

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG



BÁO CÁO THỰC HÀNH
MÔN HỌC: THỰC TẬP CƠ SỞ
Đề tài: Một số thuật toán liên quan đến đồ thị

Họ và tên: Nguyễn Đức Trí

Lớp: E22CQCN05-B

MSV: B22DCAT302

Số điện thoại: 0914760804

Hà Nội, Tháng 5/2025

Mục lục

Tự đánh giá của bản thân:	6
Bản báo cáo	7
Mô tả đề tài	7
Nội dung kiến thức tìm hiểu	8
1. Thuật toán sắp xếp topo	11
Khái niệm	11
Đặt vấn đề	11
Độ phức tạp	15
Ứng dụng	15
Bài toán	15
2. DSU và ứng dụng	17
Khái niệm	17
Độ phức tạp	18
Ứng dụng	18
Bài toán 1	18
Bài toán 2	20
3. Thuật toán Boruvka	22
Khái niệm	22
Cách hoạt động	22
Độ phức tạp	23
Ví dụ minh họa	23
So sánh ưu nhược điểm so với các thuật toán khác (Kruskal, Prim)	25
4. Thuật toán Kosaraju	26
Khái niệm	26
Code	26
Độ phức tạp	27

Demo	28
Tài liệu tham khảo	28
5. Thuật toán Tarjan	29
Khái niệm	29
Bài toán tính số cạnh cầu (bridge), đỉnh trụ (khớp : articulation point) trên đồ thị vô hướng	31
Khái niệm	31
Code	33
Thử nghiệm	34
Độ phức tạp.....	35
Tính số thành phần liên thông mạnh (SCC)	36
Tóm tắt ý tưởng :	36
Code	37
Độ phức tạp.....	39
So sánh với kosaraju	39
Tài liệu tham khảo	39
6. Luồng.....	40
Các định nghĩa	40
Ứng dụng luồng cực đại	41
I. Ứng dụng thực tế	41
II. Ứng dụng trong lý thuyết và thuật toán	42
Ví dụ cụ thể.....	42
7. Thuật toán Edmonds-Karp.....	43
“Đôi lời về lịch sử thuật toán	43
Thuật toán+ code	43
Giải thích chi tiết.....	46
Demo	50

Độ phức tạp thuật toán	50
Tài liệu tham khảo	50
8. Bài tập liên quan đến thuật toán Edmonds-Karp.....	51
Bài 1.....	51
Bài 2.....	52
9. Thuật toán Dinic.....	61
Khái niệm	61
Ý tưởng thuật toán Dinic	61
Demo	65
Độ phức tạp	65
10. Bài tập liên quan đến thuật toán Dinic	66
Bài 1.....	66
Bài 2.....	67
Tài liệu tham khảo	69
Link Github	69

Dự kiến công việc:

Tuần	Nội dung dự kiến	Chi tiết
1	Ôn lại và tìm hiểu sâu lại về các thuật toán đã học	Dfs,bfs, sử dụng bfs, dfs để tìm số thành phần liên thông, cầu, khớp,tìm đường đi, vv..
2	Thuật toán sắp xếp topo	Là thuật toán tìm một thứ tự sắp xếp các đỉnh trong đồ thị có hướng không có chu trình sao cho đảm bảo đỉnh u luôn đứng trước v nếu có cung từ u đến v
3	DSU và ứng dụng	Là một cấu trúc dữ liệu hỗ trợ hai thao tác hiệu quả: hợp nhất hai tập hợp và tìm đại diện của một phần tử/ đỉnh trên đồ thị
4	Thuật toán boruvka, ưu/nhược điểm so với Kruskal, prim	Tìm kích thước cây khung nhỏ nhất
5	Thuật toán Kosaraju	tìm số thành phần liên thông mạnh trên đồ thị có hướng
6	Thuật toán tarjan	tìm khớp, cầu của đồ thị
7	Tìm hiểu khái niệm về luồng, luồng cực đại	
8	thuật toán Edmonds-Karp	tìm luồng cực đại
9	Bài tập liên quan đến thuật toán Edmonds-Karp	tìm luồng cực đại
10	Thuật toán Dinic	tìm luồng cực đại
11	Bài tập liên quan đến thuật toán thuật toán Dinic	tìm luồng cực đại
12	Ôn lại các thuật toán, hoàn thành các việc liên quan đến báo cáo, vv...	

Tự đánh giá của bản thân:

Tuần	Chi tiết	Điểm
1	Đã hoàn thành những gì đề ra trong kế hoạch	10/10
2	Đã hoàn thành những gì đề ra trong kế hoạch	10/10
3	Đã hoàn thành những gì đề ra trong kế hoạch	10/10
4	Đã hoàn thành những gì đề ra trong kế hoạch, tuy nhiên chưa làm được nhiều bài tập để nhuần nhuyễn	9/10
5	Đã hoàn thành những gì đề ra trong kế hoạch, tuy nhiên quên không nộp đúng hạn	4/10
6	Đã hoàn thành những gì đề ra trong kế hoạch, tuy nhiên chưa thực sự tìm hiểu sâu về ứng dụng mở rộng của thuật toán	8/10
7	Tìm hiểu khái niệm về luồng, luồng cực đại	8/10
8	thuật toán Edmonds-Karp	10/10
9	Bài tập liên quan đến thuật toán Edmonds-Karp	10/10
10	Thuật toán Dinic	10/10
11	Bài tập liên quan đến thuật toán thuật toán Dinic	8/10
12	Ôn lại các thuật toán, hoàn thành các việc liên quan đến báo cáo, vv...	10/10
Tổng kết	Đã làm đầy đủ so với kế hoạch	9.5/10

Bản báo cáo

Mô tả đề tài

Tên đề tài: Một số thuật toán liên quan đến đồ thị

Mục tiêu:

- Hệ thống hóa và thực hành cài đặt các thuật toán cơ bản và nâng cao trong lý thuyết đồ thị.
- Áp dụng các thuật toán vào bài toán thực tế như tìm đường đi, phân tích mạng, luồng cực đại, phân rã đồ thị, v.v.
- Tìm hiểu ưu – nhược điểm, so sánh hiệu quả và ứng dụng của các thuật toán.

Phạm vi thực hiện:

- Thuật toán duyệt đồ thị (DFS, BFS)
- Thuật toán tìm thành phần liên thông (Kosaraju, Tarjan)
- Sắp xếp tô pô
- DSU và các bài toán liên quan đến thành phần liên thông
- Tối ưu cây khung nhỏ nhất (Kruskal, Boruvka)
- Thuật toán luồng cực đại (Edmonds-Karp, Dinic)
- Bài tập thực hành trên các trang online judge

Nội dung kiến thức tìm hiểu

2.1. Sắp xếp tôpô (Topological Sort)

- **What:** Là sắp xếp các đỉnh của đồ thị có hướng không chu trình (DAG) sao cho nếu có cạnh từ u đến v thì u đứng trước v trong thứ tự.
- **Why:** Cần thiết để giải các bài toán lập lịch, phụ thuộc dữ liệu, chuỗi tác vụ, đặc biệt khi có thứ tự ưu tiên.
- **When:** Sử dụng khi đồ thị là DAG và cần xác định thứ tự thực hiện hợp lệ.
- **Where:** Áp dụng trong trình biên dịch (xác định thứ tự biên dịch module), phân môn học (môn tiên quyết), giải bài toán từ điển ngoài hành tinh.
- **How:** Cài đặt bằng DFS với trạng thái 0-1-2 hoặc thuật toán BFS Kahn. DFS đơn giản và hiệu quả hơn trong nhiều bài toán.

2.2. DSU (Disjoint Set Union)

- **What:** Cấu trúc dữ liệu quản lý tập hợp rời rạc, với 2 thao tác chính: tìm đại diện (Find) và hợp nhất (Union).
- **Why:** Giúp phát hiện thành phần liên thông, kiểm tra chu trình, xác định cây khung trong Kruskal.
- **When:** Khi cần thao tác nhiều trên tập hợp đỉnh – thường xuyên tách/gộp.
- **Where:** Áp dụng trong đồ thị vô hướng, thuật toán Kruskal, bài toán tìm tổ tiên chung, quản lý tập hợp rời rạc.
- **How:** Dùng mảng `parent[]`, tối ưu bằng nén đường đi và gộp theo kích thước (hoặc độ sâu), giúp tăng tốc độ.

2.3. Thuật toán Boruvka

- **What:** Là thuật toán tìm cây khung nhỏ nhất (MST) bằng cách chọn cạnh rẻ nhất từ mỗi thành phần liên thông.
- **Why:** Là thuật toán cổ điển và có hiệu quả khi đồ thị dày.
- **When:** Khi muốn tìm cây khung với DSU mà không cần sắp xếp toàn bộ cạnh như Kruskal.

- **Where:** Dùng cho đồ thị vô hướng, nhiều cạnh, nơi tốc độ sắp xếp là vấn đề.
- **How:** Duyệt toàn bộ cạnh, cập nhật các cạnh "rẻ nhất", dùng DSU để gộp các thành phần liên thông, lặp lại.

2.4. Thuật toán Kosaraju

- **What:** Tìm các thành phần liên thông mạnh (SCC) trong đồ thị có hướng.
- **Why:** Giúp phân rã đồ thị, tìm chu trình mạnh, dùng trong phân tích mạng, kiểm tra ràng buộc.
- **When:** Khi cần phân tích các liên kết hai chiều mạnh.
- **Where:** Ứng dụng trong phân tích mã máy, kiểm tra phụ thuộc hệ thống, thuật toán ghép nối ràng buộc.
- **How:** Duyệt DFS theo thứ tự gốc, rồi đảo cạnh và duyệt ngược bằng stack, đánh số các SCC.

2.5. Thuật toán Tarjan

- **What:** Tìm cầu (bridge), khớp (articulation point), và SCC.
- **Why:** Dễ cài đặt hơn Kosaraju, hiệu quả cho đồ thị vô hướng.
- **When:** Khi cần xác định điểm yếu của mạng (khi bị cắt ra làm nhiều phần).
- **Where:** Phân tích mạng, xác định điểm cần dự phòng trong mạng máy tính, giao thông.
- **How:** Dùng DFS gán nhãn num[] và low[], xác định đỉnh/cạnh quan trọng bằng so sánh chỉ số.

2.6. Khái niệm về luồng & luồng cực đại

- **What:** Luồng là lượng dữ liệu/nước/dòng có thể chảy qua mạng từ nguồn đến đích, bị giới hạn bởi dung lượng.
- **Why:** Dùng để tối ưu hóa mạng phân phối, giao thông, năng lượng.
- **When:** Khi có hệ thống ràng buộc theo kiểu đầu-cuối và dung lượng.
- **Where:** Mạng điện, cấp thoát nước, logistics, ghép cặp bài toán.

- **How:** Mỗi cạnh có dung lượng, tuân thủ định luật bảo toàn luồng, tổng in = tổng out tại mỗi đỉnh (trừ source/sink).

2.7. Thuật toán Edmonds-Karp

- **What:** Phiên bản cụ thể của Ford-Fulkerson dùng BFS để tìm đường tăng luồng.
- **Why:** Tránh trường hợp luồng nhỏ \rightarrow vô hạn vòng lặp như Ford-Fulkerson gốc.
- **When:** Khi cần cài đặt đơn giản và dễ hiểu, với đồ thị vừa phải.
- **Where:** Bài toán ghép cặp, mạng đơn giản, dạy học.
- **How:** Tìm đường tăng bằng BFS, đẩy luồng theo đường đó, cập nhật mạng dư (residual), lặp lại.

2.8. Thuật toán Dinic

- **What:** Cải tiến của Ford-Fulkerson, sử dụng đồ thị tầng và DFS để đẩy luồng theo từng pha.
- **Why:** Hiệu quả hơn Edmonds-Karp trên đồ thị lớn, đặc biệt là đồ thị nhiều cạnh.
- **When:** Khi cần giải bài toán lớn hơn, tối ưu hơn.
- **Where:** Bài toán tổ chức, phân công, tối ưu hệ thống trong thực tế và lý thuyết.
- **How:** Xây dựng đồ thị tầng bằng BFS \rightarrow tìm luồng tăng bằng DFS \rightarrow cập nhật mạng dư \rightarrow lặp lại.

1. Thuật toán sắp xếp topo

Khái niệm

Sắp xếp topo là quá trình sắp xếp các đỉnh của đồ thị có hướng mà không có chu trình (**DAG**) sao cho đối với mọi cạnh $u \rightarrow v$, đỉnh u đứng trước đỉnh v .

Ứng dụng: Nó có thể dùng để giải quyết các vấn đề liên quan đến lập lịch, giải quyết các vấn đề liên quan đến mã hoá, quy tắc nào đó.

Tất nhiên, vì là DAG nên nó không thể hoạt động trong các tình huống xung đột lịch trình, vv...

Trong đó:

Mọi đồ thị vô hướng không chu trình đều tồn tại **ít nhất một** thứ tự topo.

Có thể có nhiều hơn 1 thứ tự topo.

Thuật toán:

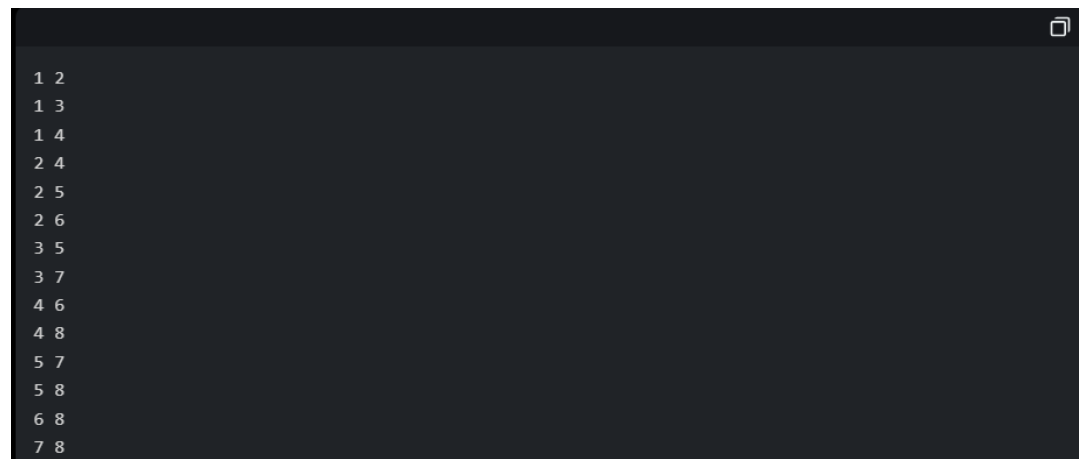
Ta có thể xây dựng nó dựa trên DFS/BFS (thuật toán Kahn). 2 cách này đều có độ phức tạp tương đương nhau, tuy nhiên, dfs có lợi thế vì code đơn giản hơn. Do vậy, em sẽ bỏ qua BFS và trình bày bằng DFS.

