Disjoint Set Union (DSU) – Hợp Tập Hợp Rời Rạc

Nguyễn Quản Bá Hồng*

Ngày 22 tháng 8 năm 2025

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Disjoint Set Union (DSU) – Hợp Tập Hợp Rời Rạc.*
  PDF: URL: .pdf.
  TEX: URL: .tex.
- .
  PDF: URL: .pdf.
  TEX: URL: .tex.

Mục lục

1 Introduction to Disjoint Set Union – Nhập Môn Hợp Tập Hợp Rời Rạc **1**
1.1 Data Structure Disjoint Set Union – Cấu trúc dữ liệu Disjoint Set Union 2
1.1.1 Naive implementation of DSU – Cài đặt "ngây thơ" của DSU 3
1.1.2 1st Optimization: Merge according to size/height – Tối ưu 1: Gộp theo kích cỡ/độ cao 3
1.1.3 2nd Optimization: Path compression – Tối ưu 2: Nén đường đi 4
1.2 Time complexity & its proof – Độ phức tạp thời gian & chứng minh 4

2 Some Applications of DSU – Vài Ứng Dụng của DSU **4**

3 Miscellaneous **4**

1 Introduction to Disjoint Set Union – Nhập Môn Hợp Tập Hợp Rời Rạc

Resources – Tài nguyên.

1. Algorithms for Competitive Programming/disjoint set union.
2. Benjamin Qi, Andrew Wang, Nathan Gong, Michael Cao. USACO Guide/Disjoint Set Union.

   Abstract. The Disjoint Set Union (DSU) data structure, which allows you to add edges to a graph & test whether 2 vertices of a graph are connected.

   – Cấu trúc dữ liệu Disjoint Set Union (DSU), cho phép bạn thêm các cạnh vào đồ thị & kiểm tra xem 2 đỉnh của đồ thị có được kết nối hay không.
3. Ngô Quang Nhật. VNOI Wiki/Disjoint Set Union.
4. Wikipedia/disjoint-set data structure.

Disjoint Set Union (abbr., DSU) là 1 cấu trúc dữ liệu hữu dụng, thường xuất hiện trong các kỳ thi Lập trình Thi Đấu, & có thể được dùng để quản lý 1 cách hiệu 1 tập hợp của các tập hợp.

Bài toán 1. *Cho 1 đồ thị $G = (V, E)$ có $|V| = n \in \mathbb{N}^\star$ đỉnh, ban đầu không có cạnh nào, $E = \emptyset$. Ta cần xử lý 2 loại truy vấn:*

1. *Thêm 1 cạnh giữa 2 đỉnh $x, y \in V$ trong đồ thị, i.e., $E = E \cup \{\{x, y\}\}$ nếu G là đồ thị vô hướng & $E = E \cup \{(x, y)\}$ nếu G là đồ thị có hướng; tuy nhiên DSU thường chỉ áp dụng cho đồ thị vô hướng nên ta chỉ xét trường hợp đồ thị vô hướng cho đơn giản.*
2. *In ra* `yes` *nếu như 2 đỉnh x, y nằm trong cùng 1 thành phần liên thông. In ra* `no` *nếu ngược lại.*

*A scientist- & creative artist wannabe, a mathematics & computer science lecturer of Department of Artificial Intelligence & Data Science (AIDS), School of Technology (SOT), UMT Trường Đại học Quản lý & Công nghệ TP.HCM, Hồ Chí Minh City, Việt Nam.
E-mail: nguyenquanbahong@gmail.com & hong.nguyenquanba@umt.edu.vn. Website: https://nqbh.github.io/. GitHub: https://github.com/NQBH.

