DISJOINT SET UNION

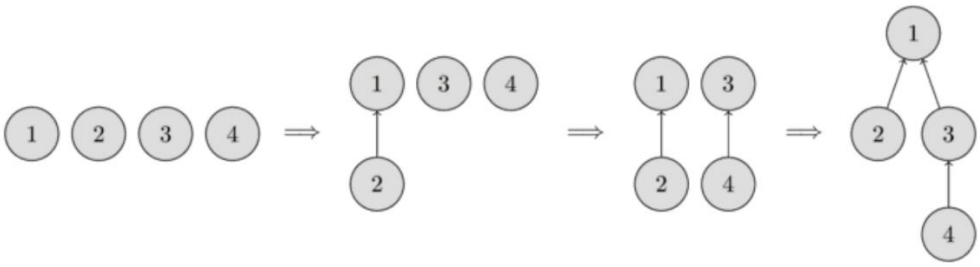
Bài viết này thảo luận về cấu trúc dữ liệu Disjoint Set Union hoặc DSU. Thường nó còn được gọi là Union Find vì hai hoạt động chính của nó.
Chúng ta có thể mô tả cấu trúc này như sau: Cho một số tập hợp con riêng biệt. Một DSU sẽ có một phép toán kết hợp hai tập hợp bất kỳ, mỗi tập hợp có thể chỉ ra một phần tử thuộc tập hợp đó. Phiên bản cổ điển cũng cho phép tạo ra một tập hợp mới.
Do đó, giao diện cơ bản của cấu trúc này chỉ bao gồm ba hoạt động:

- **make_set(v):** Tạo ra một tập hợp mới từ phần tử ban đầu v
- **union_sets(a, b):** Trộn hai tập hợp thành một tập hợp (gồm các phần tử của tập hợp a và các phần tử của tập hợp b)
- **find_set(v):** Trả về phần tử đại diện của tập hợp chứa phần tử v. Đại diện của một phần tử là một phần tử trong tập hợp đó. Phần tử đại diện sẽ được chọn trong chính cấu trúc dữ liệu và có thể thay đổi theo thời gian. Hàm này cũng cho phép kiểm tra hai phần tử u và v có cùng một tập hợp hay không (bằng cách kiểm tra find_set(u)==find_set(v))

Như mô tả ở dưới đây, cấu trúc dữ liệu cho phép thời gian thực hiện trung bình của mỗi thao tác trong thời gian gần như O(1).
Cũng có một phiên bản khác của DSU. Cấu trúc này có thời gian trung bình chậm hơn là O(logn) nhưng có thế mạnh hơn DSU thông thường.

Xây dựng cấu trúc dữ liệu hiệu quả

Chúng ta sẽ mô tả các tập hợp dưới dạng một cây, mỗi cây tương ứng với một tập hợp. Gốc của cây sẽ là phần tử đại diện của tập hợp:



Ban đầu mỗi phần tử riêng biệt là một tập hợp duy nhất, do đó mỗi đỉnh là cây của tập hợp chứa nó. Sau đó ta kết hợp tập hợp chứa phần tử 1 và tập hợp chứa phần tử 2; tiếp theo kết hợp tập hợp chứa phần tử 3 và tập hợp chứa phần tử 4; Cuối cùng kết hợp tập hợp chứa phần tử 1 và tập hợp chứa phần tử 3.

Triển khai giản đơn

Chúng ta có thể viết phần triển khai đầu tiên của cấu trúc dữ liệu Disjoint Set Union. Lúc đầu nó khá kém hiệu quả, nhưng sau này chúng ta có thể cải thiện nó bằng cách sử dụng hai cách tối ưu hóa, do đó sẽ mất thời gian gần như liên tục cho mỗi lần gọi hàm.
Như đã nói, tất cả thông tin về tập hợp sẽ được lưu trữ trong một mảng (**parent**).
Để tạo một tập hợp mới (make_set(v)) chúng ta chỉ cần tạo một cây có gốc tại đỉnh v, nghĩa là nó là tổ tiên của chính nó.
Để kết hợp hai tập hợp (toán tử make_set(a,b)), trước tiên chúng ta tìm đại diện của tập hợp chứa a và tìm đại diện của tập hợp chứa b.Nếu hai đại diện này giống hệt nhau, ta không phải làm gì vì hai tập hợp là một. Nếu không ta có thể chỉ định một cách đơn giản rằng một trong hai đại diện là cha của đại diện kia - do đó hai tập hợp được hợp nhất.