Đặt vấn đề:

Từ một đồ thị có hướng không chu trình cho trước, đưa ra 1 thứ tự topo.

Nếu có nhiều thứ tự, đưa ra bất kì cái nào cũng được.

Đồ thị:



Dưới dạng danh sách kề:

```
1: [2, 3, 4]
2: [4, 5, 6]
3: [5, 7]
4: [6, 8]
5: [7, 8]
6: [8]
7: [8]
8: []
```

Giải quyết vấn đề:

Khởi tạo:

```
vector<ll> rev_topo;
ll check[10000];

vector<ll> a[n+1];
f(i,1,n) check[i]=0;
f(i,1,m){
    ll c,d;
    cin>>c>>d;
    a[c].push_back(d);
}
f(i,1,n) {
    if(check[i]==0) topo(i,a);
}
```

Trong đó:

- n là số đỉnh, m là số cạnh
- a là mảng vector lưu danh sách kề mỗi đỉnh
- check là mảng để kiểm tra xem đỉnh đó là đỉnh đã xét(2), chưa xét(0) hay đang xét(1), ban đầu tất cả đều chưa xét
- rev_topo là vector lưu thứ tự topo. Trong đó ban đầu nó bị đảo ngược.

Hàm topo:

```
bool topo(int i, vector<int> a){
    check[i]=1;
    for(auto x:a[i]){
        if(check[x]==1){
            return false; // có chu trình
        }
        else if(check[x]==0){
            check[x]=1;
            bool tmp=topo(x,a);
            if(tmp==false) return false; // có chu trình
        }
    }
    check[i]=2;
    rev_topo.push_back(i);
    return true;
}
```

Giải thích:

Với mảng check:

Bảng 0: đỉnh này chưa được xét tới

Bảng 1: hàm topo của đỉnh này đang được xét (nếu đỉnh kề nó cũng có check =1 thì có chu trình)

Bảng 2: đã xét, không thể tạo ra chu trình nào nữa.

Với hàm topo():

false nếu có chu trình

True nếu không

Main:

Đây có thể là đồ thị không liên thông, nên bắt buộc phải kiểm tra mọi đỉnh.

Nếu phát hiện có chu trình thì dừng lại

Sau đó ta đảo ngược rev_topo và in ra.

```

f(i,1,n) {
    if(check[i]==0) {
        if(!topo(i,a)) return 0;
    }
}
reverse(rev_topo.begin(),rev_topo.end());
for(auto x:rev_topo){
    cout<<x<<" ";
}

```

Kết quả trả về:

1 3 2 5 7 4 6 8

Kiểm tra từng cạnh

1 2: 1 đứng trước 2 (1 ở vị trí 0, 2 ở vị trí 2) -> Đúng.

1 3: 1 đứng trước 3 (1 ở vị trí 0, 3 ở vị trí 1) -> Đúng.

1 4: 1 đứng trước 4 (1 ở vị trí 0, 4 ở vị trí 5) -> Đúng.

2 4: 2 đứng trước 4 (2 ở vị trí 2, 4 ở vị trí 5) -> Đúng.

2 5: 2 đứng trước 5 (2 ở vị trí 2, 5 ở vị trí 3) -> Đúng.

2 6: 2 đứng trước 6 (2 ở vị trí 2, 6 ở vị trí 6) -> Đúng.

3 5: 3 đứng trước 5 (3 ở vị trí 1, 5 ở vị trí 3) -> Đúng.

3 7: 3 đứng trước 7 (3 ở vị trí 1, 7 ở vị trí 4) -> Đúng.

4 6: 4 đứng trước 6 (4 ở vị trí 5, 6 ở vị trí 6) -> Đúng.

4 8: 4 đứng trước 8 (4 ở vị trí 5, 8 ở vị trí 7) -> Đúng.

5 7: 5 đứng trước 7 (5 ở vị trí 3, 7 ở vị trí 4) -> Đúng.

5 8: 5 đứng trước 8 (5 ở vị trí 3, 8 ở vị trí 7) -> Đúng.

6 8: 6 đứng trước 8 (6 ở vị trí 6, 8 ở vị trí 7) -> Đúng.

7 8: 7 đứng trước 8 (7 ở vị trí 4, 8 ở vị trí 7) -> Đúng.

Độ phức tạp:

$$O(V + E)$$

V là số đỉnh, E là số cạnh.

Mỗi đỉnh được thăm đúng 1 lần: $O(V)$.

Mỗi cạnh được duyệt đúng 1 lần trong quá trình DFS: $O(E)$.

Tổng: $O(V + E)$.

Ứng dụng

Giả sử mỗi đỉnh là một mã môn học, với cạnh (u, v) thì u là môn tiên quyết của v, ta có thể tạo ra một thứ tự để đăng kí các môn học.

Bài toán

Problem - 510C - Codeforces

Tóm tắt, phân tích bài toán:

Cho một danh sách các chuỗi đã được sắp xếp theo thứ tự từ điển “đặc biệt”.

Hãy tìm nó, nếu không tìm được in ra “Impossible”.

Trường hợp đặc biệt, nếu chuỗi a là tiền tố của chuỗi b, thì chuỗi a luôn được sắp xếp đứng trước, nếu không thì in ra “Impossible”.

Do có tính chất bắc cầu nên chỉ cần so 2 chuỗi kề nhau.

Ý tưởng:

Giả sử với đề bài:

2

petr

egor

ta sẽ so sánh 2 chuỗi đầu với nhau, tìm ra kí tự đầu tiên khác nhau, ở đây là p với e, vì p đứng trước, nên nó tương đương với $p \rightarrow e$. Ta coi đây là 1 cung từ p đến e, và tương tự như vậy để được nhiều cung. Và sau đó đáp án chính là thứ tự topo.

```

ll n;
cin>>n;
vector<ll>a[100];
string s[n+1];
f(i,1,n) cin>>s[i];
f(i,0,25) check[i]=0;
f(i,1,n-1){
    // so sánh s[i] vs s[i+1]
    ll j=0;
    while(1) {
        if(s[i][j]!=s[i+1][j]) break;
        j++;
        if(j>=s[i+1].length()||j>=s[i].length()) break;
    }
    if(j==s[i+1].length()&&j<s[i].length()){ // length s[i+1]<s[i]

        cout<<"Impossible";
        return 0;
    }
    if(j<s[i].length()) a[s[i][j]-'a'].push_back(s[i+1][j]-'a');
}

```

```

f(i,0,25){
    if(check[i]==0)
    {
        if(!topo(i,a)){
            cout<<"Impossible";
            return 0;
        }
    }
}
reverse(rev_topo.begin(),rev_topo.end());
for(auto x:rev_topo){
    cout<<(char)(x+'a');
}

```


2. DSU và ứng dụng

Khái niệm

“Disjoint Set Union hay Union-Find là một cấu trúc dữ liệu hỗ trợ hai thao tác hiệu quả: hợp nhất hai tập hợp và tìm đại diện của một phần tử. DSU thường được sử dụng để quản lý các tập hợp không giao nhau và hỗ trợ kiểm tra kết nối trong đồ thị.

Union: Hợp nhất hai tập hợp chứa hai phần tử cho trước.

Find: Tìm đại diện (root) của tập hợp chứa phần tử cho trước”

Ứng dụng: tìm thành phần liên thông của đồ thị vô hướng hoặc thành phần liên thông yếu của đồ thị có hướng, tìm tổ tiên.

Cài đặt cấu trúc dữ liệu:

Cài đặt đơn giản:

n: số lượng đỉnh

parent: đại diện/ tổ tiên của đỉnh đó, ban đầu là chính nó.

Find: tìm tổ tiên thật sự của nó

Union: Gộp 2 tập hợp làm 1, hay nói cách khác là khiến 2 tất cả các đỉnh có tổ tiên là v thành tổ tiên là u. Nếu đã cùng tổ tiên (cùng thành phần liên thông trước khi union, thì return false)

```
struct DSU{
    ll n;
    ll parent[10000];
    DSU(ll nn){
        this->n=nn;
        f(i,1,n) parent[i]=i;
    }
    ll find(ll u){
        if(u!=parent[u]) return find(parent[u]);
        else return u;
    }
    bool Union(ll u,ll v){
        u=find(u);
        v=find(v);
        if(u==v) return false;
        parent[v]=u;
        return true;
    }
};
```

Độ phức tạp:

Mỗi thao tác find có thể có độ phức tạp $O(n)$

Mỗi thao tác Union cũng có thể có độ phức tạp $O(n)$

Ứng dụng

Bài toán 1

Tìm số thành phần liên thông

Ý tưởng: Ta sẽ coi mỗi đỉnh trong 1 thành phần liên thông đều chung 1 parent.

Như vậy, chỉ cần kiểm tra xem có bao nhiêu đỉnh $u = \text{parent}[u]$ là xong.

Hoặc, sau mỗi một thao tác gộp thành công, 2 thành phần liên thông sẽ gộp thành 1, như vậy giảm số thành phần liên thông

```
int main(){
    freopen("in.inp", "r", stdin);
    ll n, m;
    cin >> n >> m;
    DSU dsu(n);
    ll cnt = n;
    f(i, 1, m){
        ll a, b;
        cin >> a >> b;
        if(dsu.Union(a, b)) cnt--;
    }
    cout << cnt;
}
```

Đây là kết quả:

TLE 33 / 45 C++20

Vì có độ phức tạp lớn $O(m * n)$

Vì vậy, chúng ta cần một cách tối ưu hơn:

Tối ưu nén đường đi:

```
ll find(ll u){
    if(u!=parent[u]) return parent[u]=find(parent[u]);
    else return u;
}
```

Chỉ cần sửa hàm find như vậy, độ phức tạp tối đa cũng chỉ là $O(\log n)$, tuy nhiên nếu không xuất hiện việc union, thì lần find tiếp theo chỉ mất $O(1)$

Tối ưu theo kích cỡ:

```
ll parent[100001], sz[100001];
DSU(ll nn){
    this->n=nn;
    f(i,1,n) {
        parent[i]=i;
        sz[i]=1;
    }
}
```

Ta gọi một mảng sz, trong đó sz[i] lưu trữ số lượng đỉnh được đại diện bởi i, hay nói cách khác là số đỉnh x có parent[x]=i.

Sau đó tìm đỉnh có sz lớn hơn làm tổ tiên chung.

```

bool Union(ll u,ll v){
    u=find(u);
    v=find(v);
    if(u==v) return false;
    if(sz[u]<sz[v]) swap(u,v);
    sz[u]+=sz[v];
    parent[v]=u;
    sz[v]=0;
    return true;
}

```

Cách trên sẽ giúp giảm số đỉnh phải đổi parent hơn, xác suất find chỉ mất $O(1)$ cao hơn $O(n)$

Sử dụng kết hợp cả 2 và nộp:

AC 45 / 45

Ngoài ra, DSU còn hỗ trợ rất tốt cho một thuật toán đã được học, Kruskal

Bài toán 2

Cây khung nhỏ nhất - LQDOJ: Le Quy Don Online Judge

Ý tưởng: Duyệt từng cạnh, Kiểm tra xem nếu thêm cạnh đấy có tạo chu trình không (nếu union trả về true thì không) , nếu không thì lấy.

```

#define fi first
#define se second
#define pii pair<ll,pair<ll,ll>>

```

Ta lưu trữ thông tin các cạnh dưới dạng `pair<value,pair<start,end>>`, để tiện khi gọi sort.

```

ll n,m;
cin>>n>>m;
vector<pii>edge(m+1);
f(i,1,m){
    ll a,b,v;
    cin>>a>>b>>v;
    edge[i].fi=v;
    edge[i].se.fi=a;
    edge[i].se.se=b;
}
sort(edge.begin()+1,edge.end());
ll ans=0;
DSU dsu(n);
f(i,1,m){
    if(dsu.Union(edge[i].se.fi,edge[i].se.se)){
        ans+=edge[i].fi;
    }
}
cout<<ans;

```

Độ phức tạp:

Sắp xếp các cạnh: $O(m \log(m))$

Mỗi lần DSU trung bình mất $\log n$, ta duyệt m cạnh , nên sẽ là $O(m \log n)$

Tổng $O(m*(\log m + \log n))$

3. Thuật toán Boruvka

Khái niệm

“

- **Boruvka** là một trong những thuật toán đầu tiên được phát triển để tìm cây khung nhỏ nhất.
- Thuật toán liên tục chọn cạnh nhỏ nhất từ mỗi thành phần con và hợp nhất các thành phần này lại.

Cách hoạt động

1. Bắt đầu với mỗi đỉnh là một thành phần con riêng lẻ.
2. Trong mỗi bước, chọn cạnh nhỏ nhất nối mỗi thành phần con với một thành phần khác.
3. Hợp nhất các thành phần con lại bằng các cạnh vừa chọn.
4. Lặp lại cho đến khi tất cả các thành phần được hợp nhất thành một cây. ”

Về cơ bản, thuật toán này cũng sử dụng DSU.

Mảng `cheap[i]` lưu cạnh có độ dài nhỏ nhất của cây i với một cây khác.

Chúng ta duyệt từng cạnh, với điều kiện tính tới thời điểm hiện tại, 2 đỉnh của cạnh đó chưa cùng một cây/set/ thành phần liên thông. Có nghĩa là đây là cạnh nối 2 cây lại với nhau nếu như được union. Và ta sẽ tìm được `cheap`.

