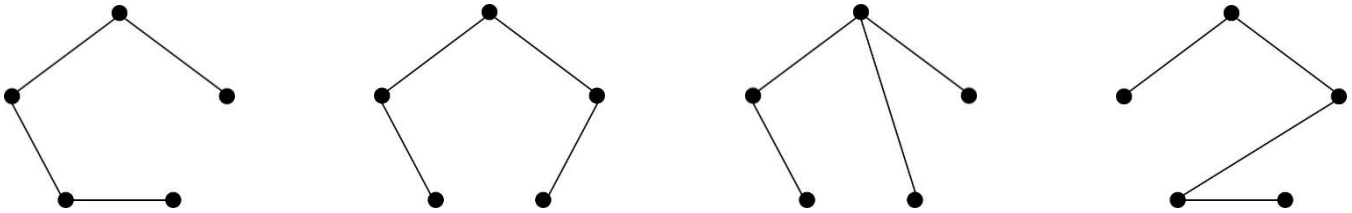
CÂY VÀ CÂY KHUNG

1. Định nghĩa cây

Cây là một đồ thị mà phiên bản vô hướng của nó liên thông và không chứa chu trình. Rừng là một đồ thị không liên thông mà mỗi thành phần liên thông của nó là một cây.

Các hình minh họa cây.



1.1. Định lý:

Cho đồ thị vô hướng G gồm n đỉnh và m cạnh, các tính chất sau là tương đương

(1) G là cây

(2) G liên thông và có $n-1$ cạnh ($m = n-1$)

(3) G không chứa chu trình và có $n-1$ cạnh

(4) G không có chu trình và nếu thêm vào 1 cạnh sẽ sinh ra đúng 1 chu trình

(5) Giữa 2 đỉnh bất kỳ có duy nhất 1 đường đi đơn

(6) G liên thông và mọi cạnh đều là cầu

1.2. Một số khái niệm trên cây

- Lá: đỉnh có bậc 1.
- Gốc: là đỉnh được chọn đại diện cho tập đỉnh thuộc cây, mỗi cây có duy nhất một gốc.
- 2 đỉnh có cạnh nối trực tiếp trên cây có quan hệ cha con, đỉnh cha là đỉnh được duyệt trước trên đường đi từ gốc.
- Chiều cao cây: là số cạnh trên đường đi dài nhất từ gốc đến lá.

Nhận xét:

- Giả sử G là cây gồm $n \geq 2$ đỉnh thì G có ít nhất 2 đỉnh là lá.
- Nếu đồ thị liên thông và có số cạnh ≥ số đỉnh thì đồ thị chứa chu trình.

2. Cây khung

Cho đồ thị vô hướng liên thông $G(V, E)$ gồm n đỉnh và $m(m \geq n-1)$ cạnh. Đồ thị $T(V, E')$ là đồ thị con của G trong đó $E' \subset E$ được gọi là cây khung của đồ thị G nếu T là 1 cây.

Cây T được tạo bằng cách chọn ra $n-1$ cạnh từ m cạnh của G thỏa 1 trong 2 điều kiện sau:

- Liên thông
- Không chứa chu trình

Định lý: Mọi đồ thị liên thông đều tồn tại ít nhất một cây khung.

3. Cây khung nhỏ nhất

Cho đồ thị vô hướng, liên thông $G(V, E)$ gồm n đỉnh và $m(m \geq n-1)$ cạnh. Mỗi cạnh có một trọng số w (trọng số có thể là độ dài đường đi, thời gian đi, chi phí, …). Bài toán đặt ra là tìm một cây khung T của G sao cho tổng trọng số các cạnh của cây T là nhỏ nhất.

Để tìm cây khung nhỏ nhất thì ý tưởng chung là xây dựng dần cây khung T bằng cách chọn ra $n-1$ cạnh có trọng số nhỏ nhất sao cho T liên thông hoặc T không chứa chu trình.

3.1. Thuật toán Prim

Ý tưởng: Xây dựng cây khung T bằng cách chọn ra $n-1$ cạnh có trọng số nhỏ nhất và luôn duy trì đồ thị liên thông trong quá trình xây dựng cây T.

Thuật toán sẽ thực hiện $n-1$ bước, mỗi bước đưa vào cạnh nhỏ nhất nối giữa một đỉnh bên ngoài cây T và một đỉnh thuộc cây T.