1.1 Data Structure Disjoint Set Union – Cấu trúc dữ liệu Disjoint Set Union

Cho tiện, ta đánh số n đỉnh của đồ thị G bởi $1, 2, \ldots, n$ (trường hợp n đỉnh được dán nhãn bởi $v_1, v_2, \ldots, v_n$ hoàn toàn tương tự vì ta có thể làm việc trên chỉ số i của v_i), khi đó $V = [n]$. Giả sử G có $c \coloneqq$ `num_connected_component` $\in \mathbb{N}^\star$ (số thành phần liên thông) $C_1, C_2, \ldots, C_c$ với $\{C_i\}_{i=1}^{c}$ là 1 phân hoạch của $V = [n]$, i.e.:

$$\bigcup_{i=1}^{c} C_i = [n],\ C_i \cap C_j = \emptyset,\ \forall i, j \in [c],\ i \neq j.$$

Nếu ta coi mỗi đỉnh của đồ thị $G = (V, E)$ là 1 phần tử & mỗi thành phần liên thông (connected component) trong đồ thị là 1 tập hợp, truy vấn 1 sẽ trở thành gộp 2 tập hợp lần lượt chứa phần tử x, y thành 1 tập hợp mới & truy vấn 2 trở thành việc hỏi 2 phần tử x, y có nằm trong cùng 1 tập hợp không.

Để tiện tính toán & lý luận về mặt toán học cho riêng cấu trúc dữ liệu DSU, sau đây là 1 định nghĩa lai Toán–Tin mang tính cá nhân của tác giả [NQBH], hoàn toàn không chính thống trong Lý thuyết Đồ thị:

Định nghĩa 1 (Chỉ số thành phần liên thông)**.** *Cho đồ thị vô hướng $G = (V, E)$ với $V = [n]$. Gọi $C(i) \subset [n]$ là thành phần liên thông của $G = (V, E)$ chứa đỉnh $i \in [n]$ & gọi chỉ số của thành phần liên thông chứa đỉnh i là* $\mathrm{cid}(i)$*, i.e., $i \in C_{\mathrm{cid}(i)} \equiv C(i)$, với hàm* $\mathrm{cid} : [n] \to [c]$ *được gọi là* hàm chỉ số liên thông*.*

Với định nghĩa 1, ta có ngay

$$\begin{cases} i \in C(i) = C_{\mathrm{cid}(i)} \subset [n], \\ i, j \text{ are connected},\ \forall j \in C(i). \end{cases} \quad \forall i \in [n].$$

Ở đây, ta coi mỗi đỉnh của đồ thị tự liên thông với chính nó theo nghĩa đỉnh đó đến được (reachability) chính đỉnh đó thông qua 1 đường đi có độ dài 0, được gọi là 1 *đường đi tầm thường*, chứ không phải theo nghĩa khuyên (loop).

Lemma 1 (A characterization of connectedness – 1 đặc trưng hóa của tính liên thông)**.** *Cho đồ thị vô hướng $G = (V, E)$. (i) 2 đỉnh trên 1 đồ thị G không liên thông với nhau, i.e., không có đường đi nào trên G nối chúng khi & chỉ khi chúng thuộc 2 thành phần liên thông khác nhau, i.e.,*

$$\boxed{i, j \text{ are not connected} \Leftrightarrow C(i) \neq C(j) \Leftrightarrow C(i) \cap C(j) = \emptyset \Leftrightarrow \mathrm{cid}(i) \neq \mathrm{cid}(j),\ \forall i, j \in [n].}$$

(ii) 2 đỉnh trên đồ thị G liên thông với nhau, i.e., có đường đi trên trên G nối chúng khi & chỉ khi chúng cùng thuộc 1 thành phần liên thông, i.e.:

$$\boxed{i, j \text{ are connected} \Leftrightarrow C(i) = C(j) \Leftrightarrow C(i) \cap C(j) \neq \emptyset \Leftrightarrow \mathrm{cid}(i) = \mathrm{cid}(j),\ \forall i, j \in [n].}$$

Với truy vấn 1, giả sử 2 đỉnh $i, j \in [n]$ chưa có cạnh nối chúng trực tiếp, i.e., $\{i, j\} \notin E$. Có 2 trường hợp xảy ra:

- Trường hợp 1: Giả sử i, j cùng thuộc 1 thành phần liên thông, theo Lem. 1, có $C(i) = C(j)$, $\mathrm{cid}(i) = \mathrm{cid}(j)$ nên ta chỉ cần thêm cạnh $\{i, j\}$ vào tập cạnh E: $E \leftarrow E \cup \{\{i, j\}\}$ hay `edge_list.append({i, j})` mà không cần cập nhật c tập liên thông $\{C_i\}_{i=1}^{c}$ hay hàm chỉ số liên thông $\mathrm{cid}(\cdot)$.
- Trường hợp 2: Giả sử i, j thuộc 2 thành phần liên thông khác nhau, theo 1, có $C(i) \neq C(j)$, $\mathrm{cid}(i) \neq \mathrm{cid}(j)$. Khi đó, việc nối cạnh i, j với nhau, i.e., cập nhập tập cạnh bằng cách thêm cạnh $\{i, j\}$ vào tập cạnh E: $E \leftarrow E \cup \{\{i, j\}\}$ hay `edge_list.append({i, j})`, nhưng điều khác biệt ở đây nằm ở chỗ: ta cũng cần phải cập nhật tập các thành phần liên thông & hàm chỉ số liên thông như sau:
  1. Cập nhật thành phần liên thông chứa cả i & j bằng cách lấy hợp của 2 tập hợp $C(i), C(j)$, i.e., $C_{\mathrm{new}}(i) = C_{\mathrm{new}}(j) \coloneqq C(i) \cup C(j)$.
  2. Cập nhật hàm chỉ số liên thông: $\mathrm{cid}_{\mathrm{new}}(i) = \mathrm{cid}_{\mathrm{new}}(j) = \min\{\mathrm{cid}(i), \mathrm{cid}(j)\}$ (cũng có thể lấy max thay vì min nhưng theo quy ước của DSU data structure, ta nên lấy min để đảm bảo tính nhất quán[1]) & sau đó ta có thể đánh chỉ số các thành phần liên thông lại nếu cần vì sau khi nối i, j với nhau, số thành phần liên thông c đã giảm đi 1, i.e., $c \leftarrow c - 1$ hay `-c`.

Với truy vấn 2, với 2 đỉnh $i, j \in [n]$, $i \neq j$, ta có thể kiểm tra 2 đỉnh này có cùng nằm trong 1 thành phần liên thông hay không bằng cách so sánh $C(i)$ & $C(j)$ hoặc $\mathrm{cid}(i)$ & $\mathrm{cid}(j)$, nhờ Lem. 1. Việc thực hiện truy vấn này khá hiển nhiên do bản chất của cấu trúc dữ liệu DSU chính là rã sự liên thông thành các thành phần liên khác nhau để tiện quản lý.

Về mặt cài đặt, để giải Bài toán 1, ta sẽ xây dựng 1 cấu trúc dữ liệu có 3 thao tác sau:

1. `make_set(v)` tạo ra 1 tập hợp mới chỉ chứa phần tử v: output of `make_set(v)` $= \{v\}$.
2. `union_sets(a,b)` gộp tập hợp chứa phần tử a & tập hợp chứa phần tử b thành 1, i.e., bước cập nhật thành phần liên thông chung bằng cách lấy hợp của 2 thành phần liên thông tương ứng: $C_{\mathrm{new}}(i) = C_{\mathrm{new}}(j) \coloneqq C(i) \cup C(j)$ đã để cập.

[1] Consistency is 1 of the kings in natural sciences.

3. `find_set(v)` cho biết *đại diện* (representative) của tập hợp có chứa phần tử v (phần tử đại diện cho v không nhất thiết phải là v, e.g., người đỡ đầu cho 1 tập hợp các đứa trẻ trong 1 trại trẻ mồ côi). Đại diện này là sẽ 1 phần tử của tập hợp đó & có thể thay đổi sau mỗi lần gọi thao tác `union_sets` (e.g., nếu chọn phần tử đại diện là đỉnh có giá trị nhỏ nhất thì nếu tìm được đỉnh mới nhỏ hơn phần tử đại diện hiện tại, thì phải cập nhật phần tử đại diện mới này). Ta có thể sử dụng đại diện đó để kiểm tra 2 phần tử có nằm trong cùng 1 tập hợp hay không. a, b nằm trong cùng 1 tập hợp nếu như đại diện của 2 tập chứa chúng là giống nhau & không nằm trong cùng 1 tập hợp nếu ngược lại.

Ta có thể xử lý 3 thao tác này 1 cách hiệu quả với các tập hợp được biểu diễn dưới dạng các cây (tree-based representation), mỗi phần tử là 1 đỉnh & mỗi cây tương ứng với 1 tập hợp. Gốc của mỗi cây sẽ là đại diện của tập hợp đó.