Sau đó lại duyệt từng thành phần liên thông để thêm các cạnh `cheap` đó vào cây.

```

ll cnt=n,ans=0;
while(cnt>1){
    vector<ll>cheap(n+1,-1);
    f(i,1,m){
        ll a=dsu.find(edge[i].se.se);
        ll b=dsu.find(edge[i].se.fi);
        ll val=edge[i].fi;
        if(a==b) continue;
        if(cheap[a]==-1 || edge[cheap[a]].fi>val) cheap[a]=i;
        if(cheap[b]==-1 || edge[cheap[b]].fi>val) cheap[b]=i;
    }
    f(i,1,n){
        if(cheap[i]==-1) continue;
        pii e=edge[cheap[i]];
        if(!dsu.Union(e.se.se,e.se.fi)) continue;
        ans+=e.fi;
        cnt--;
    }
}
cout<<ans;

```

Độ phức tạp:

Thao tác tìm cheap: $O(m \log n)$

Thao tác tìm cạnh trong cây khung: $O(n \log n)$

Tuy nhiên với 2 thao tác trên, $\log n$ là trường hợp xấu nhất, còn thực tế đa số đều chỉ ở mức $O(1)$

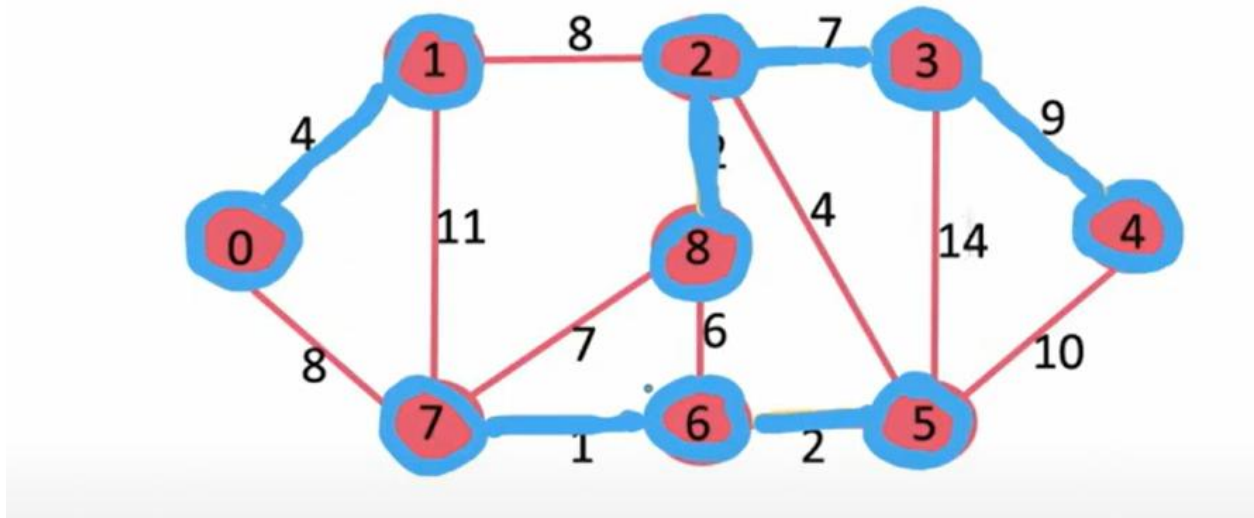
Số vòng while: tối đa $\log n$ do sau mỗi vòng số cây giảm đi ít nhất 1 nửa

Tổng $O((\log n)^2 * (m + n))$.

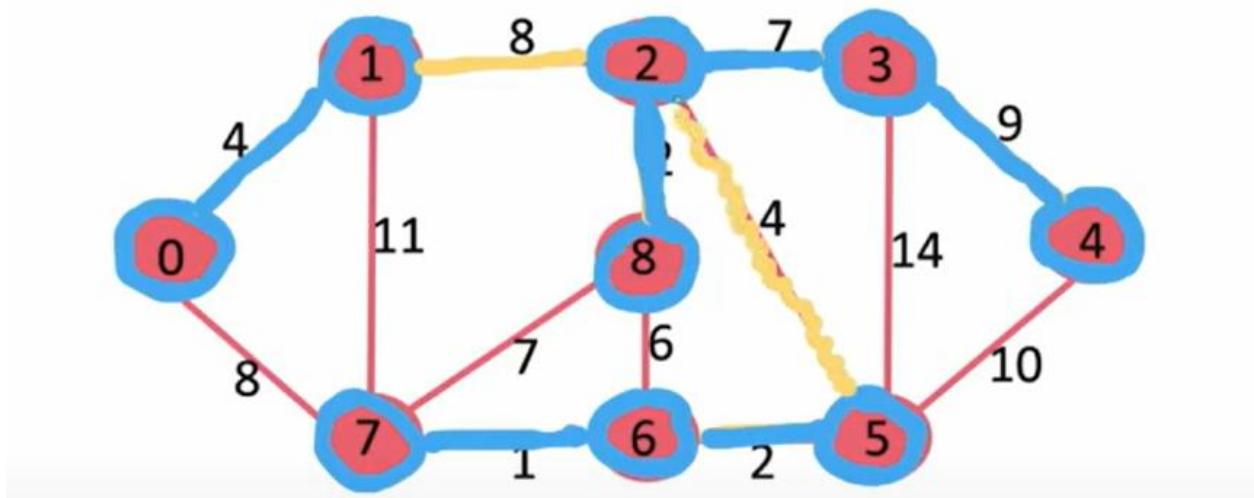
Đây là cách tính khá là tiêu cực do thực tế đã có nhiều tối ưu, người ta chứng minh được độ phức tạp thực tế chỉ ở mức $O(\log(n) * (m + n))$.

Ví dụ minh họa:

Sau vòng while đầu tiên:



Sau vòng while thứ 2: (vậy là đã có đáp án)



Chạy thử nghiệm:


```
C:\Users\LEGION
Edge List:
0 1 4
0 7 8
1 2 8
1 7 11
2 3 7
2 5 4
2 8 2
3 4 9
3 5 14
4 5 10
5 6 2
6 7 1
6 8 6
7 8 7
MST: 37
```

Kết quả như kỳ vọng.

So sánh ưu nhược điểm so với các thuật toán khác (Kruskal, Prim)

So với kruskal:

Ưu điểm: Trong trường hợp đồ thị dày (n nhiều cạnh) thì boruvka có độ phức tạp tốt hơn

Nhược điểm: cài đặt khó hơn, trong trường hợp đồ thị ít cạnh thì độ phức tạp kém hơn.

So với prim:

Cài đặt dễ hơn

Độ phức tạp: nhìn chung tệ hơn

4. Thuật toán Kosaraju

Khái niệm

Là thuật toán tìm số thành phần liên thông mạnh (SCC) trong đồ thị có hướng.

Các bước :

Bước 1 : tạo **đồ thị nghịch đảo b** của **đồ thị a** đã được nhập => a và b có cùng số SCC

Bước 2 :

Duyệt dfs đồ thị a, sau đây những đỉnh không còn đường đi sẽ ném vào **stack s**.

Bước 3 : khai báo **cnt=0**, lấy từng đỉnh trong s ra, nếu chưa được duyệt thì :

+ dfs b

+ cnt++

Số SCC chính là cnt, và mỗi SCC sẽ gồm các đỉnh đã được duyệt trong lần dfs đó.

Code:

```
ll check[10001];
stack<ll> s;
void dfsa(ll i, vector<ll>a[]){
    check[i]=1;
    for(auto j:a[i]){
        if(check[j]==0){
            dfsa(j,a);
        }
    }
    s.push(i);
}
void dfsb(ll i, vector<ll>a[]){
    check[i]=1;
    for(auto j:a[i]){
        if(check[j]==0){
            dfsb(j,a);
        }
    }
}
cout<<i<<" ";
}
```

```

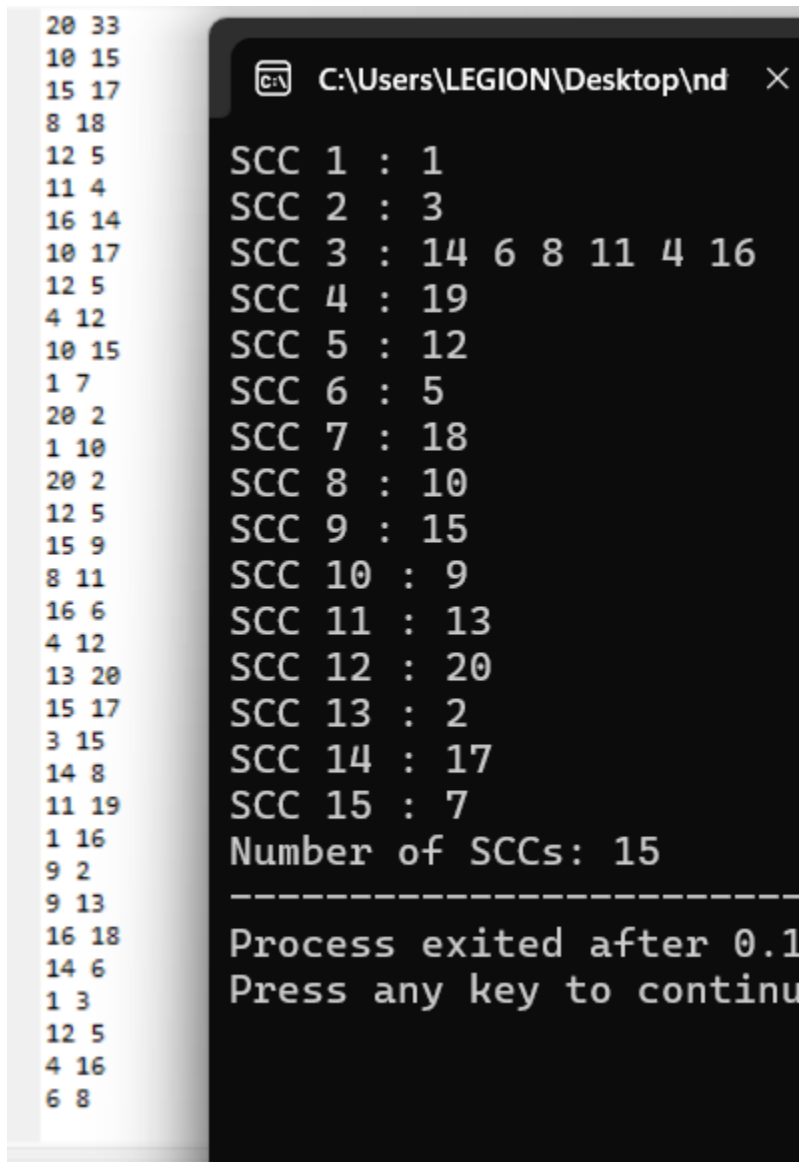
cin>>n>>m;
vector<ll> a[n+1];
vector<ll> b[n+1];
f(i,1,m){
    ll s,e;
    cin>>s>>e;
    a[s].push_back(e);
    b[e].push_back(s);
}
f(i,1,n) check[i]=0;
f(i,1,n) {
    if(check[i]==0) dfsa(i,a);
}
ll cnt=0;
f(i,1,n) check[i]=0;
while(!s.empty()){
    ll i=s.top();
    s.pop();
    if(check[i]==0){
        cout<<"SCC " <<cnt<<" : ";
        dfsb(i,b);
        cnt++;
        cout<<endl;
    }
}
}

```

Độ phức tạp

$O(2 \cdot \text{dfs}) = O(2 \cdot (n+m))$

Demo :



```
20 33
10 15
15 17
8 18
12 5
11 4
16 14
10 17
12 5
4 12
10 15
1 7
20 2
1 10
20 2
12 5
15 9
8 11
16 6
4 12
13 20
15 17
3 15
14 8
11 19
1 16
9 2
9 13
16 18
14 6
1 3
12 5
4 16
6 8

SCC 1 : 1
SCC 2 : 3
SCC 3 : 14 6 8 11 4 16
SCC 4 : 19
SCC 5 : 12
SCC 6 : 5
SCC 7 : 18
SCC 8 : 10
SCC 9 : 15
SCC 10 : 9
SCC 11 : 13
SCC 12 : 20
SCC 13 : 2
SCC 14 : 17
SCC 15 : 7
Number of SCCs: 15
-----
Process exited after 0.1
Press any key to continu
```

Tài liệu tham khảo

[Kosaraju's Algorithm in C | GeeksforGeeks](#)

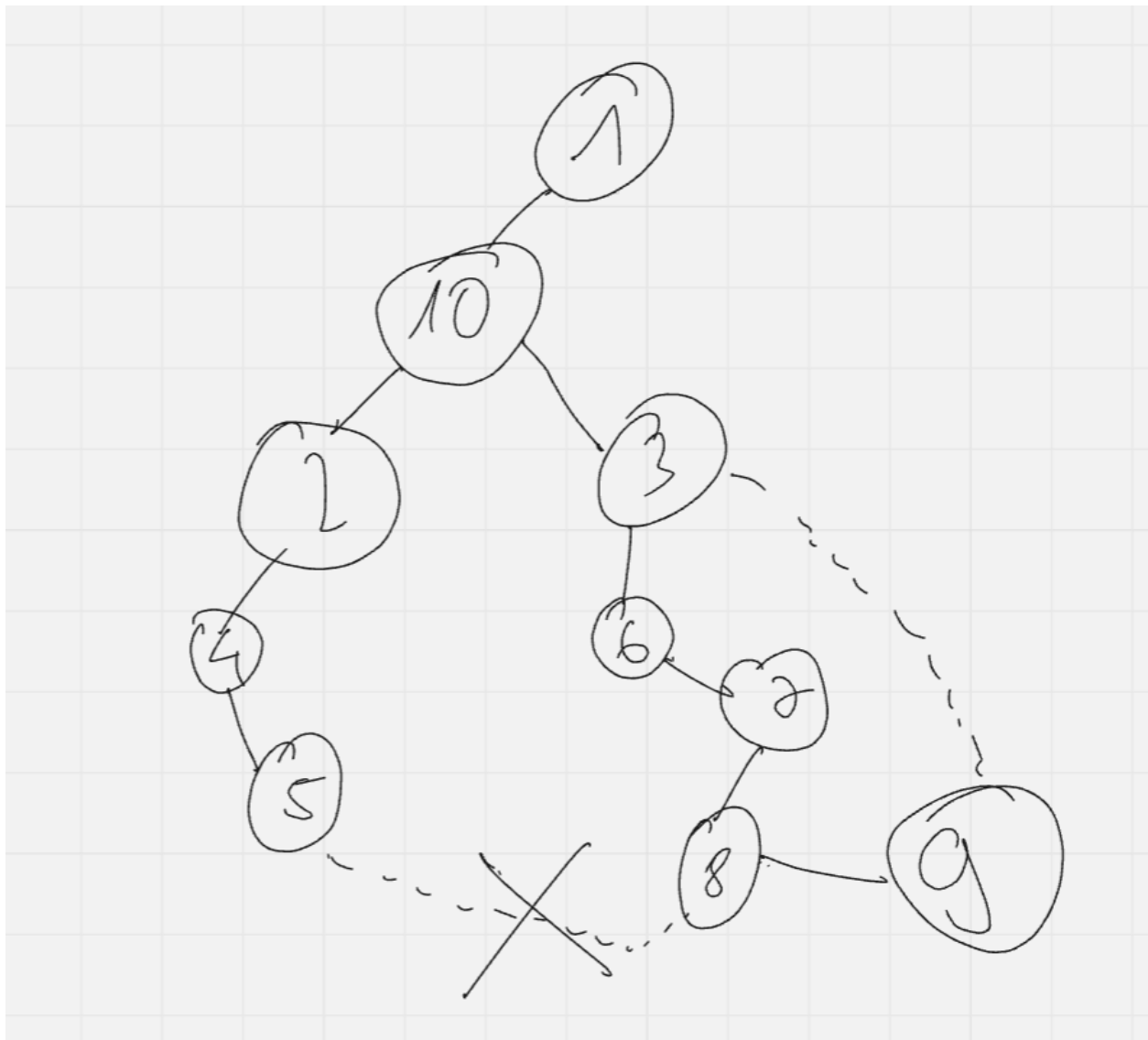
5. Thuật toán Tarjan

Khái niệm

Là thuật toán dùng để xác định cầu/ khớp cho đồ thị vô hướng, số SCC cho đồ thị có hướng.

