**Disjoint Set Union (DSU) - Complete Guide**

**What is DSU?**

**Disjoint Set Union** is a data structure that efficiently handles a collection of **disjoint sets** (sets that don't overlap). It supports two main operations:

* **Find**: Which set does an element belong to?
* **Union**: Merge two sets into one

**Real-World Analogy**

Think of **friend groups** in a school:

* Initially, everyone is in their own group (alone)
* When two people become friends, their groups merge
* We want to quickly check: "Are person A and person B in the same friend group?"

**Basic Example**

Let's say we have 5 people: **{0, 1, 2, 3, 4}**

**Initial State**

Sets: {0}, {1}, {2}, {3}, {4}

Each person is in their own group

**After Union(0,1)**

Sets: {0,1}, {2}, {3}, {4}

Person 0 and 1 are now friends

**After Union(2,3)**

Sets: {0,1}, {2,3}, {4}

Person 2 and 3 are now friends

**After Union(0,2)**

Sets: {0,1,2,3}, {4}

Since 0 is connected to 1, and 2 is connected to 3,

joining 0 and 2 merges all four into one group!

**How DSU Works Internally**

**1. Parent Array Representation**

We represent each set as a **tree** where:

* Each element points to its parent
* The root of the tree represents the entire set

Initial parent array:

Index: 0 1 2 3 4

Parent: 0 1 2 3 4

(Everyone is their own parent = root of their own tree)

**2. Find Operation**

To find which set an element belongs to, we follow parent pointers until we reach the root:

int find(int x) {

if(parent[x] == x) return x; // x is the root

return find(parent[x]); // Keep going up

}

**Example**: After some unions, if parent array is [0, 0, 2, 2, 4]:

* Find(1): parent[1] = 0, parent[0] = 0 → Root is 0
* Find(3): parent[3] = 2, parent[2] = 2 → Root is 2

**3. Union Operation**

To merge two sets, we make the root of one set point to the root of another:

void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if(rootX != rootY) {

parent[rootX] = rootY; // Make rootX point to rootY

}

}

**Step-by-Step Example**

Let's trace through unions with elements {0,1,2,3,4}:

**Initial State**

Parent: [0, 1, 2, 3, 4]

Trees: 0 1 2 3 4

**Union(0,1)**

find(0) = 0, find(1) = 1

parent[0] = 1

Parent: [1, 1, 2, 3, 4]

Trees: 1 1 2 3 4

/

0

**Union(2,3)**

find(2) = 2, find(3) = 3

parent[2] = 3

Parent: [1, 1, 3, 3, 4]

Trees: 1 1 3 3 4

/ /

0 2

**Union(0,2)**

find(0) = find(1) = 1

find(2) = find(3) = 3

parent[1] = 3

Parent: [1, 3, 3, 3, 4]

Trees: 3 3 3 4

/| |

1 2 (itself)

/

0

Final structure: {0,1,2,3}, {4}

**Problem: Inefficient Trees**

Without optimization, trees can become **very tall** (like a linked list), making find operations slow.

Worst case after many unions:

4 → 3 → 2 → 1 → 0

Finding 4 takes 4 steps!

**Optimization 1: Path Compression**

During find operations, make all nodes point directly to the root:

int find(int x) {

if(parent[x] != x) {

parent[x] = find(parent[x]); // Path compression

}

return parent[x];

}

**Before path compression:**

4 → 3 → 2 → 1 → 0

**After find(4) with path compression:**

0

/|\\\

1 2 3 4

All nodes now point directly to root!

**Optimization 2: Union by Rank**

When uniting two sets, attach the smaller tree under the larger tree's root:

class DSU {

vector<int> parent, rank;

public:

DSU(int n) : parent(n), rank(n, 0) {

for(int i = 0; i < n; i++) parent[i] = i;

}

int find(int x) {

if(parent[x] != x) {

parent[x] = find(parent[x]);

}

return parent[x];

}

void unite(int x, int y) {

int px = find(x), py = find(y);

if(px == py) return;

if(rank[px] < rank[py]) {

parent[px] = py;

} else if(rank[px] > rank[py]) {

parent[py] = px;

} else {

parent[py] = px;

rank[px]++;

}

}

};

**Complete Working Example**

Let's solve: "Find number of connected components in a graph"

#include <iostream>

#include <vector>

using namespace std;

class DSU {

private:

vector<int> parent, rank;

int components;

public:

DSU(int n) : parent(n), rank(n, 0), components(n) {

for(int i = 0; i < n; i++) {

parent[i] = i;

}

}

int find(int x) {

if(parent[x] != x) {

parent[x] = find(parent[x]);

}

return parent[x];

}

bool unite(int x, int y) {

int px = find(x), py = find(y);

if(px == py) return false;

if(rank[px] < rank[py]) {

parent[px] = py;

} else if(rank[px] > rank[py]) {

parent[py] = px;

} else {

parent[py] = px;

rank[px]++;

}

components--;

return true;

}

int getComponents() { return components; }

};

int main() {

// Graph with 5 nodes and edges: (0,1), (1,2), (3,4)

DSU dsu(5);

cout << "Initial components: " << dsu.getComponents() << endl; // 5

dsu.unite(0, 1);

cout << "After unite(0,1): " << dsu.getComponents() << endl; // 4

dsu.unite(1, 2);

cout << "After unite(1,2): " << dsu.getComponents() << endl; // 3

dsu.unite(3, 4);

cout << "After unite(3,4): " << dsu.getComponents() << endl; // 2

// Check if 0 and 2 are connected

cout << "Are 0 and 2 connected? " << (dsu.find(0) == dsu.find(2)) << endl; // Yes

// Check if 0 and 3 are connected

cout << "Are 0 and 3 connected? " << (dsu.find(0) == dsu.find(3)) << endl; // No

return 0;

}

**Time Complexity**

With both optimizations:

* **Find**: O(α(n)) where α is inverse Ackermann function
* **Union**: O(α(n))
* **α(n)**: Practically constant for all reasonable n (< 5 for n < 10^600)

**Common Applications**

1. **Kruskal's MST Algorithm**
2. **Cycle Detection in Undirected Graphs**
3. **Connected Components**
4. **Dynamic Connectivity**
5. **Percolation Problems**

**Your Original Problem Connection**

In your code, you were:

1. **Removing edges** to increase components
2. **Counting components** repeatedly

DSU optimization works because:

1. **Adding edges** (union) is DSU's natural operation
2. **Component counting** is maintained automatically
3. **Reverse thinking**: Instead of removing edges ≤ threshold, add edges > threshold

This transforms O(VE) repeated BFS into O(α(V)) DSU operations!