Ban đầu, mỗi phần tử thuộc 1 tập hợp riêng biệt: $\{1\}, \{2\}, \ldots, \{n\}$, nên mỗi đỉnh là 1 cây riêng biệt, cũng là 1 thành phần liên thông riêng biệt. $|E|$ bước tiếp theo, ở bước thứ $i \in [|E|]$, ta gộp 2 tập hợp chứa phần tử $a_i, b_i \in [n]$, $a_i \neq b_i$. Sau $|E|$ bước, ta được c thành phần liên thông $\{C_i\}_{i=1}^{c}$. Với cách cài đặt này, ta sẽ lưu 1 mảng `parent` với `parent[v]` là cha của phần tử v.

1.1.1 Naive implementation of DSU – Cài đặt "ngây thơ" của DSU

Nếu 1 cây được đánh số sao cho nhãn của node cha luôn nhỏ hơn nhãn của node con thì ta có thể dễ dàng định nghĩa khái niệm *gốc* của 1 cây như sau:

Định nghĩa 2. *Cho 1 đồ thị vô hướng $G = (V, E)$, $V = [n]$. Gốc của 1 cây chứa 1 đỉnh $i \in [n]$ được định nghĩa bởi công thức*

$$r(i) \coloneqq \min\{j \in [n]; i, j \text{ are connected}\}.$$

Để tạo 1 tập hợp mới gồm phần tử v bởi `make_set(v)`, ta chỉ cần tạo 1 cây có gốc là v, với `parent[v] = v` (giống như kiểu phim *Predestination* (2014) – tạm dịch: *Tiền Định/Định Mệnh*, mình là cha & là mẹ của chính mình, cũng là con của chính mình luôn). Để gộp 2 tập hợp lần lượt chứa 2 phần tử $a, b \in [n]$ bởi `union_sets(a,b)`, ta sẽ tìm gốc $r(a) \in [n]$ của cây có chứa phần tử a & gốc $r(b) \in [n]$ của cây có chứa phần tử b. Nếu 2 giá trị này giống nhau $r(a) = r(b) = r \in [n]$, thì $r \in C(a) \cap C(b) \Rightarrow C(a) \cap C(b) \neq \emptyset$ nên theo Lem. 1(ii), $C(a) = C(b)$ hay a, b đã liên thông sẵn, nên ta sẽ không làm gì do 2 phần tử a, b đã nằm trong cùng 1 tập hợp chứa gốc chung r. Còn nếu không, i.e., $r(a) \neq r(b)$, ta sẽ đặt gốc cây này là cha của gốc cây còn lại. Cụ thể hơn, nếu muốn khớp với Định nghĩa 2, ta áp dụng truy vấn 1 để thêm 1 cạnh giữa 2 đỉnh $r(a), r(b) \in [n]$: $E \leftarrow E \cup \{\{r(a), r(b)\}\}$. Dễ thấy điều này sẽ gộp 2 cây lại thành 1. Hơn nữa, gốc chung mới vừa được cập nhật của cây chứa cả 2 đỉnh a, b chính là $\min\{r(a), r(b)\}$.

Để tìm ký hiệu của 1 tập hợp có chứa phần tử v bởi `find_set(v)`, ta đơn giản nhảy lên các tổ tiên của đỉnh v cho đến khi ta đến gốc của cây. Thao tác này có thể dễ dàng được cài đặt bằng đệ quy.

```
void make_set(int v) {
    parent[v] = v; // tạo ra cây mới có gốc là đỉnh v
}

int find_set(int v) {
    if (v == parent[v]) return v; // trả về đỉnh v nếu như đỉnh v là gốc của cây
    return find_set(parent[v]); // đệ quy lên cha của đỉnh v
}

void union_set(int a, int b) {
    a = find_set(a); // tìm gốc của cây có chứa đỉnh a
    b = find_set(b); // tìm gốc của cây có chứa đỉnh b
    if (a != b) parent[b] = a; // gộp 2 cây nếu như 2 phần tử ở 2 cây khác nhau
}
```

Vì đây là cách cài đặt ngây thơ, ta có thể dễ dàng tạo ra 1 ví dụ sao cho khi sử dụng cách cài đặt này, cây sẽ trở thành 1 đoạn thẳng gồm n phần tử. Khi đó, độ phức tạp của thao tác `find_set` sẽ là $O(n)$ – hiển nhiên không thể chấp nhận được, nên ta tìm hiểu 2 phương pháp tối ưu thuật toán sau.