Một số mảng/ khái niệm sẽ sử dụng :

Cây dfs : là cây được tạo bởi các cạnh trong quá trình dfs, các cạnh đó trong chủ đề này sẽ gọi là cạnh nét liền, các cạnh còn lại là nét đứt, ví dụ



Num: dùng để lưu thứ tự duyệt dfs của 1 đỉnh

Low: cho biết giá trị của đỉnh có num nhỏ nhất mà đỉnh đang xét (đỉnh i) có thể đi tới (với đồ thị vô hướng thì không tính cha trực tiếp của nó, và quãng đường không đi ngược lại các cạnh trên cây dfs, ví dụ đi từ 9 tới 8 qua cạnh (8,9) là đi ngược), khởi tạo bằng $low = num$

Cách xác định :

Với cạnh nét liền bắt đầu từ i, kết thúc tại x :

```
low[i] = min(low[x], low[i]);
```

Với cạnh nét đứt :

```
low[i] = min(low[i], num[x]);
```

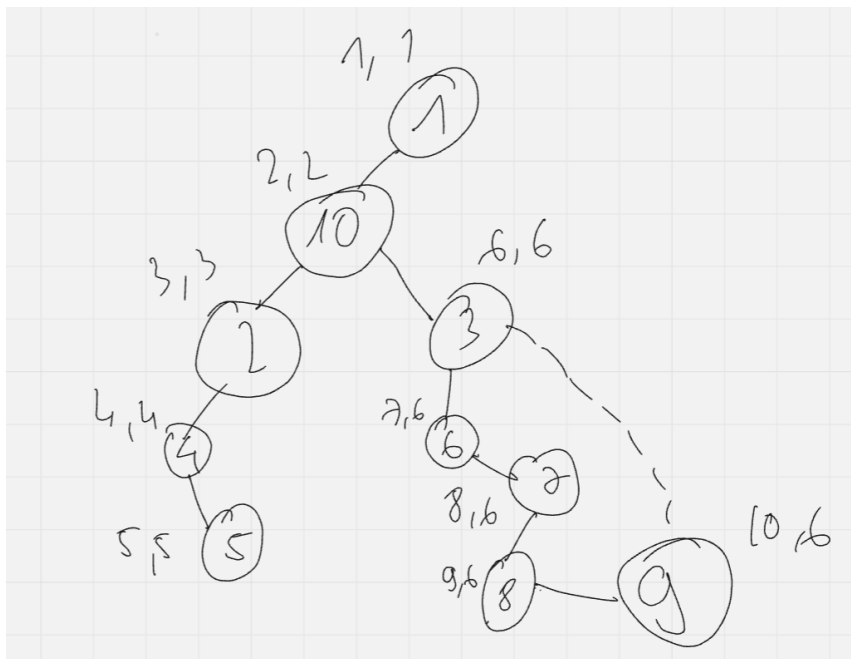
Suy ra : để $low[i] < num[i]$ thì từ i phải có đường đi tới tổ tiên của i, và với đồ thị vô hướng thì không tính cha của i và các cạnh đi ngược

$low[i] = num[x]$ khi i có đường đi tới x thoả mãn điều kiện như trên

Check: kiểm tra xem đã được duyệt dfs chưa

Ví dụ:

Hiện thị num, low từng đỉnh, trong đó (10,6) là 1 cạnh nhưng không thuộc cây dfs

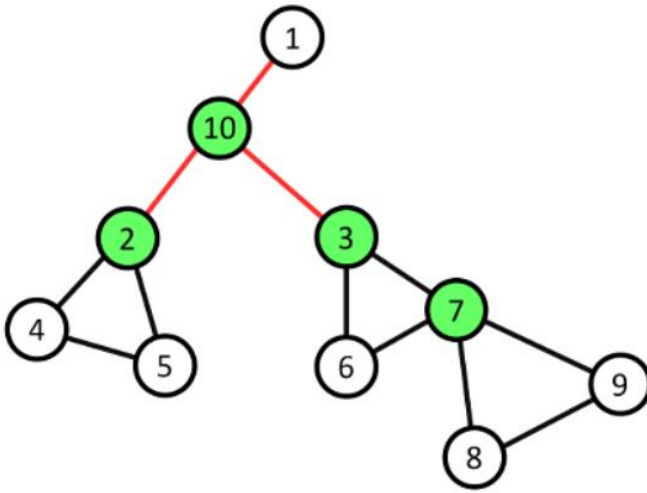


Bài toán tính số cạnh cầu (bridge), đỉnh trụ (khớp : articulation point) trên đồ thị vô hướng

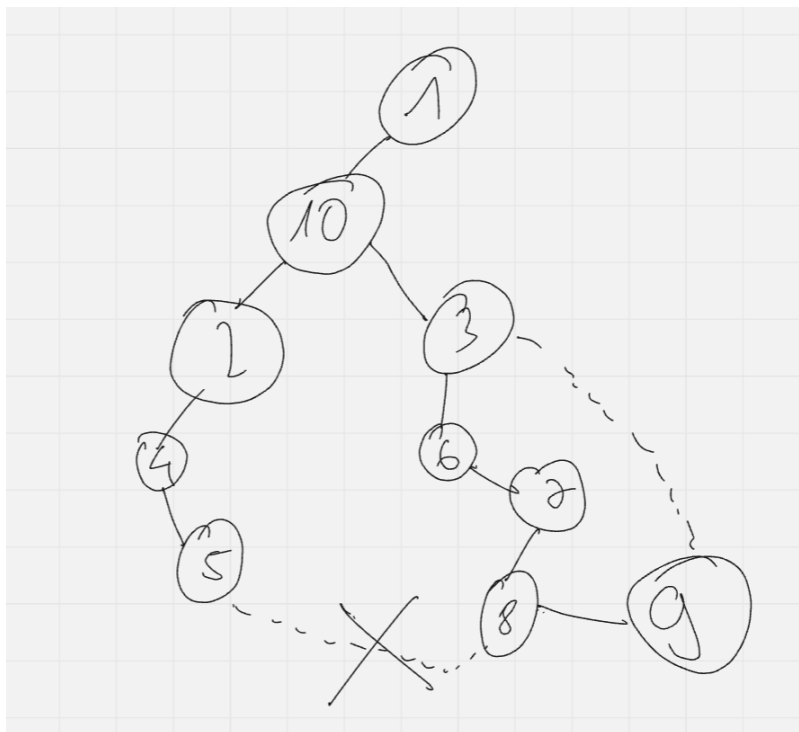
Khái niệm

Là cầu hoặc khớp \Leftrightarrow khi xoá nó sẽ tăng số thành phần liên thông.

- Các cạnh màu đỏ là cạnh cầu.
- Các đỉnh màu xanh lá là đỉnh khớp.



Duyệt dfs :



Các cạnh nét liền là các cạnh được sử dụng trong quá trình dfs.

Giả sử đây chưa phải tất cả các cạnh của đồ thị

Nhận xét : với đồ thị vô hướng, nếu như cố định cây dfs. Thì chỉ còn lại những cạnh nối từ đỉnh u đến đỉnh v là cha trực tiếp của nó (ví dụ như 9 và 3), và không tồn tại cạnh nối 2 đỉnh từ 2 gốc cây con khác nhau (ví dụ 5 và 8), vì nếu tồn tại cạnh (5,8) , thứ tự dfs, cây dfs sẽ thay đổi.

Để 1 cạnh (u,v) là **cạnh cầu** thì $low[v]=num[v]$ vì $low[v]$ luôn $\leq num[v]$ và nếu $low[v] < num[v] \Leftrightarrow$ tồn tại 1 cạnh nối từ tổ tiên của v đến v hoặc ít nhất 1 hậu duệ của v . Ví dụ với cạnh (8,9), nếu tồn tại cạnh nối từ 9 đến 1 trong (1,10,3,6,7) thì cạnh đó sẽ không là cạnh cầu.

Để u là **đỉnh trụ** thì cần ít nhất 1 cạnh (u,v) sao cho $low[v] \geq num[u]$, vì nếu $low[v] < num[u]$ thì v có đường đi tới tổ tiên của u mà không đi qua $u \Leftrightarrow u$ không là khớp

Ngoài ra với đỉnh gốc, thì chỉ cần có 2 con trực tiếp trong cây là đỉnh trụ

Code :

```
ll n,e,timer=0,bridge=0,artp=0;
ll check[10001], num[10001], low[10001], parent[10001],art[10001];
void dfs(ll i,vector<ll>a[]){
    check[i]=1; timer++;
    ll cnt=0;
    low[i]=num[i]=timer;
    for(auto x:a[i]){
        if(check[x]==0){
            parent[x]=i;
            dfs(x,a);
            if(low[x]>=num[i]) art[i]=1;
            low[i]=min(low[x],low[i]);
            cnt++;
            if(num[x]==low[x]) bridge++;
        }
        else if(x!=parent[i]) {
            low[i]=min(low[i],num[x]);
        }
    }
    if(parent[i]==-1){
        if(cnt>1) {
            art[i]=1; artp++;
        }
    }
    else{
        if(art[i]==1) artp++;
    }
}
```

```

int main(){
    freopen("in.inp","r",stdin);
    //freopen("out.oup","w",stdout);
    cin>>n>>e;
    vector<ll>a[n+1];
    f(i,1,e){
        ll c,d;
        cin>>c>>d;
        a[c].push_back(d);
        a[d].push_back(c);
    }
    f(i,1,n){
        check[i]=0;
        parent[i]=-1;
        art[i]=0;
    }
    f(i,1,n){
        if(check[i]==0) dfs(i,a);
    }
    //f(i,1,n) cout<<i<<" "<<low[i]<<" "<<num[i]<<endl;
    cout<<artp<<" "<<bridge;
}

```

Thử nghiệm :

Kết quả chính xác.

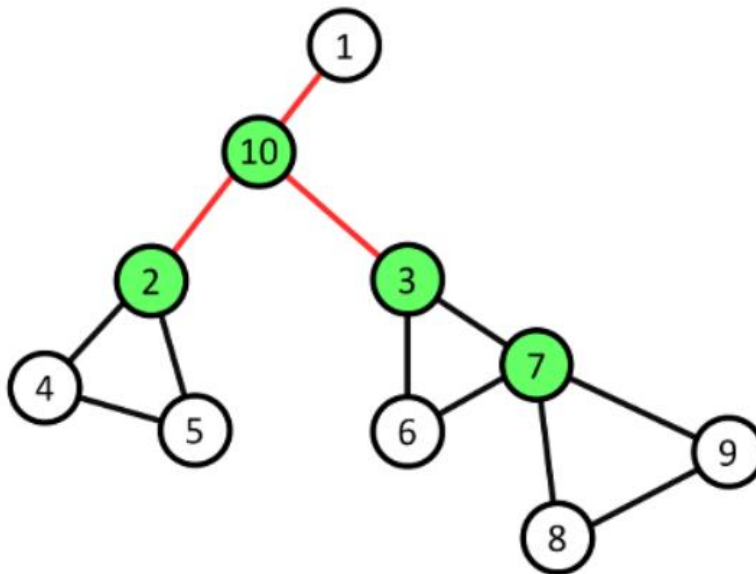
Nộp trên online judge :

Execution Results

✓ x22

> Test case #1: **Accepted** [0.005s,2.02 MB] (0/0)
 > Test case #2: **Accepted** [0.005s,3.53 MB] (1/1)
 > Test case #3: **Accepted** [0.005s,2.00 MB] (1/1)
 Test case #4: **Accepted** [0.004s,1.99 MB] (1/1)
 Test case #5: **Accepted** [0.005s,1.93 MB] (1/1)
 Test case #6: **Accepted** [0.006s,3.33 MB] (1/1)
 Test case #7: **Accepted** [0.007s,3.41 MB] (1/1)
 Test case #8: **Accepted** [0.011s,3.86 MB] (1/1)
 Test case #9: **Accepted** [0.039s,4.82 MB] (1/1)
 Test case #10: **Accepted** [0.040s,5.72 MB] (1/1)
 Test case #11: **Accepted** [0.039s,5.69 MB] (1/1)
 Test case #12: **Accepted** [0.035s,5.62 MB] (1/1)
 Test case #13: **Accepted** [0.037s,5.74 MB] (1/1)
 Test case #14: **Accepted** [0.005s,1.94 MB] (1/1)
 Test case #15: **Accepted** [0.005s,1.94 MB] (1/1)
 Test case #16: **Accepted** [0.006s,3.26 MB] (1/1)
 Test case #17: **Accepted** [0.005s,2.02 MB] (1/1)
 Test case #18: **Accepted** [0.007s,3.33 MB] (1/1)
 Test case #19: **Accepted** [0.011s,3.60 MB] (1/1)
 Test case #20: **Accepted** [0.006s,2.04 MB] (1/1)
 Test case #21: **Accepted** [0.005s,2.02 MB] (1/1)
 Test case #22: **Accepted** [0.008s,3.32 MB] (1/1)

- Các cạnh màu đỏ là cạnh cầu.
- Các đỉnh màu xanh lá là đỉnh khớp.



s\LEGION\Desktop\ndt\in.inp - [Executing] - Dev-C++ 5.11
Search View Project Execute Tools AStyle Window Help
ls)
Classes Debug ly thuyết tarjan.cpp in.inp

```

1 10 12
2 1 10
3 10 2
4 10 3
5 2 4
6 4 5
7 5 2
8 3 6
9 6 7
10 7 3
11 7 8
12 8 9
13 9 7
14

```

C:\Users\LEGION\Desktop\nd
4 3

Process exited after 0.3954 seconds with return value 0
Press any key to continue . . . |

Độ phức tạp

$O(n+e)$

Tính số thành phần liên thông mạnh (SCC)

Tóm tắt ý tưởng :

1 scc có 1 đại diện là 1 đỉnh i duy nhất, đại diện đó có $low[i]=num[i]$,

Ta sẽ thêm tất cả các đỉnh được duyệt vào 1 stack, và đến khi tìm được đỉnh i có $low[i]=num[i]$ thì pop các đỉnh trong stack đến hết đỉnh i , đó là các đỉnh cùng thành phần liên thông với i . Vì sao? Vì chúng tồn tại đường đi tới i , mà từ i lại có thể đi qua tất cả các đỉnh còn lại.