Cuối cùng, việc tìm đại diện (phép tính find_set(v)): Chúng ta chỉ cần leo lên đỉnh tổ tiên của đỉnh v cho đến khi chúng ta đi được đến đỉnh gốc, tức là đỉnh mà tổ tiên của nó là chính nó. Thao tác này có thể dễ dàng thực hiện bằng cách gọi đệ qui.

```
void make_set(int v) {
    parent[v] = v;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return find_set(parent[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b)
        parent[b] = a;
}
```

Tuy vậy, việc thực hiện như trên không hiệu quả. Có thể dễ dàng xây dựng được ví dụ trong đó cây thoái hóa thành một chuỗi dài. Trong trường hợp đó, mỗi lần gọi find_set(v) có thể mất thời gian là O(n).

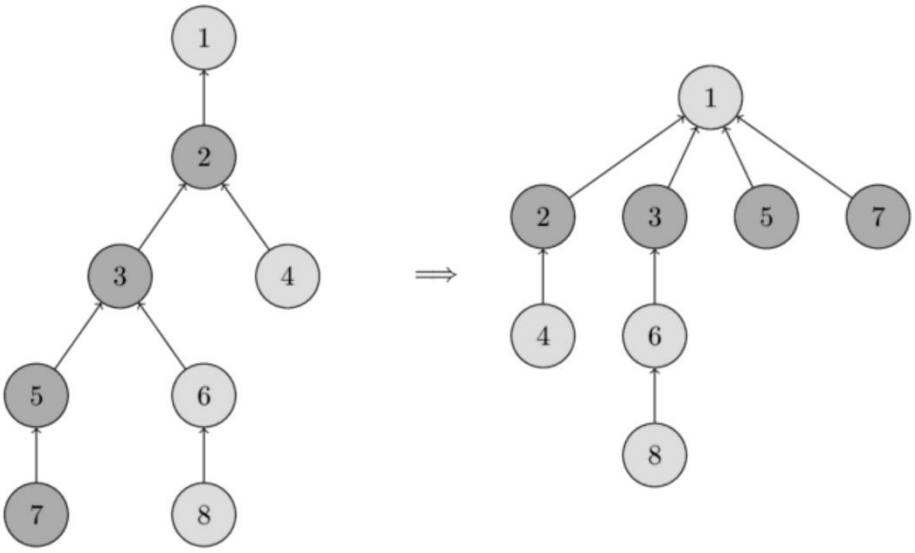
Điều này khác xa với độ phức tạp mà chúng ta muốn có (thời gian gần như không đổi). Do đó chúng ta sẽ xem xét hai cách tối ưu hóa dưới đây để đẩy nhanh thời gian thực hiện các hàm một cách đáng kể.

Tối ưu hóa nén đường dẫn

Tối ưu này xây dựng nhằm để tăng tốc hàm find_set.

Nếu ta gọi find_set(v) cho một đỉnh v, thực sự chúng ta đã tìm thấy đại diện p cho tất cả các đỉnh nằm trên đường đi từ v đến p. Bí quyết là làm cho đường dẫn của tất cả các nút này ngắn hơn bằng cách đặt trực tiếp p là đỉnh cha cho tất cả các đỉnh nằm trên đường đi từ v đến p.

Hình dưới đây là một ví dụ. Ở bên trái là cây ban đầu còn ở bên phải là cây bị nén sau khi gọi find_set(7), nó sẽ rút ngắn các đường dẫn cho các nút 7, 5, 3 và 2.



Phiên bản mới của find_set có thể được viết như dưới đây:

```
int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
```

Việc triển khai đơn giản như dự định: Đầu tiên tìm đại diện cho chính đỉnh gốc, sau đó trong quá trình trả về đệ qui, các nút được thăm sẽ được gắn trực tiếp với gốc.
Sử đổi trên đã cho được độ phức tạp trung bình giảm xuống còn O(logn) cho mỗi lần gọi. Sửa đổi thứ hai sẽ làm cho tốc độ nhanh hơn.