3.2. Thuật toán Kruskal

Ý tưởng: Xây dựng cây khung T bằng cách chọn ra $n-1$ cạnh có trọng số nhỏ nhất sao cho các cạnh không tạo thành chu trình.

Thuật toán

- Sắp xếp các cạnh có thứ tự tăng dần theo trọng số $w_1 \leq w_2 \leq \cdots \leq w_m$
- Khởi tạo cây $T = \{\emptyset\}$
- Lần lượt xét các cạnh theo thứ tự đã sắp để chọn ra $n-1$ cạnh theo quy tắc: với mỗi cạnh e_i, nếu e_i không tạo thành chu trình với các cạnh của cây T thì thêm e_i vào cây T.

3.2.1. Cài đặt

Nhận xét rằng nếu 2 đỉnh u và v thuộc cùng một miền liên thông thì khi thêm cạnh (u, v) vào cây sẽ tạo thành chu trình. Sử dụng mảng $lab[1..n]$ để gán nhãn cho các đỉnh, $lab[u]$ cho biết số hiệu của miền liên thông chứa đỉnh u.

Nếu $lab[u] = lab[v]$ thì u, v cùng miền liên thông do đó không thể thêm cạnh (u, v) vào cây. Ngược lại nếu $lab[u] \neq lab[v]$ thì thêm cạnh (u, v) vào cây và hợp nhất nhãn của 2 miền liên thông này.

Ban đầu mỗi đỉnh được xem như một thành phần liên thông riêng biệt, khởi tạo $lab[u] = u, \forall u$.

```
struct edge
{
    int u, v, w, id;
};

edge e[maxM]; //danh sách cạnh
vector<int> T; //cây khung T

void UpdateLabel(int u, int v)
{
    int labu = lab[u], labv = lab[v];
    for (int i = 1; i <= n; ++i)
        if (lab[i] == labu) lab[i] = labv;
}

bool cmp(edge e1, edge e2)
{
    return (e1.w < e2.w);
}

void Kruskal()
{
    sort(e, e+m, cmp);
    for (int u = 1; u <= n; ++u) lab[i] = u;
    for (int i = 0; i < m && T.size() < n-1; ++i)
    {
```

```
        int u = e[i].u, v = e[i].v, id = e[i].id;
        if (lab[u] != lab[v])
        {
            T.push_back(id);
            UpdateLabel(u, v);
        }
    }
}
```

Độ phức tạp cài đặt $O(n^2)$

3.2.2. Cải tiến thuật toán với CTDL Disjoint Set-Union

Nhận xét rằng thuật toán phải thực hiện $n-1$ bước để thêm $n-1$ cạnh vào cây khung nên không thể giảm số bước thực hiện. Do vậy, thuật toán chỉ có thể cải tiến ở thao tác hợp nhất nhãn của 2 miền liên thông. Mặt khác, trong quá trình xây dựng cây khung, mỗi miền liên thông chính là một cây. Do đó, nếu chọn một đỉnh làm gốc cho mỗi cây thì thao tác hợp nhất 2 cây chỉ cập nhật nhãn của 2 đỉnh gốc thay vì phải cập nhật toàn bộ đỉnh trong cây. Khi đó, việc xác định 2 đỉnh có cùng miền liên thông được thực hiện bằng cách lần ngược lên đến gốc để kiểm tra 2 đỉnh có cùng gốc hay không.

Để giúp thực hiện quá trình lần ngược, ta sử dụng mảng $par[1..n]$ với ý nghĩa $par[u]$ là đỉnh cha trực tiếp của đỉnh u. Ban đầu, mọi đỉnh có cha là chính nó, khởi tạo $par[u] = u, \forall u$

Hàm lần ngược theo đỉnh cha để tìm gốc của cây chứa đỉnh u

```
int FindRoot(int u)
{
    while (par[u] != u)
        u = par[u];
    return u;
}
```

Nhận thấy rằng tính hiệu quả của thao tác lần ngược phụ thuộc vào chiều cao cây. Do vậy, sau khi tìm được r là gốc của đỉnh u, ta cần thực hiện thao tác nén đường (*path compression*), nghĩa là cho r là cha trực tiếp của u: $par[u] = r$. Điều này giúp việc lần ngược được thực hiện hiệu quả hơn ở những lần sau.