Tuy nhiên, phải sửa lại cách tính low 1 chút, cụ thể :

```
else if(instack[x]==1) low[i]=min(low[i],num[x]);
```

Thật ra nếu

```
else if(instack[x]==1) low[i]=min(low[i],low[x]);
```

Với cách trên : $low[i]$ đại diện cho đỉnh có num bé nhất mà i đi tới

Với cách dưới : $low[i]$ chính là đỉnh đại diện scc của i (giả sử đỉnh đại diện đó là x , nếu x không còn cây con nào sau khi duyệt cây con trực tiếp chứa i thì đúng, còn không thì có thể sai)

thì cũng không sai, vì đơn giản việc chúng ta làm chỉ là loại bỏ việc đỉnh i trở thành đại diện 1 scc nếu $low/num[x]<num[i]$, và việc nó xác định đại diện của nó dựa vào việc loại đỉnh đó ra khỏi stack.

Mảng instack dùng để biết nó đang trong stack hay không.

Nếu không trong, nó đã thuộc 1 scc khác đã xác định, không cần quan tâm

Code :

```
Untitled1.cpp  in.inp
7  #define pii pair<ll,pair<ll,ll>>
8  #define pii pair<ll,ll>
9  ll num[10001],low[10001],check[10001],instack[10001];
10 ll cnt=0,ans=0;
11 stack<ll>s;
12 void dfs(ll i,vector<ll>a[]){
13     cnt++;
14     num[i]=cnt;
15     low[i]=cnt;
16     check[i]=1;
17     if(instack[i]==0) {
18         s.push(i);
19         instack[i]=1;
20     }
21     for(auto x:a[i]){
22         if(check[x]==0){
23             dfs(x,a);
24             low[i]=min(low[x],low[i]);
25         }
26         else if(instack[x]==1) low[i]=min(low[i],num[x]);
27     }
```

```
    if(low[i]==num[i]) {
        ans++;
        while(1){
            if(s.empty()) break;
            if(s.top()==i){
                instack[s.top()]=0;
                s.pop();
                break;
            }
            if(s.empty()) break;
            instack[s.top()]=0;
            s.pop();
        }
    }
}
```

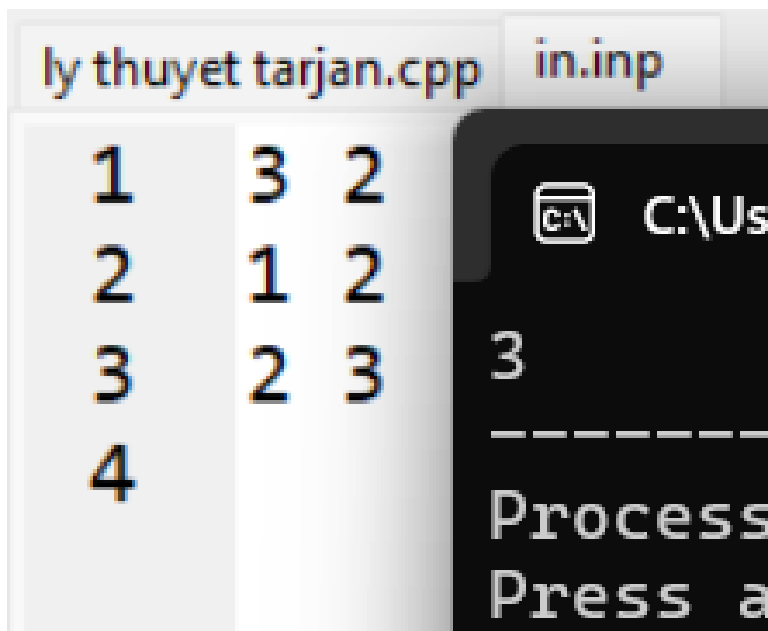
Chạy trên online judge :

Execution Results

✓ ×15

- Test case #1: **Accepted** [0.004s,2.02 MB] (1/1)
- Test case #2: **Accepted** [0.003s,1.88 MB] (1/1)
- Test case #3: **Accepted** [0.063s,4.88 MB] (1/1)
- Test case #4: **Accepted** [0.004s,1.94 MB] (1/1)
- Test case #5: **Accepted** [0.004s,1.95 MB] (1/1)
- Test case #6: **Accepted** [0.004s,2.03 MB] (1/1)
- Test case #7: **Accepted** [0.004s,2.03 MB] (1/1)
- Test case #8: **Accepted** [0.004s,1.91 MB] (1/1)
- Test case #9: **Accepted** [0.012s,3.71 MB] (1/1)
- Test case #10: **Accepted** [0.005s,3.22 MB] (1/1)
- Test case #11: **Accepted** [0.055s,4.61 MB] (1/1)
- Test case #12: **Accepted** [0.054s,5.20 MB] (1/1)
- Test case #13: **Accepted** [0.004s,2.02 MB] (1/1)
- Test case #14: **Accepted** [0.005s,3.60 MB] (1/1)
- Test case #15: **Accepted** [0.004s,2.02 MB] (1/1)

Với test case : chính xác



Độ phức tạp

$O(n+e)$

So sánh với kosaraju

Code phức tạp hơn

Độ phức tạp tốt hơn do chỉ cần 1 lần dfs, còn kosaraju cần 2

Tài liệu tham khảo

Tài liệu giáo khoa chuyên tin quyển 1 – Hồ Sĩ Đàm – NXB Giáo Dục Việt Nam

6. Luồng

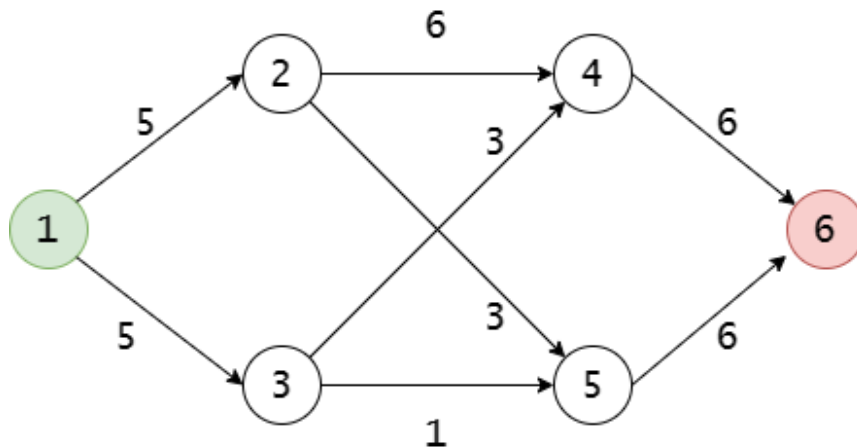
"

Các định nghĩa

Có rất nhiều hình ảnh thực tế để miêu tả một mạng và luồng trên mạng đó, như một mạng điện, một mạng kết nối dữ liệu giữa các máy, hay phổ biến hơn là một hệ thống ống nước.

Một đồ thị được gọi là **mạng** (network) nếu nó là đồ thị **có hướng**, trong đó:

- Tồn tại một đỉnh không có cạnh đi vào, gọi là **đỉnh phát/nguồn** (source)
- Tồn tại một đỉnh không có cạnh đi ra, gọi là **đỉnh thu/đích** (sink)
- Mỗi cạnh được gán một trọng số, gọi là **khả năng thông qua/dung lượng** (capacity) của cạnh.

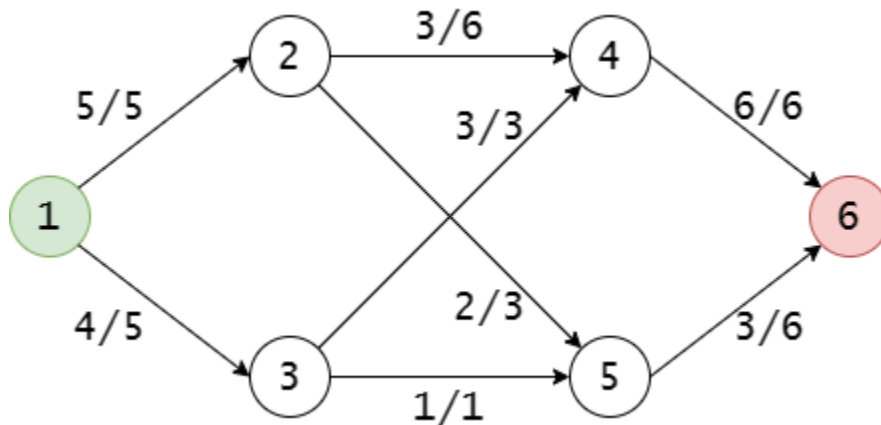


Một mạng hợp lệ. Đỉnh phát và đỉnh thu được đánh dấu bằng hai màu khác.

Một **luồng** (flow) trên mạng là một phép gán cho mỗi cạnh một số thực thoả mãn:

- Luồng trên mỗi cạnh có giá trị không vượt quá khả năng thông qua của cạnh đó:
- Với mọi đỉnh không trùng với đỉnh phát và đỉnh thu, tổng luồng trên các cạnh đi vào bằng tổng luồng trên các cạnh đi ra. Tính chất này tương đối giống với định luật I Kirchhoff của dòng điện.

- Giá trị được gọi là **luồng trên cạnh**
- **Giá trị của luồng** là tổng luồng trên các cạnh đi ra khỏi đỉnh phát, cũng chính là tổng luồng trên các cạnh đi vào đỉnh thu.



Một luồng hợp lệ. Giá trị f/c trên cạnh biểu diễn luồng/khả năng thông qua.

"

Luồng cực đại là luồng có giá trị lớn nhất.

Ứng dụng luồng cực đại

I. Ứng dụng thực tế

1. Hệ thống cấp thoát nước / mạng điện

- Mô hình: mỗi ống/cáp là một cạnh với dung lượng.
- Tìm **lượng dòng lớn nhất** từ trạm nguồn đến trạm tiêu thụ.
- Giúp thiết kế mạng lưới tối ưu.

2. Giao thông / mạng vận tải

- Đường là các cạnh với giới hạn số xe/giờ.
- Tìm luồng cực đại giữa hai địa điểm → **tối ưu hóa hạ tầng**

3. Mạng máy tính

- Tối ưu băng thông từ server đến người dùng.
- Đảm bảo **tải phân phối hợp lý** qua các nút trung gian.

4. Xếp ca làm việc / phân công nhiệm vụ (Job assignment)

- Công nhân và công việc là 2 tập.
- Nếu một người làm được một công việc \rightarrow nối cạnh giữa họ.
- Tìm luồng cực đại từ nguồn đến bồn \rightarrow giải bài toán **ghép cặp cực đại**.

5. Quản lý chuỗi cung ứng (Supply chain)

- Các nhà máy, kho, và cửa hàng là các đỉnh.
- Capacity là năng lực sản xuất/vận chuyển.
- Dùng maximum flow để xác định **khả năng cung ứng tối đa**.

II. Ứng dụng trong lý thuyết và thuật toán

1. Tìm ghép cặp cực đại trong đồ thị hai phía (Maximum Bipartite Matching)

- Biến bài toán ghép cặp thành bài toán luồng.
- Rất phổ biến trong các bài toán phân công, lựa chọn.

2. Tìm đường đi phân tách nhỏ nhất (Min cut path)

- Kết quả phụ của max flow là **minimum cut**.
- Dùng để xác định phần yếu nhất trong mạng lưới (dễ bị chia cắt nhất).

Ví dụ cụ thể

Bài toán: Có n học sinh và m câu lạc bộ. Mỗi học sinh chọn được tối đa 2 CLB, mỗi CLB nhận tối đa 3 học sinh. Có thể phân bố không?

Giải pháp:

- Tạo đồ thị luồng với:
 - Nguồn \rightarrow học sinh (capacity = 1)
 - Học sinh \rightarrow CLB (nếu chọn được)
 - CLB \rightarrow đích (capacity = giới hạn học sinh)
- Chạy **max flow**, kiểm tra xem tổng luồng có bằng số học sinh không.

7. Thuật toán Edmonds-Karp

“Đôi lời về lịch sử thuật toán

Năm 1956, L. R. Ford Jr. và D. R. Fulkerson đề xuất một phương pháp để tìm ra luồng cực đại trên mạng. Tuy nhiên, phương pháp này không chỉ rõ việc tìm *đường tăng luồng* như thế nào. Đến năm 1972, Jack Edmonds and Richard Karp đã hoàn thiện phương pháp trên bằng cách sử dụng thuật BFS để tìm *đường tăng luồng*.

Nhiều tài liệu mà chúng ta đang dùng có sử dụng cụm từ "thuật toán Ford-Fulkerson" để gọi thuật tìm luồng cực đại hoàn chỉnh, và biến "thuật toán Edmonds-Karp" thành một thuật xa lạ kì quặc nào đó. Điều này có lẽ cũng ... không hẳn là sai. Em sẽ sử dụng tên Edmonds-Karp cho thuật toán, và chỉ gọi là "phương pháp Ford-Fulkerson" thôi.”