Hợp nhất theo kích cỡ/cấp bậc

Trong cách tối ưu hóa này ta sẽ thay đổi hoạt động của union_set. Cụ thể ta sẽ thay đổi cách mà đại diện của cây được gán làm cha đại diện của cây còn lại. Trong phiên bản ban đầu, cây thứ hai luôn được gắn vào cây thứ nhất. Trên thực tế điều này có thể dẫn đến việc cây thành chuỗi có độ dài O(n). Với sự tối ưu hóa này, chúng ta tránh điều này bằng cách lựa chọn cẩn thận việc cây nào được gán vào cây nào.
Có nhiều phương pháp phỏng đoán được sử dụng. Phổ biến nhất là hai cách tiếp cận sau: Trong cách tiếp cận đầu tiên chúng ta sẽ sử dụng kích cỡ (số lượng đỉnh) làm thứ hạng và trong cách tiếp cận thứ hai, chúng ta sử dụng độ sâu của cây (chính xác hơn là giới hạn trên của độ sâu cây, vì độ sâu sẽ nhỏ hơn khi áp dụng nén đường dẫn)
Trong cả hai cách tiếp cận, bản chất của việc tối ưu hóa là giống nhau: Chúng ta gắn cây có thứ hạng thấp hơn vào cây có thứ hạng lớn hơn.
Dưới đây là việc kết hợp theo kích thước:

```
void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
```

Và đây là việc kết hợp dựa trên độ sâu của cây:

```
void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
```

```
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

Cả hai cách tối ưu trên đều tương đương về độ phức tạp về thời gian và bộ nhớ nên trong thực tế, bạn có thể sử dụng bất kỳ cách nào cũng được.

Độ phức tạp

Như đã đề cập ở trên, nếu chúng ta kết hợp cả hai tối ưu hóa - nén đường và kết hợp theo kích thước/độ sâu chúng ta sẽ đạt hiệu quả thời gian gần như không đổi. Người ta đã chứng minh được độ phức tạp là $O(\alpha(n))$ trong đó $\alpha(n)$ là nghịch đảo hàm Akermann tăng rất chậm. Trên thực tế nó không vượt quá 4 với mọi $n \leq 10^{600}$.
Ngoài ra điều đáng nói là DSU với kết hợp theo kích thước/độ sâu sẽ có thời gian thực hiện trung binh các thao tác là O(logn).

Tìm các thành phần liên thông của đồ thị

Đây là ứng dụng hiển nhiên của DSU
Cụ thể: Ban đầu chúng ta có một đồ thị rỗng. Chúng ta phải thêm dần các đỉnh và các cạnh vô hướng và trả lời các truy vấn dạng (a,b) - "các đỉnh a và b có cùng nằm trong một thành phần liên thông của đồ thị không?"
Ở đây, chúng ta có thể xây dựng trực tiếp cấu trúc dữ liệu DSU và nhận được giải pháp xử lý việc bổ sung thêm một đỉnh hoặc một cạnh và thực hiện truy vấn trong khoảng thời gian trung bình gần như không đổi.
Ứng dụng này thường hay áp dụng để tìm cây khung của đồ thị:

- Đầu tiên ta coi mỗi đỉnh của đồ thị là tập hợp chỉ gồm duy nhất đỉnh đó
- Khi thêm cạnh (u,v) nếu u và v thuộc hai tập hợp khác nhau thì tiến hành kết hợp hai tập hợp này thành một. Khi đó (u,v) là một cạnh của cây khung tìm được

Chương trình dưới đây tìm cây khung đồ thị dựa theo danh sách cạnh. DSU được sử dụng bằng cách kết hợp cả hai kỹ thuật nén đường và kết hợp theo kích thước:

```
// Input:
pair<int,int> E[maxm];                  // Mảng các cạnh
int m;                                  // Số cạnh
// Output:
int InTree[maxm];                       // InTree[k]=1/0 ứng với thuộc/không thuộc cây

// Code:
int parent[maxn];

void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}
int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
```

```
int union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
        return 1;
    }
    return 0;
}

int main() {
    ....
    for(int u = 1; u <= n; ++u) make_set(u);
    for(int i = 1; i <= m; ++i) {
        int u = E[i].first, v = E[i].second;
        InTree = union_sets(u, v);
    }
}
```

Nếu như các cạnh được sắp xếp theo trọng số tăng dần ta thu được một cây khung tối tiểu (cây khung có tổng trọng số các cạnh là bé nhất). Cây khung này còn được gọi là cây khung Kruskal. Hai tính chất cần ghi nhớ của cây khung Kruskal:

- Tổng trọng số các cạnh của cây khung là bé nhất (cây khung tối tiểu)
- Giá trị lớn nhất của cạnh tham gia vào cây khung là bé nhất

Nếu sắp xếp các cạnh theo giá trị trọng số giảm dần ta cũng thu được một cây khung. Cây khung này có tính chất là **giá trị nhỏ nhất của cạnh tham gia vào cây khung là lớn nhất**.