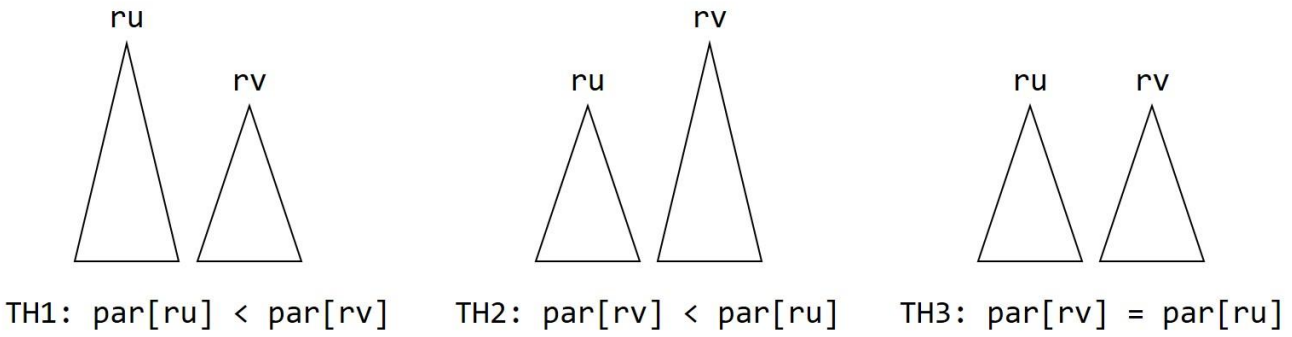
Mặt khác, để góp phần làm giảm số bước ở quá trình lần ngược, việc hợp nhất 2 cây cần hạn chế chiều cao của cây tăng lên. Để thực hiện điều này, mảng par được thay đổi với ý nghĩa như sau:

- $par[u]$ vừa là đỉnh cha trực tiếp của u, vừa là hạng của u. Gốc có hạng càng thấp thì cây càng cao. Khi hợp nhất 2 cây, nếu chiều cao của cây tăng lên thì hạng của gốc sẽ giảm đi. Do vậy $par[u]$ có thể nhận giá trị âm.
  + Khởi tạo mỗi đỉnh đều là một gốc và chiều cao 0 nên $par[u] = 0, \forall u$.
  + Nếu $par[u] > 0$ thì $par[u]$ là cha trực tiếp của u và $par[u]$ không phải là chiều cao cây gốc u.
  + Nếu $par[u] \leq 0$ thì u là gốc. Khi đó $|par[u]|$ là chiều cao của cây gốc u.

Cài đặt hàm `FindRoot` để tìm gốc của một đỉnh

```
int FindRoot(int u)
{
    if (par[u] <= 0)
        return u;
    return par[u] = FindRoot(par[u]); //nén đường
}
```

Việc hợp nhất 2 cây gốc ru và rv được minh họa theo 3 trường hợp sau:



Ở trường hợp 1 thì `ru` được chọn làm gốc của cây hợp nhất, các trường hợp còn lại thì `rv` được chọn làm gốc. Chỉ có trường hợp 3 mới làm cây hợp nhất tăng chiều cao, nghĩa là `par[rv]` sẽ giảm giá trị đi 1.

```
void Union(int ru, int rv)
```

```
{
    //nếu chiều cao của cây ru lớn hơn chiều cao của cây rv
    if (par[ru] < par[rv])
        par[rv] = ru; //ru là cha trực tiếp của rv
    else
    {
        par[rv] = min(par[rv], par[ru]-1);
        par[ru] = rv; //rv là cha trực tiếp của ru
    }
}

void BuildTree()
{
    sort(e, e+m, cmp);
    for (int i = 0, i < m && T.size() < n-1; ++i)
    {
        int u = e[i].u, v = e[i].v, id = e[i].id;
        int ru = FindRoot(u), rv = FindRoot(v);
        if (ru != rv)
        {
            T.push_back(id);
            Union(ru, rv);
        }
    }
}
```

Độ phức tạp cài đặt của thao tác (sau khi sắp xếp danh sách cạnh) trong trường hợp xấu nhất là $O(n \log n)$