1.1.2 1st Optimization: Merge according to size/height – Tối ưu 1: Gộp theo kích cỡ/độ cao

Phương pháp tối ưu này sẽ thay đổi thao tác `union_sets`, i.e., thay đổi cách xét trong 2 cây đang gộp, gốc của cây nào sẽ là cha của gốc của cây còn lại. Có khá nhiều cách để xét điều này, nhưng 2 cách được sử dụng nhiều nhất chính là gộp theo kích cỡ & gộp theo độ cao của cây. Giả sử mỗi cây có 1 giá trị, lần lượt là kích cỡ & độ cao của cây theo 2 cách gộp. Ở cả 2 cách gộp, ta sẽ luôn đặt gốc của cây có giá trị lớn hơn là cha của gốc của cây có giá trị nhỏ hơn.

Thao tác `union_sets` được tối ưu gộp theo kích cỡ:

```
// merge according to size
void make_set_size(int v) {
    parent[v] = v;
    size[v] = 1; // ban đầu tập hợp chứa v có kích cỡ là 1
}
```

```
void union_set_size(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b]) swap(a, b); // đặt biến a là gốc của cây có kích cỡ lớn hơn
        parent[b] = a;
        size[a] += size[b]; // cập nhật kích cỡ của cây mới gộp lại
    }
}
```

Thao tác `union_sets` được tối ưu gộp theo độ cao:

```
// merge according to size
void make_set_height(int v) {
    parent[v] = v;
    rank[v] = 0; // gốc của cây có độ cao là 0
}

void union_set_height(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b]) swap(a, b); // đặt biến a là gốc của cây có độ cao lớn hơn
        parent[b] = a;
        if (rank[a] == rank[b]) ++rank[a]; // nếu 2 cây có cùng 1 độ cao, độ cao của cây mới
            // sau khi gộp sẽ tăng 1
    }
}
```

Chỉ cần sử dụng phương pháp tối ưu này, với 1 trong 2 cách cài đặt, độ phức tạp của thao tác `find_set` sẽ trở thành $O(\log n)$. Tuy nhiên, ta có thể tối ưu hóa hơn nữa khi kết hợp với phương pháp tối ưu thứ 2.

Question 1. *Có cách nào hybrid để trộn giữa 2 cách hợp theo kích cỡ & độ cao không?*

1.1.3 2nd Optimization: Path compression – Tối ưu 2: Nén đường đi

Phương pháp tối ưu nén đường đi nhằm tăng tốc thao tác `find_set`. Giả sử ta gọi hàm `find_set(v)` với 1 đỉnh $v \in [n]$ bất kỳ, ta tìm được p là gốc của cây, đồng thời cũng là giá trị của mọi hàm `find_set(u)` với u là 1 đỉnh nằm trên đường đi từ v đến p.[2] Cách tối ưu ở đây chính là làm cho đường đi đến gốc của các đỉnh u ngắn đi bằng cách gán trực tiếp cha của các đỉnh u này thành p.

```
// path compression
int find_set_path_compression(int v) {
    if (v == parent[v]) return v; // trả về đỉnh v nếu như đỉnh v là gốc của cây
    int p = find_set(parent[v]); // đệ quy lên cha của đỉnh v
    parent[v] = p; // nén đoạn từ v lên gốc của cây
    return p;
}
```

1 cách cài đặt khác của thao tác `find_set` thường được sử dụng nhiều trong Competitive Programming do tính ngắn gọn của nó:

```
// brief find set
int find_set_brief(int v) {
    return v == parent[v] ? v : parent[v] = find_set_brief(parent[v]);
}
```

1.2 Time complexity & its proof – Độ phức tạp thời gian & chứng minh

2 Some Applications of DSU – Vài Ứng Dụng của DSU

3 Miscellaneous

[2] NQBH: Trong bài viết gốc VNOI Wiki/disjoint set union, tác giả viết nhầm thành "với u là 1 đỉnh nằm trên đường đi từ u đến p" sửa lại thành "với u là 1 đỉnh nằm trên đường đi từ v đến p".