Về cơ bản, Ford-Fulkerson đã nghĩ ra phương pháp sử dụng cạnh ngược để tính toán, nhưng chỉ nói chung chung là tìm đường đi hợp lệ, nhưng không nêu rõ cách tìm, Edmonds-Karp đã chỉ ra BFS là 1 cách tương đối tốt, chứng minh nó tốt hơn DFS.

Ngoài ra, BFS được ra đời năm 1959, bởi Edward F. Moore, sau phương pháp Ford-Fulkerson 3 năm.

Thuật toán+ code

Định nghĩa:

```
#define ll long long
#define f(i,a,b) for(int i=a;i<=b;i++)
#define fi first
#define se second
#define pii pair<ll,pair<ll,ll>>
#define pii pair<ll,ll>
```

Các mảng, vector, biến:

```
ll cap[1001][1001],parent[1001],r[1001][1001], check[1001];
vector<ll>g[1001];
ll n,e,s,t;
```

Trong đó:

n,e,s,t: số đỉnh, số cạnh, nguồn, đích

Cap: khả năng thông qua của đường dẫn đó

Parent: điểm cha, điểm trước của nó trong đường đi từ s tới t theo phương pháp BFS

r: lưu khả năng thông qua còn lại, sức chứa còn lại của đường dẫn, cạnh đó, bao gồm cả cạnh ngược

Cạnh ngược: nếu tồn tại cạnh (u,v) thì (v,u) chính là cạnh ngược

check: kiểm tra xem đã được duyệt chưa (sử dụng trong BFS)

g: vector lưu danh sách kề của đồ thị.

Hàm main:

```
int main(){
    //freopen("in.inp","r",stdin);
    //freopen("out.oup","w",stdout);
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cin>>n>>e>>s>>t;
    f(i,1,e){
        ll a,b;
        cin>>a>>b;
        cin>>cap[a][b];
        r[a][b]=cap[a][b];
        r[b][a]=0;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    cout<<edmondsKarp();
}
```

Hàm bfs:

Chức năng: tìm đường đi từ s tới t, nếu không có trả về false.

Lưu ý: không đi qua các cạnh có $r \leq 0$.

```
11 bfs(){
    queue<ll> q;
    q.push(s);
    f(i,1,n) {
        parent[i]=i;
        check[i]=0;
    }
    check[s]=1;
    while(!q.empty()){
        ll i=q.front();
        if(i==t) return 1;
        q.pop();
        //cout<<i<<endl;
        for(auto x:g[i]){
            if(check[x]==1) continue;
            if(r[i][x]==0) continue;
            q.push(x);
            parent[x]=i;
            check[x]=1;
        }
    }
    return 0;
}
```

Hàm edmondsKarp:

Chức năng: trả về giá trị luồng cực đại

```

11 edmondsKarp(){
    11 ans=0;
    while(1){
        if(!bfs()) break;
        11 minr=10000000;
        11 x=t;
        while(1){
            //cout<<x<<" ";
            if(x==s) break;
            11 p=parent[x];
            minr =min(minr,r[p][x]);
            x=p;
        }
        //cout<<endl;
        x=t;
        while(1){
            if(x==s) break;
            11 p=parent[x];
            r[p][x]-=minr;
            r[x][p]+=minr;
            x=p;
        }
        ans+=minr;
    }
    return ans;
}

```

Giải thích chi tiết

Đường tăng: là một đường đi từ đỉnh nguồn s đến đỉnh đích t trong đồ thị dư, mà trên tất cả các cạnh của đường đi đó đều còn sức chứa > 0 ($r > 0$).

Bước 1: Dùng BFS để tìm đường tăng từ s đến t. Nếu không tìm được (hết đường tăng), thoát khỏi vòng lặp.

Bước 2: minr: r bé nhất trong quãng đường từ s tới t, chính là giá trị mà ta có thể cộng vào giá trị của luồng, tìm minr theo phương pháp tìm đường đi từ s->t sau khi đã bfs (đã được học, không nhắc lại).

Bước 3: cập nhật lại r.

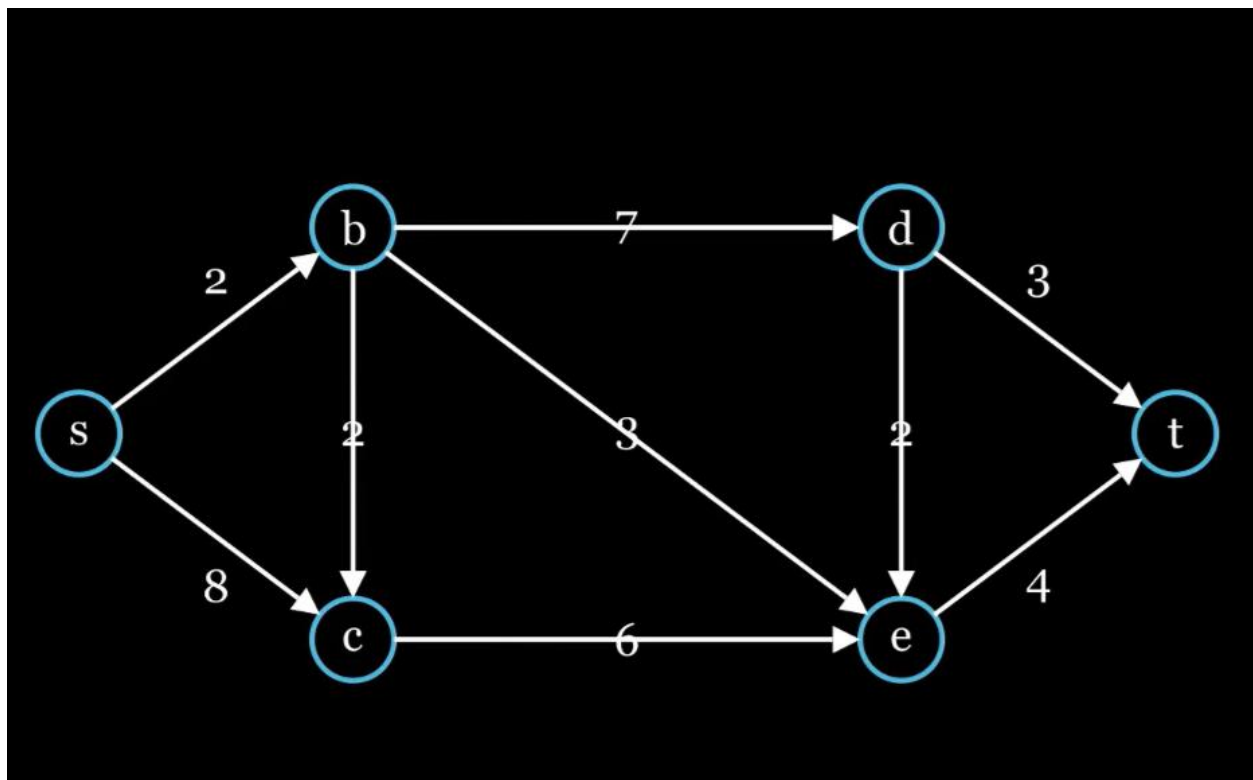
Thực tế, đồ thị chỉ tồn tại cạnh (p,x), tại sao ta vẫn xét cạnh (x,p), vẫn cho đi qua nếu $r > 0$, và sau khi $r[p][x] -= \text{minr}$ thì $r[x][p] += \text{minr}$?

Giải thích:

Ta gọi (x,p) là 1 **cạnh ngược**, và coi nó là 1 cạnh của đồ thị. Khi khởi tạo, $r[x][p] = 0$ nên nó không được duyệt thông qua bfs.

Đôi lúc, đường đi từ s->t theo bfs không phải đường đi tối ưu, ta cần hoàn tác lại nó. Về cơ bản, $r[x][p]$ chính là giá trị đã được sử dụng, cũng là giá trị có thể hoàn trả lại lưu lượng. Hay $c[p][x] = r[p][x] + r[x][p]$ Ví dụ:

Với mạng:

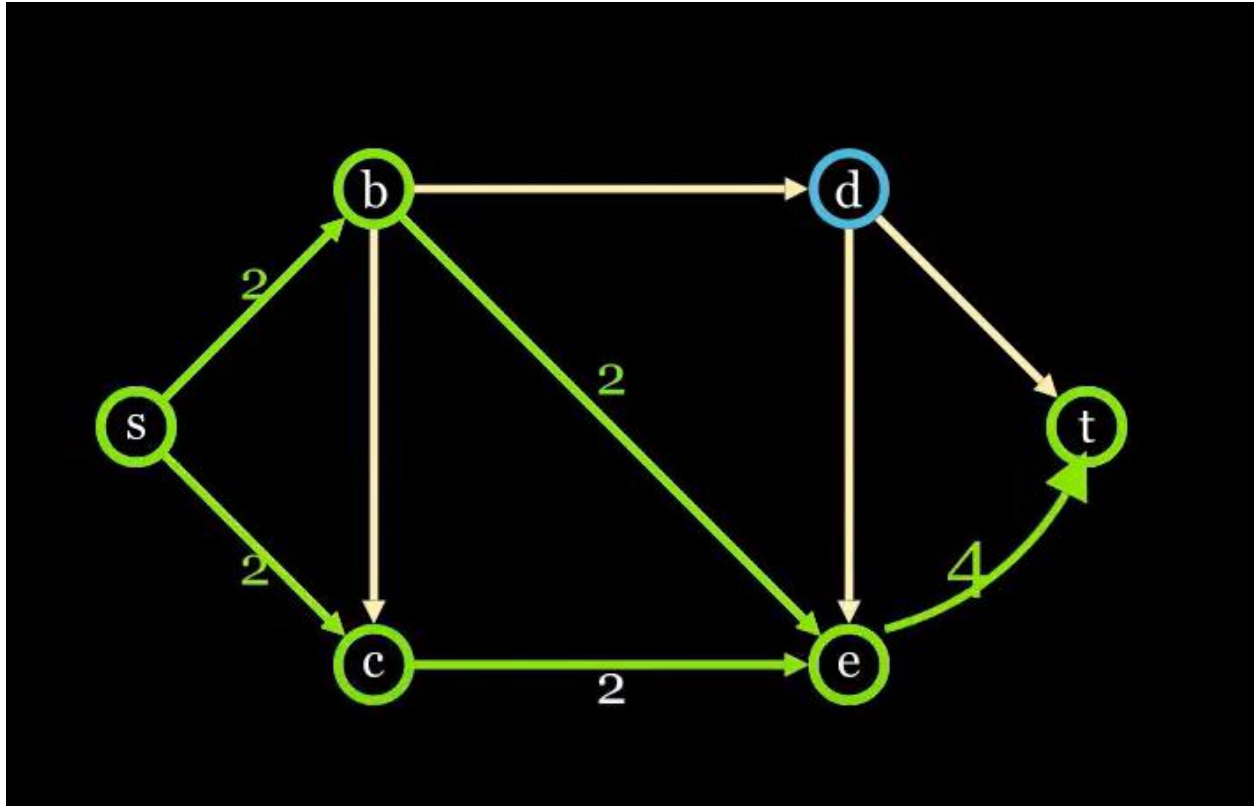


Giả sử ta đã bfs và thu được kết quả sau (đây là 1 luồng):

Với đường đi:

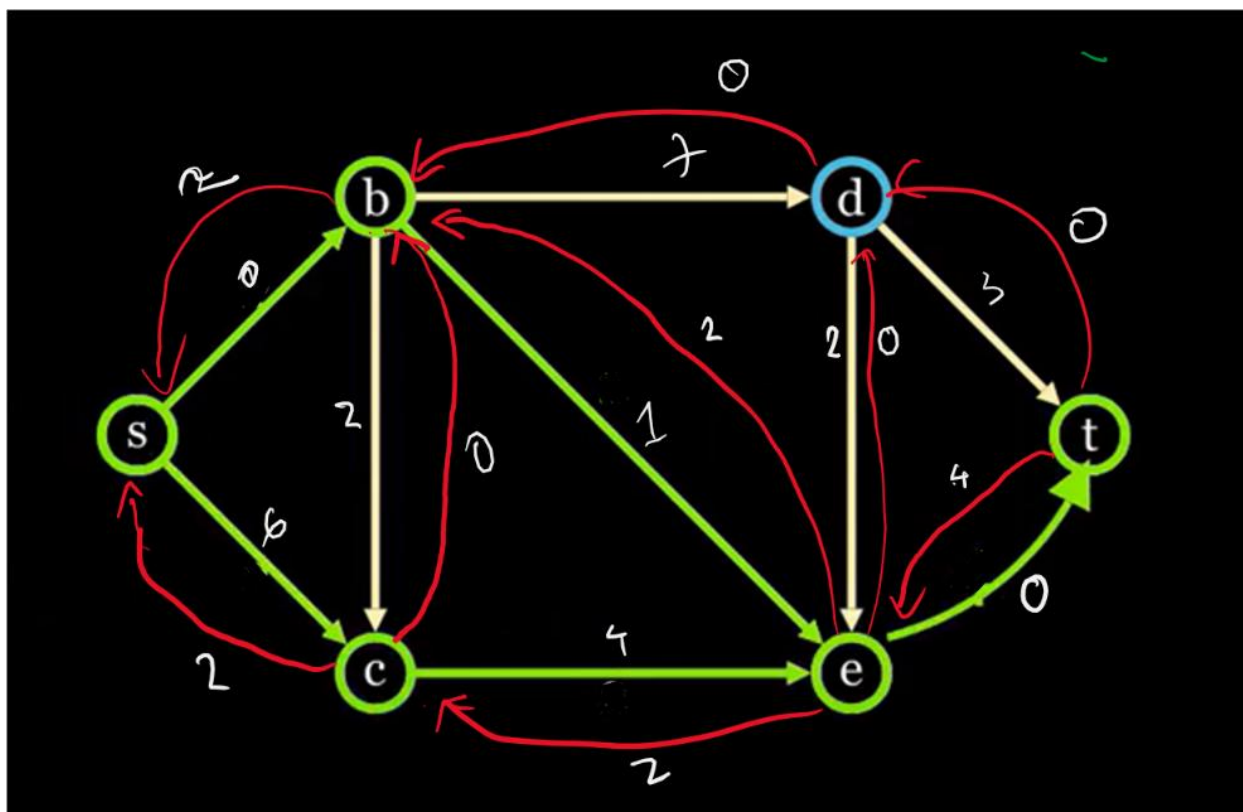
s-b-e-t: 2

s-c-e-t: 2



Thực tế, đã hết đường đi, tuy nhiên đây không phải luồng cực đại.

