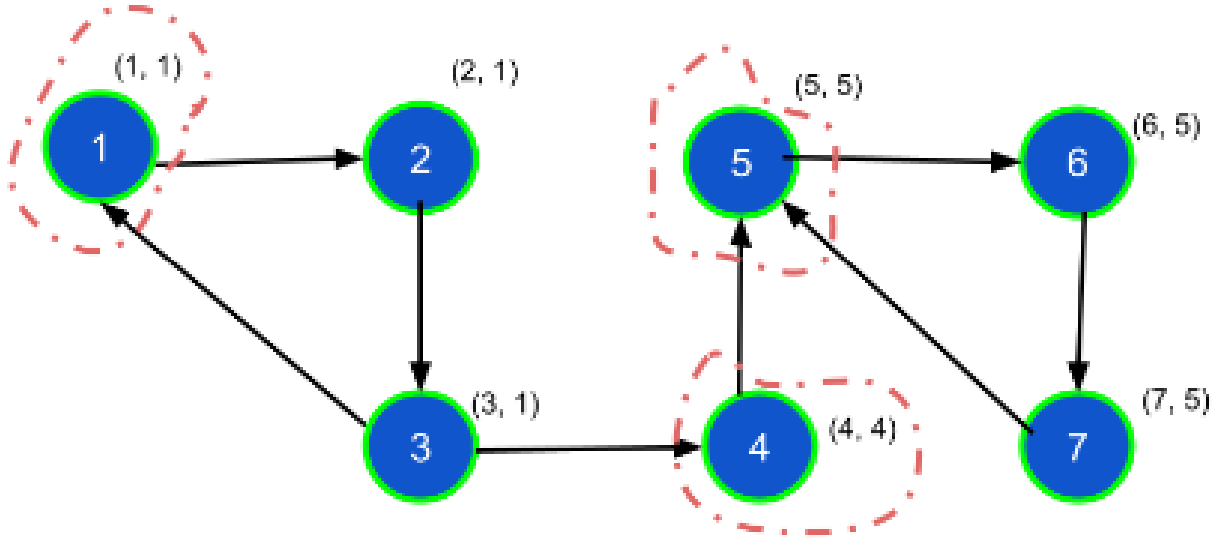
```
void DFS(int u){
    visited[u] = true;
    disc[u] = low[u] = ++timer;
    for(int v : adj[u]){
        if(!visited[v]){
            DFS(v);
            low[u] = min(low[u], low[v]);
        }
        else{
            low[u] = min(low[u], disc[v]);
        }
    }
}
```

```
int main(){
    nhap();
    for(int i = 1; i <= n; i++){
        if(!visited[i]){
            DFS(i);
        }
    }
    for(int i = 1; i <= n; i++){
        cout << disc[i] << ' ' << low[i];
        cout << endl;
    }
}
```

2. Thành phần liên thông mạnh:

b. Thuật toán Tarjan:

Thuật toán Tarjan dựa trên quan sát, những đỉnh u thỏa mãn disc[u] = low[u] sẽ là đỉnh bắt đầu của một thành phần liên thông mạnh.



vectice	1	2	3	4	5	6	7
disc	1	2	3	4	5	6	7
low	1	1	1	4	5	5	5

2. Thành phần liên thông mạnh:

b. Thuật toán Tarjan:

Code

INPUT

```
7 8
1 2
2 3
3 1
3 4
4 5
5 6
6 7
7 5
```

```cpp
int n, m, timer = 0, scc = 0;
vector<int> adj[1005];
int disc[1005], low[1005];
bool visited[1005], in_stack[1005];
stack<int> st;

void nhap(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        int x, y; cin >> x >> y;
        adj[x].push_back(y);
    }
    memset(visited, false, sizeof(visited));
    memset(in_stack, false, sizeof(visited));
}
```

```cpp
void DFS(int u){
    visited[u] = true; in_stack[u] = true;
    disc[u] = low[u] = ++timer;
    st.push(u);
    for(int v : adj[u]){
        if(!visited[v]){
            DFS(v);
            low[u] = min(low[u], low[v]);
        }
        else{
            low[u] = min(low[u], disc[v]);
        }
    }
    if(low[u] == disc[u]){
        ++scc;
        while(st.top() != u){
            cout << st.top() << ' ';
            in_stack[st.top()] = false;
            st.pop();
        }
        cout << st.top() << endl;
        in_stack[st.top()] = false;
        st.pop();
    }
}
```

```cpp
int main(){
    nhap();
    for(int i = 1; i <= n; i++){
        if(!visited[i]){
            DFS(i);
        }
    }
    cout << scc << endl;
}
```

