Disjoint Sets Data Structure - Complete Notes

Disjoint Sets (also called Union-Find) track collections of disjoint (non-overlapping) sets with efficient **Find** and **Union** operations.

1. Core Concepts

What are Disjoint Sets?

- Represent **separate components** in an undirected graph (no common vertices between sets).
- **Intersection** of any two disjoint sets = **empty set**.
- Used for **cycle detection** in undirected graphs.

Key Operations (Only 2 Needed)

1. **Find(x)**: Determine which set vertex **x** belongs to (set membership).
2. **Union(S1, S2)**: Merge two sets when connecting vertices with an edge.

---

2. Cycle Detection Algorithm

Core Principle

```
For each edge (u, v):
1. Find set of u → Find(u)
2. Find set of v → Find(v)
3. If Find(u) == Find(v) → CYCLE DETECTED! (don't add edge)
4. Else → Union the two sets
```

Key Insight: Same set membership = path already exists between vertices → adding edge creates cycle.

Step-by-Step Example (8-Vertex Graph)

Detailed Graph Example Walkthrough

Initial State: 8 singleton sets {1},{2},{3},{4},{5},{6},{7},{8}

Edge	Find(u)	Find(v)	Same Set?	Action	New Sets
1-2	{1}	{2}	No	Union	{1,2}, {3}, {4}, {5}, {6}, {7},

Edge	Find(u)	Find(v)	Same Set?	Action	New Sets
					{8}
3-4	{3}	{4}	No	Union	{1,2}, {3,4}, {5}, {6}, {7}, {8}
5-6	{5}	{6}	No	Union	{1,2}, {3,4}, {5,6}, {7}, {8}
7-8	{7}	{8}	No	Union	{1,2}, {3,4}, {5,6}, {7,8}
2-4	{1,2}	{3,4}	No	Union	{1,2,3,4}, {5,6}, {7,8}
2-5	{1,2,3,4}	{5,6}	No	Union	{1,2,3,4,5,6}, {7,8}
1-3	{1,2,3,4,5,6}	{1,2,3,4,5,6}	**Yes**	**CYCLE!**	Don't add
6-8	{1,2,3,4,5,6}	{7,8}	No	Union	{1,2,3,4,5,6,7,8}
5-7	{1,2,3,4,5,6,7,8}	{1,2,3,4,5,6,7,8}	**Yes**	**CYCLE!**	Don't add

3. Graphical Representation (Tree Structure)

Each set has ONE representative (root/parent).

```
Initial: 1→self, 2→self, 3→self, 4→self...

After 1-2:     1(min-2)
               ↓
               2

After 3-4:     3(min-2)
               ↓
               4

Union 2-4:     1(min-4)
             ↙   ↓
            2    3
                 ↓
                 4
```

Union Rule: Attach **smaller tree** to **larger tree** root (by node count).

4. Array Representation (parent[] array)

Single array tracks parent relationships - indices = vertices.

Array Encoding Rules

```
parent[i] = -1     → i is ROOT (set representative)
parent[i] = j      → i's parent is j (j ≠ i)
parent[root] = -N  → root with N nodes in set
```

Complete Array Evolution

```
Initial: [-1,-1,-1,-1,-1,-1,-1,-1]

After 1-2: [-2, 1,-1,-1,-1,-1,-1,-1]  // 1 is root of {1,2}

After 3-4: [-2, 1,-2, 3,-1,-1,-1,-1]  // 3 is root of {3,4}

After 2-4: [-4, 1, 1, 1,-1,-1,-1,-1]  // Union: parent[3] = 1

After 2-5: [-6, 1, 1, 1, 1,-1,-1,-1]  // Union: parent[5] = 1

Find 1-3: Find(1)=1, Find(3)=1 → SAME SET → CYCLE
```

---

5. Time Optimizations

A. Weighted Union (Union by Rank/Size)

```
When Union(A, B):
1. Compare |set A| vs |set B| (from parent[root])
2. Attach SMALLER tree to LARGER tree root
3. Update size of new root: sizeA + sizeB
```

Keeps trees balanced → O(α(n)) time (almost constant).

B. Collapsing Find (Path Compression)

Problem: Deep trees → long Find chains.

```
Before collapse: 6→5→1(root)
Find(6): 6→5→1 (2 hops)
```

```
After collapse: 6→1(root)
Next Find(6): 1 hop
```

During Find(x):

1. Trace path to root
2. **Directly link** all nodes on path to root
3. Return root

---

6. Complete Pseudocode

```
Initialize(n):
    parent = [-1] * n  // All singletons

Find(x):
    if parent[x] < 0:      // x is root
        return x
    else:
        parent[x] = Find(parent[x])  // Path compression
        return parent[x]

Union(u, v):
    rootU = Find(u)
    rootV = Find(v)

    if rootU == rootV:
        return False  // Cycle!

    // Weighted union
    if parent[rootU] > parent[rootV]:  // |U| < |V|
        parent[rootU] = rootV
        parent[rootV] += parent[rootU]  // Update size
    else:
        parent[rootV] = rootU
        parent[rootU] += parent[rootV]
    return True
```

---

7. Key Applications

- **Kruskal's MST Algorithm**: Add minimum edges without cycles

- **Connected Components** in undirected graphs
- **Dynamic Connectivity** queries

8. Time Complexity Summary

Operation	Naive	Optimized (WU + CF)
Find	O(n)	O(α(n)) ≈ O(1)
Union	O(n)	O(α(n)) ≈ O(1)
m operations	O(mn)	O(m α(n))

α(n) = **Inverse Ackermann** → Grows slower than log log log n (effectively constant).