Đây là đồ thị trên với các giá trị của r:



Đường đi thực tế đã hết, tuy nhiên, nếu xét trên đồ thị trên, ta vẫn còn: s-c-e-b-d-t với $\text{minr}=2$, và giá trị luồng sau khi có đường đi này là 6, chính là luồng cực đại.

Ta có 3 đường đi:

s-b-e-t: 2

s-c-e-t: 2

s-c-e-b-d-t: 2

Tại sao lại cho phép đi qua (e,b) dù thực tế đồ thị không có cạnh này:

Bản chất, đây chính là việc thu hồi lại minr đơn vị từ $b \rightarrow e$, và cho minr đẩy đổi hướng sang $b \rightarrow d$, và đi tiếp đến s, có nghĩa là: s-b-e-t đã được chuyển đổi 1 phần (2 đơn vị) thành s-b-d-t, còn s-b-e-t có thể tồn tại ở thực tế (nếu $r[e][b] > 0$) và ngược lại.

Còn s-c-e-b-d-t, từ e, sẽ đi tiếp phần đường còn lại của s-b-e-t, ở đây là e-t, thành s-c-e-t, mà s-c-e-t đã có sẵn nên ta chỉ việc cộng thêm giá trị.

Các đường đi thực tế:

s-c-e-t: 4

s-b-d-t: 2

Nếu cần biểu thị cụ thể luồng đó, với mỗi đường dẫn (a,b), ta chỉ cần in $r[b][a]$

Demo

Đồ thị trên với 1 là s, 2 là b, 3 là c, 4 là d, 5 là e, 6 là t.

The screenshot shows a C++ IDE with two files: `ly thuyet edmondsKarp.cpp` and `in.inp`. The `in.inp` file contains a 10x6 grid of numbers representing a flow network. The output window shows the maximum flow and the flow values for each edge.

From	To	Flow
1	2	2
1	3	4
2	4	2
3	5	4
4	6	2
5	6	4

Maximum flow: 6

Process exited after 0.1846 seconds
Press any key to continue . .

Độ phức tạp thuật toán

$$O(E^2V)$$

Tài liệu tham khảo

[Edmonds-Karp Algorithm | Brilliant Math & Science Wiki](#)

8. Bài tập liên quan đến thuật toán Edmonds-Karp

Bài 1

Luồng cực đại trên mạng - VNOJ: VNOI Online Judge

Phân tích : đây chính là bài toán tìm luồng cực đại, chỉ cần nộp code trên chắc chắn AC :

Execution Results

✓ ×12

- Test case #1: **Accepted** [0.010s, 2.03 MB] (0/0)
- Test case #2: **Accepted** [0.010s, 2.02 MB] (1/1)
- Test case #3: **Accepted** [0.009s, 3.37 MB] (1/1)
- Test case #4: **Accepted** [0.008s, 3.20 MB] (1/1)
- Test case #5: **Accepted** [0.012s, 4.10 MB] (1/1)
- Test case #6: **Accepted** [0.011s, 4.02 MB] (1/1)
- Test case #7: **Accepted** [0.026s, 8.63 MB] (1/1)
- Test case #8: **Accepted** [0.062s, 16.26 MB] (1/1)
- Test case #9: **Accepted** [0.089s, 19.49 MB] (1/1)
- Test case #10: **Accepted** [0.102s, 19.66 MB] (1/1)
- Test case #11: **Accepted** [0.097s, 19.56 MB] (1/1)
- Test case #12: **Accepted** [0.017s, 2.02 MB] (1/1)

Resources: 0.102s, 19.66 MB

Final score: 11/11 (0.150/0.150 points)

Bài 2

Giao lưu - VNOJ: VNOI Online Judge

✔ Giao lưu

Cuộc thi giao lưu "Tết Ta Tín (TTT)" giữa hai đội Sư Phạm (SP) và Tổng Hợp (TH) có m bài toán tin học, mỗi đội có n học sinh tham dự. Các bài toán được đánh số từ 1 đến m và các học sinh của mỗi đội được đánh số từ 1 tới n .

Học sinh của hai đội đều là những lập trình viên xuất sắc, tuy nhiên mỗi học sinh có thể giải quyết những bài toán thuộc sở trường của mình hiệu quả hơn những bài khác.

Hãy giúp thầy My tổ chức cuộc thi theo thể thức sau:

- Chọn đúng n cặp đấu, mỗi cặp gồm 01 học sinh SP và 01 học sinh TH làm 01 bài toán trong số những bài toán này.
- Có đúng n bài toán được mang ra thi.
- Học sinh nào cũng được tham gia.
- Bài toán cho cặp đấu bất kỳ phải thuộc sở trường của cả hai thí sinh trong cặp.

Biết rằng luôn tồn tại phương án thực hiện yêu cầu trên.

Input

Dòng 1: Chứa hai số n, m ($1 \leq n \leq m \leq 255$).

Trong n dòng tiếp theo, dòng thứ i ghi danh sách các bài toán thuộc sở trường của học sinh SP thứ i .

Trong n dòng tiếp theo, dòng thứ j ghi danh sách các bài toán thuộc sở trường của học sinh TH thứ j .

Output

Gồm m dòng, dòng thứ k ghi số hiệu thí sinh SP và số hiệu thí sinh TH trong cặp đấu bằng bài toán k , nếu bài toán k không được mang ra thi thì ghi vào dòng này hai số 0.

Phân tích:

Ta sẽ sử dụng luồng cực đại để giải quyết bài toán này, cụ thể tất cả sức chứa các cạnh là 1, coi mỗi bài toán là 1 đỉnh, mỗi học sinh là 1 đỉnh.

Có 1 đỉnh phát và 1 đỉnh thu, đỉnh phát sẽ có đường dẫn tới học sinh SP, mỗi học sinh SP sẽ có đường dẫn tới bài toán sở trường, và mỗi bài toán sẽ ghép với học sinh TH có sở trường là bài toán đó, mỗi học sinh TH sẽ có đường dẫn đến đỉnh phát. Điều này sẽ đảm bảo luôn tìm ra cách ghép cặp 1 SP và 1 TH, đồng thời 2 người này sẽ giải 1 bài toán để phân cao thấp.

Cụ thể ở đây sẽ là thuật toán Edmonds-Karp

Do các cạnh chỉ có giá trị là 1 nên chúng ta có thể code đơn giản hơn rất nhiều:

```
ll bfs(){
    f(i,0,t) {
        check[i]=0;
        parent[i]=-1;
    }
    queue<ll>q;
    q.push(s);
    check[s]=1;
    while(!q.empty()){
        ll i=q.front();
        q.pop();
        for(auto x:g[i]){
            if(check[x]==0&&r[i][x]==1){
                q.push(x);
                check[x]=1;
                parent[x]=i;
                if(x==t) return 1;
            }
        }
    }
    return 0;
}
```

```
void edmondsKarp(){
    while(bfs()){
        ll x=t;
        while(1){
            if(x==s) break;
            ll p=parent[x];
            r[p][x]--;
            r[x][p]++;
            x=p;
        }
    }
}
```

Dưới đây là bước khởi tạo, kết nối đỉnh thu, đỉnh phát với các học sinh:

```
int main(){
    //freopen("in.inp","r",stdin);
    //freopen("out.oup","w",stdout);
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cin>>n>>m;
    cin.ignore();
    t=2*n+m+1;
    //sp: 1-n, th: n+1 -> 2n, prob: 2n+1 -> 2n+m
    f(i,1,n){
        g[s].push_back(i);
        g[i].push_back(s);
        g[n+i].push_back(t);
        g[t].push_back(n+i);
        r[s][i]=1;
        r[i][s]=0;
        r[n+i][t]=1;
        r[t][n+i]=0;
    }
```

Còn đây là kết nối học sinh với bài toán:

```
f(i,1,n){
    string s;
    getline(cin,s);
    stringstream ss;
    ss<<s;
    ll a;
    while(ss>>a){
        g[i].push_back(a+2*n);
        g[a+2*n].push_back(i);
        r[i][a+2*n]=1;
        r[a+2*n][i]=0;
    }
}
f(i,1,n){
    string s;
    getline(cin,s);
    stringstream ss;
    ss<<s;
    ll a;
    while(ss>>a){
        g[a+2*n].push_back(n+i);
        g[n+i].push_back(a+2*n);
        r[a+2*n][n+i]=1;
        r[n+i][a+2*n]=0;
    }
}
```

Sau đó gọi hàm Edmonds-Karp và in ra:

```
edmondsKarp();
f(i,1,m){
    ll a=2*n+i;
    ll ans=0;
    f(j,1,n){
        if(r[a][j]==1) {
            ans=j;
            break;
        }
    }
    cout<<ans<< " ";
    ans=0;
    f(j,1,n){
        if(r[j+n][a]==1){
            ans=j;
            break;
        }
    }
    cout<<ans;
    cout<<endl;
}
```

Kết quả:

Execution Results

✓ ✗ -----

Test case #1:	Accepted	[0.009s,	3.52 MB] (0/0)
Test case #2:	Wrong Answer	[0.013s,	7.90 MB] (0/0)

Judge feedback

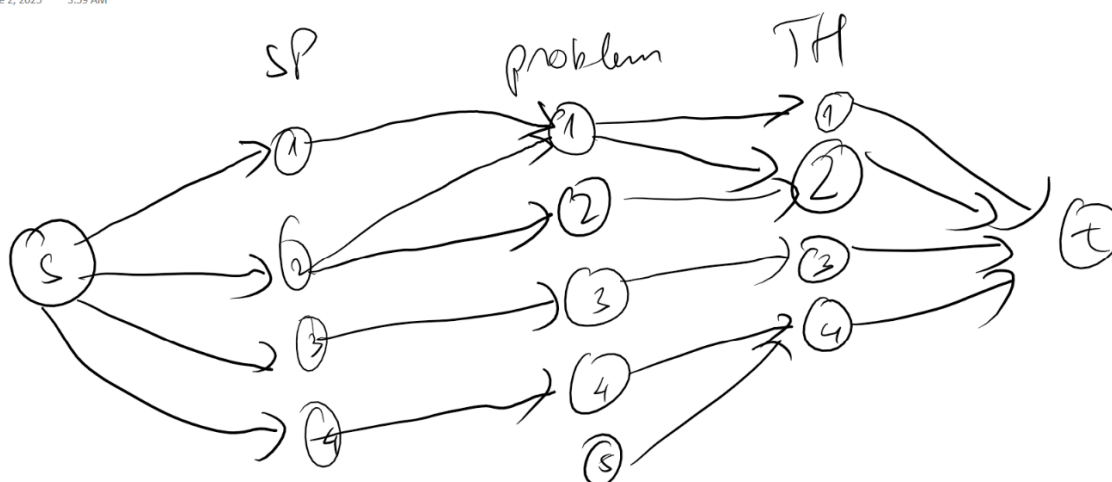
wrong answer Contestant's solution has 236 matches, differs from n = 250.

Test case #3:	—	(0/0)
Test case #4:	—	(0/0)
Test case #5:	—	(0/0)
Test case #6:	—	(0/0)
Test case #7:	—	(0/0)
Test case #8:	—	(0/0)
Test case #9:	—	(0/0)
Test case #10:	—	(0/0)
Test case #11:	—	(0/1)

Sai, chúng ta hãy cùng tìm hiểu nguyên nhân:

Với trường hợp sau:

Monday, June 2, 2025 3:59 AM



Nếu xử lý như trên thì SP1, TH1 và SP2, TH2 sẽ cùng thi đấu với nhau bài 1, cụ thể là cách giải trên sẽ in ra:

1 1

0 0

3 3

4 4

0 0

Mà đề bài yêu cầu có đúng n bài toán được đem ra thi, như vậy là sai, đáp án chính xác phải là:

1 1

2 2

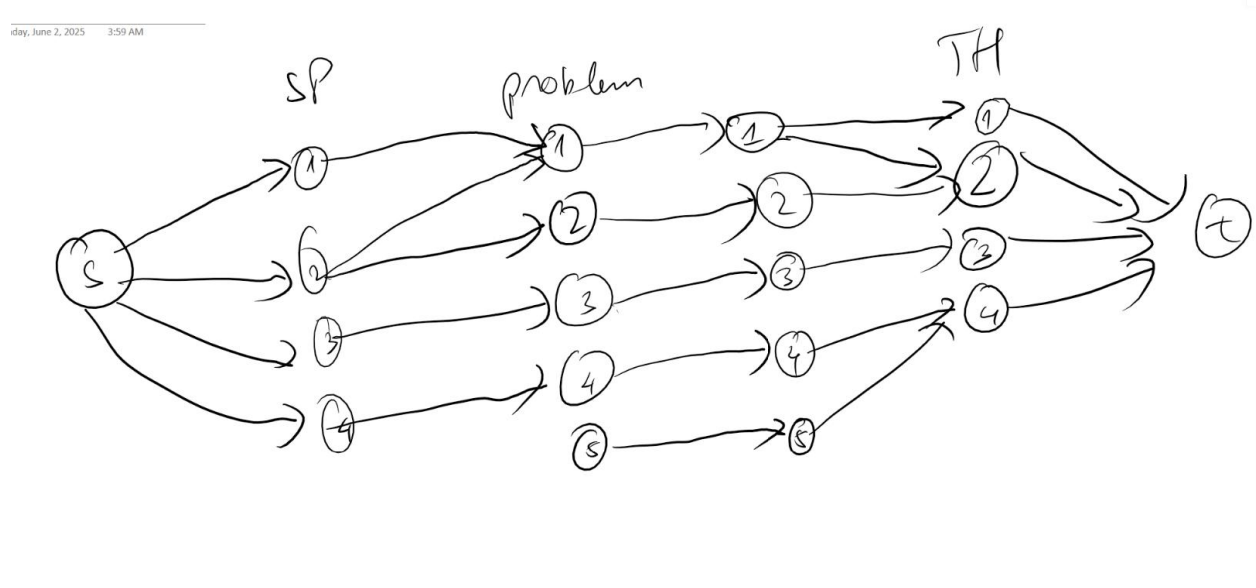
3 3

4 4

0 0

Cách giải quyết:

Ta phải đảm bảo lưu lượng tối đa đi qua các đỉnh đại diện cho các bài toán là 1, vậy nên chỉ cần nhân bản các bài toán và nối các nhân bản (nói hơi khó hiểu nên chúng ta sẽ có hình minh họa của ví dụ trên):



Thiết kế mạng như trên sẽ đảm bảo mỗi bài toán chỉ dành cho tối đa 1 cặp đầu.