OUTPUT

```
7 6 5
4
3 2 1
3
```

2. Tìm đỉnh trụ, cạnh cầu với Tarjan:

a. Đỉnh trụ:

Ý tưởng:

Cho đồ thị G vô hướng, trên đồ thị có 2 đỉnh U và V, giữa U và V đang có đường đường đi từ U tới V theo thứ tự duyệt DFS, nếu ta có thể từ V đi tới tổ tiên A của U mà không cần thông qua U điều đó chứng tỏ đồ thị có cạnh ngược và việc loại bỏ U khỏi đồ thị không làm tăng số thành phần liên thông của đồ thị, khi đó U không phải là đỉnh trụ (Articulation point)

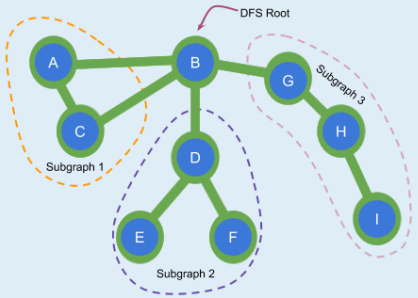
2. Tìm đỉnh trụ, cạnh cầu với Tarjan:

a. Đỉnh trụ:

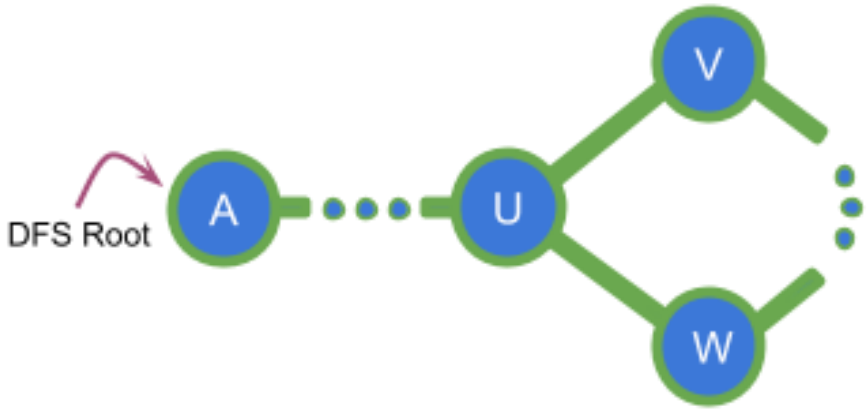
Có 2 trường hợp để U là đỉnh trụ trên đồ thị

Tất cả đường đi từ A tới V đều phải đi qua U

U là gốc của cây DFS với ít nhất 2 node con



Vì thế điều kiện để đỉnh U là đỉnh trụ là: disc[u] <= low[v]. Trong đó disc[u] < low[v] thể hiện rằng từ v không thể tìm được đường đi nào khác tới tổ tiên của u mà không thông qua u. disc[u] = low[v] trong trường hợp u là gốc của chu trình chứa v.



2. Tìm đỉnh trụ, cạnh cầu với Tarjan:

a. Đỉnh trụ:

Code

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

using ll = long long;

int n, m, timer = 0;
vector<int> adj[1005];
int disc[1005], low[1005];
bool visited[1005], AP[1005];