Sửa code:

Chú thích :

```
//sp: 1-n, th: n+1 -> 2n, prob1: 2n+1 -> 2n+m, prob2: 2n+m+1 -> 2n+2m
```

Thay đổi t :

```
t=2*n+2*m+1;
```

Sửa phần kết nối bài toán và học sinh TH :

```
f(i,1,n){
    string s;
    getline(cin,s);
    stringstream ss;
    ss<<s;
    ll a;
    while(ss>>a){
        g[a+2*n+m].push_back(n+i);
        g[n+i].push_back(a+2*n+m);
        r[a+2*n+m][n+i]=1;
        r[n+i][a+2*n+m]=0;
    }
}
```

Thêm kết nối bài toán và nhân bản của chính nó :

```
f(i,1,m){
    ll j=2*n+i;
    g[j].push_back(j+m);
    g[j+m].push_back(j);
    r[j][j+m]=1;
    r[j+m][j]=0;
}
```

Sửa phần in ra :

```
f(i,1,m){
    ll a=2*n+i;
    ll ans=0;
    f(j,1,n){
        if(r[a][j]==1) {
            ans=j;
            break;
        }
    }
    cout<<ans<< " ";
    ans=0;
    f(j,1,n){
        if(r[j+n][a+m]==1){
            ans=j;
            break;
        }
    }
    cout<<ans;
    cout<<endl;
}
```

Kết quả :

Execution Results

✓ x11

- > Test case #1: Accepted [0.004s, 3.50 MB] (0/0)
- > Test case #2: Accepted [0.009s, 9.88 MB] (0/0)
- > Test case #3: Accepted [0.003s, 3.34 MB] (0/0)
- Test case #4: Accepted [0.004s, 3.50 MB] (0/0)
- Test case #5: Accepted [0.004s, 3.38 MB] (0/0)
- Test case #6: Accepted [0.006s, 5.82 MB] (0/0)
- Test case #7: Accepted [0.015s, 8.09 MB] (0/0)
- Test case #8: Accepted [0.005s, 6.35 MB] (0/0)
- Test case #9: Accepted [0.011s, 8.86 MB] (0/0)
- Test case #10: Accepted [0.009s, 8.51 MB] (0/0)
- Test case #11: Accepted [0.005s, 7.18 MB] (1/1)

Resources: 0.015s, 9.88 MB
Final score: 1/1 (0.390/0.390 points)

9. Thuật toán Dinic

Khái niệm

Thuật toán Dinic cũng dựa trên phương pháp Ford-Fulkerson.

Có những trường hợp thuật Edmonds-Karp chạy chưa ổn lắm, điển hình là khi mạng có rất nhiều cạnh, ví dụ có dạng của đồ thị đầy đủ với $\frac{V(V-1)}{2}$ cạnh thì độ phức tạp của thuật toán sẽ là $O(V^5)$, rất khủng khiếp. Thuật toán Dinic sẽ làm giảm độ phức tạp của thuật đi một chút.

Thuật toán này được Yefim A. Dinits (nhiều tài liệu để tên là E. A. Dinic) đề xuất năm 1970. Nó được chứng minh là có độ phức tạp $O(EV^2)$ tốt hơn thuật toán Edmonds-Karp.

Năm 1975, Shimon Even và Alon Itai đề xuất 1 giải pháp tối ưu để cải thiện độ phức tạp, trong bài báo cáo này của em chính là mảng pos

Ý tưởng thuật toán Dinic

Thuật toán Dinic là một cải tiến của phương pháp Ford-Fulkerson với 2 giai đoạn chính trong mỗi pha:

1. Xây dựng đồ thị tầng (level graph) bằng BFS từ s. Chỉ giữ lại các cạnh còn khả năng và đi từ cấp thấp hơn đến cao hơn.
2. Tìm đường tăng bằng DFS, chỉ duyệt các cạnh trong đồ thị tầng để đẩy luồng.

Lặp lại 2 bước trên cho đến khi không thể xây dựng đồ thị tầng chứa t.

Code :

```
ll cap[1001][1001], r[1001][1001], lv[1001], pos[1001];  
vector<ll> g[1001];  
ll n, e, s, t;
```

n, e, s, t: số đỉnh, số cạnh, nguồn, đích

Cap: khả năng thông qua của đường dẫn đó

Parent: điểm cha, điểm trước của nó trong đường đi từ s tới t theo phương pháp BFS

r: lưu khả năng thông qua còn lại, sức chứa còn lại của đường dẫn, cạnh đó, bao gồm cả cạnh ngược

Cạnh ngược: nếu tồn tại cạnh (u,v) thì (v,u) chính là cạnh ngược

lv[i]: tầng (level) của đỉnh i trong BFS.

pos[i]: vị trí tiếp theo cần duyệt trong DFS của đỉnh i để tránh duyệt lại.

g: vector lưu danh sách kề của đồ thị.

Hàm bfs(): xây dựng đồ thị tầng: là đồ thị đánh số thứ tự cho độ sâu của đỉnh đó trong cây bfs, với gốc =1

```
void bfs(){
    f(i,1,n) lv[i]=0;
    lv[s]=1;
    queue<ll>q;
    q.push(s);
    while(!q.empty()){
        ll top=q.front();
        q.pop();
        for(auto x:g[top]){
            if(lv[x]==0&& r[top][x]>0){
                q.push(x);
                lv[x]=lv[top]+1;
                if(x==t) return;
            }
        }
    }
}
```

Hàm DFS: Tìm đường tăng luồng.

Do còn phải cập nhật đường tăng luồng, nên sau khi đến t, ta cần cập nhật r, nên không thể đi tiếp sau khi quay lui. Điều này khiến cho ta phải dfs tối đa n lần, tuy nhiên, nếu sử dụng pos, việc loại bỏ các đỉnh không đến được t đã cải thiện đáng kể độ phức tạp.

```
ll inf=1000001;
ll dfs(ll i,ll minr){
    if(i==t) return minr;
    for(;pos[i]<g[i].size();pos[i]++){
        ll j=g[i][pos[i]];
        if(lv[j]!=lv[i]+1) continue;
        if(r[i][j]<=0) continue;
        minr=min(minr,r[i][j]);
        ll x=dfs(j,minr);
        if(x!=0) {
            r[i][j]-=x;
            r[j][i]+=x;
            return x;
        }
    }
    return 0;
}
```

Hàm dinic:

Sau mỗi vòng lặp, sau khi tạo lại đồ thị tăng dựa trên các cạnh ngược mới, nên cũng cần reset pos.

```
ll dinic(){
    ll ans=0;
    while(1){
        bfs();
        if(lv[t]==0) break;
        f(i,1,n) pos[i]=0;
        while(1){
            ll x=dfs(s,inf);
            if(x==0) break;
            ans+=x;
        }
    }
    return ans;
}
```

Main:


```

int main(){
    //freopen("in.inp","r",stdin);
    //freopen("out.oup","w",stdout);
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cin>>n>>e>>s>>t;
    f(i,1,e){
        ll a,b;
        cin>>a>>b;
        cin>>cap[a][b];
        cap[b][a]=0;
        cap[a][a]=0;
        cap[b][b]=0;
        r[a][b]=cap[a][b];
        r[b][a]=0;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    cout<<dinic();
}

```

Demo

The screenshot shows a C++ IDE with three tabs: 'ly thuyet edmondsKarp.cpp', 'in.inp', and 'ly thuyet dinic.cpp'. The 'in.inp' tab displays a flow network with 9 nodes and the following edges and capacities:

From	To	Capacity
1	6	8
1	5	6
2	1	2
2	5	3
3	1	3
3	5	3
4	2	4
4	6	6
5	2	5
5	3	3
6	3	4
6	4	3
7	3	5
7	1	1
8	4	6
8	6	6
9	5	6
9	6	6

The 'ly thuyet dinic.cpp' tab shows the output of the Dinic's algorithm, indicating the maximum flow is 9 and listing the flow values for each edge:

```

Maximum flow: 9
1 -> 2: 5
1 -> 3: 4
2 -> 4: 3
2 -> 5: 2
3 -> 4: 3
3 -> 5: 1
4 -> 6: 6
5 -> 6: 3

```

Độ phức tạp

$O(EV^2)$

10. Bài tập liên quan đến thuật toán Dinic

Bài 1

Luồng cực đại trên mạng - VNOJ: VNOI Online Judge

Phân tích: Đơn giản chỉ cần code lại đúng thuật toán dinic

Execution Results

✓ ×12

- Test case #1: **Accepted** [0.005s, 2.02 MB] (0/0)
- Test case #2: **Accepted** [0.006s, 2.03 MB] (1/1)
- Test case #3: **Accepted** [0.006s, 3.43 MB] (1/1)
- Test case #4: **Accepted** [0.006s, 3.17 MB] (1/1)
- Test case #5: **Accepted** [0.006s, 3.95 MB] (1/1)
- Test case #6: **Accepted** [0.008s, 4.02 MB] (1/1)
- Test case #7: **Accepted** [0.017s, 8.58 MB] (1/1)
- Test case #8: **Accepted** [0.030s, 16.12 MB] (1/1)
- Test case #9: **Accepted** [0.044s, 19.47 MB] (1/1)
- Test case #10: **Accepted** [0.042s, 19.63 MB] (1/1)
- Test case #11: **Accepted** [0.045s, 19.51 MB] (1/1)
- Test case #12: **Accepted** [0.006s, 2.00 MB] (1/1)

Resources: 0.045s, 19.63 MB

Final score: 11/11 (0.150/0.150 points)

So sánh với Edmonds-Karp (bên dưới): tuy cùng AC nhưng Dinic cho thấy tốc độ nhanh hơn ở những testcase lớn, cụ thể từ testcase 8 đến 11.

Execution Results

✓ ×12

- Test case #1: **Accepted** [0.010s, 2.03 MB] (0/0)
- Test case #2: **Accepted** [0.010s, 2.02 MB] (1/1)
- Test case #3: **Accepted** [0.009s, 3.37 MB] (1/1)
- Test case #4: **Accepted** [0.008s, 3.20 MB] (1/1)
- Test case #5: **Accepted** [0.012s, 4.10 MB] (1/1)
- Test case #6: **Accepted** [0.011s, 4.02 MB] (1/1)
- Test case #7: **Accepted** [0.026s, 8.63 MB] (1/1)
- Test case #8: **Accepted** [0.062s, 16.26 MB] (1/1)
- Test case #9: **Accepted** [0.089s, 19.49 MB] (1/1)
- Test case #10: **Accepted** [0.102s, 19.66 MB] (1/1)
- Test case #11: **Accepted** [0.097s, 19.56 MB] (1/1)
- Test case #12: **Accepted** [0.017s, 2.02 MB] (1/1)

Resources: 0.102s, 19.66 MB

Final score: 11/11 (0.150/0.150 points)

Bài 2

Giao lưu - VNOJ: VNOI Online Judge

Ở phần Edmonds – Karp đã phân tích nên chỉ cần thay bằng Dinic là xong:

Do không phải quan tâm đến độ tăng luồng nên code có phần được rút gọn:

```
ll lv[1100], pos[1100];
void bfslv(){
    f(i, 0, t) lv[i]=0;
    lv[s]=1;
    queue<ll> q;
    q.push(s);
    while(!q.empty()){
        ll i=q.front();
        q.pop();
        for(auto x:g[i]){
            if(lv[x]==0 && r[i][x]==1){
                q.push(x);
                lv[x]=lv[i]+1;
            }
        }
    }
}
```

```

ll dfs(ll i){
    if (i==t) return 1;
    for(;pos[i]<g[i].size();pos[i]++){
        ll x=g[i][pos[i]];
        if(lv[x]!=lv[i]+1) continue;
        if(r[i][x]<=0) continue;
        ll y=dfs(x);
        if(y==1){
            r[i][x]--;
            r[x][i]++;
            return 1;
        }
    }
    return 0;
}

```

```

void dinic(){
    while(1){
        bfs1v();
        if(lv[t]==0) break;
        f(i,0,t) pos[i]=0;
        while(1){
            ll x=dfs(s);
            if(x==0) break;
        }
    }
}

```

Kết quả:

Execution Results

✓ x11

> Test case #1: Accepted [0.004s,3.37 MB] (0/0)
 > Test case #2: Accepted [0.007s,9.93 MB] (0/0)
 > Test case #3: Accepted [0.004s,3.35 MB] (0/0)
 Test case #4: Accepted [0.004s,3.48 MB] (0/0)
 Test case #5: Accepted [0.004s,3.36 MB] (0/0)
 Test case #6: Accepted [0.006s,5.82 MB] (0/0)
 Test case #7: Accepted [0.011s,7.98 MB] (0/0)
 Test case #8: Accepted [0.006s,6.54 MB] (0/0)
 Test case #9: Accepted [0.009s,8.66 MB] (0/0)
 Test case #10: Accepted [0.008s,8.63 MB] (0/0)
 Test case #11: Accepted [0.009s,7.24 MB] (1/1)

Resources: 0.011s, 9.93 MB

Final score: 1/1 (0.390/0.390 points)

Tài liệu tham khảo

Sách:

- Giải thuật và lập trình – Lê Minh Hoàng
- Tài liệu giáo khoa chuyên tin – Hồ Sĩ Đàm

Slide:

- Lý thuyết đồ thị 2009 – HCMUS

Website:

- CLB Tin học Việt Nam VNOI: <https://wiki.vnoi.info/>
- [Main Page - Algorithms for Competitive Programming](#)
- [GeeksforGeeks | Your All-in-One Learning Portal](#)

Khoá học trực tuyến:

- [ITMO Academy: pilot course - Codeforces](#) : Từ ITMO, một trong những đại học hàng đầu Liên Bang Nga về Khoa học máy tính.

AI:

- [Chat | Google AI Studio](#)
- [Google Gemini](#)
- [ChatGPT](#)
- [DeepSeek - Into the Unknown](#)

Link Github

[nguyenductripro/TTCS](https://github.com/nguyenductripro/TTCS)