void nhap(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        int x, y; cin >> x >> y;
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    memset(visited, false, sizeof(visited));
    memset(visited, false, sizeof(AP));
}
```

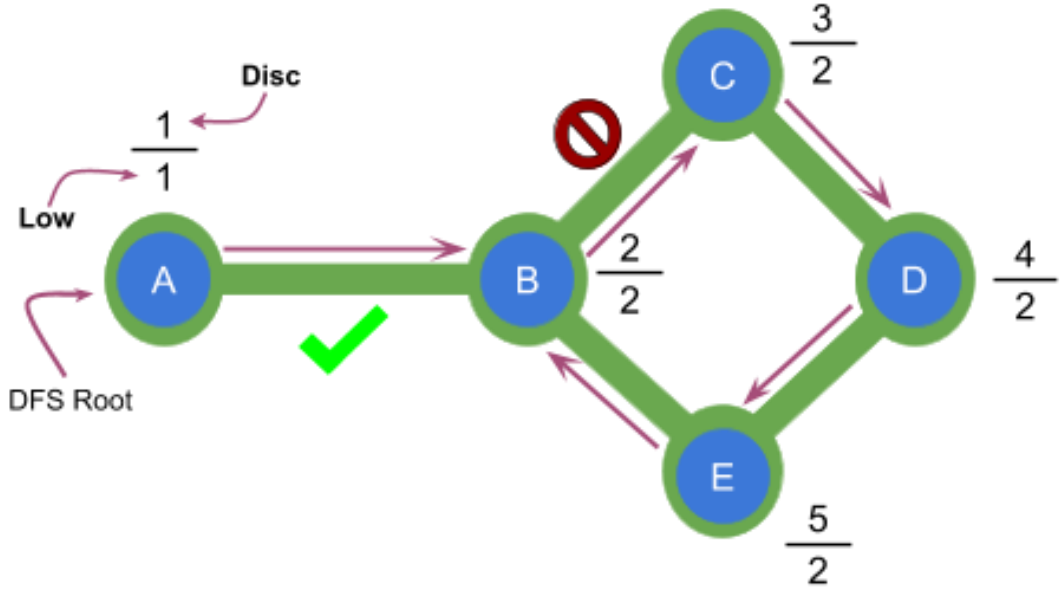
```
void DFS(int u, int par){
    visited[u] = true;
    disc[u] = low[u] = ++timer;
    int child = 0;
    for(int v : adj[u]){
        if(v == par) continue;
        if(!visited[v]){
            DFS(v, u);
            ++child;
            low[u] = min(low[u], low[v]);
            if(par != -1 && disc[u] <= low[v])
                AP[u] = true;
            }
        }
        else{
            low[u] = min(low[u], disc[v]);
        }
    }
    if(par == -1 && child > 1) AP[u] = true;
}
```

```
int main(){
    nhap();
    for(int i = 1; i <= n; i++){
        if(!visited[i]){
            DFS(i, -1);
        }
    }
    for(int i = 1; i <= n; i++){
        if(AP[i]) cout << i << ' ';
    }
```

2. Tìm đỉnh trụ, cạnh cầu với Tarjan:

b. Cạnh cầu:

Điều kiện để cạnh U, V là cạnh cầu: disc[u] < low[v], trong trường hợp này không tồn tại dấu bằng, vì nếu u là gốc của chu trình chứa v thì loại bỏ u, v sẽ không làm tăng số thành phần liên thông của đồ thị. Ví dụ hình dưới đấy (B, C) sẽ không là cạnh cầu, nhưng B lại là đỉnh trụ



2. Tìm đỉnh trụ, cạnh cầu với Tarjan:

b. Cạnh cầu:

Code

```
typedef pair<int, int> ii;
int n, m, timer = 0;
vector<int> adj[1005];
int disc[1005], low[1005];
bool visited[1005];
vector<ii> bridge;

void nhap(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        int x, y; cin >> x >> y;
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    memset(visited, false, sizeof(visited));
}
```

```
void DFS(int u, int par){
    visited[u] = true;
    disc[u] = low[u] = ++timer;
    for(int v : adj[u]){
        if(v == par) continue;
        if(!visited[v]){
            DFS(v, u);
            low[u] = min(low[u], low[v]);
            if(disc[u] < low[v]){
                bridge.push_back({u, v});
            }
        }
        else{
            low[u] = min(low[u], disc[v]);
        }
    }
}
```

```
int main(){
    nhap();
    for(int i = 1; i <= n; i++){
        if(!visited[i]){
            DFS(i, -1);
        }
    }
    for(ii e : bridge){
        cout << e.first << ' ' <<
e.second << endl;
    }
}
```