Union-Find (Disjoint Set)

Data Structures

Minseok Jeon
DGIST

November 2, 2025

Table of Contents

Introduction to Union-Find

What is Union-Find?

Union-Find (Disjoint Set) is a data structure that maintains dynamic connectivity across disjoint sets.

Key Concepts:

- **Disjoint Set:** Collection of non-overlapping sets
- **Representative:** Each set has a representative (root) element
- **Parent Array:** Each element points to its parent
- **Forest:** Multiple trees, each representing a set

Core Operations:

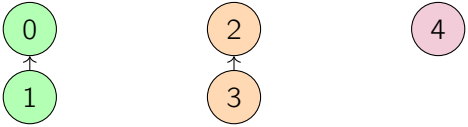
- `find(x)`: Find the representative of x's set
- `union(x, y)`: Merge the sets containing x and y
- `connected(x, y)`: Check if x and y are in the same set

Visual Representation

Initial State (n=5):

Each element is its own set

After union(0,1) and union(2,3):



Three disjoint sets: {0,1}, {2,3}, {4}

Applications

Union-Find has numerous practical applications:

1. **Kruskal's Algorithm:** Finding minimum spanning trees
2. **Network Connectivity:** Checking if nodes can communicate
3. **Percolation:** Modeling fluid flow through materials
4. **Image Segmentation:** Grouping similar pixels
5. **Social Networks:** Finding friend circles
6. **Cycle Detection:** Detecting cycles in undirected graphs
7. **Dynamic Equivalence:** Tracking equivalent elements
8. **Number of Islands:** Counting connected components in a grid

Key Advantage: Nearly O(1) operations with proper optimizations!

Basic Implementation

Naive Implementation

```python
class UnionFind:
    def __init__(self, n):
        # Initially, each element is its own parent
        self.parent = list(range(n))
        self.count = n  # Number of disjoint sets

    def find(self, x):
        """Find root of element x"""
        while self.parent[x] != x:
            x = self.parent[x]
        return x

    def union(self, x, y):
        """Merge sets containing x and y"""
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            self.parent[root_x] = root_y
```

Example Usage

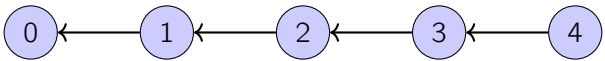
```
# Create Union-Find with 5 elements
uf = UnionFind(5)

# Union operations
uf.union(0, 1)  # Merge {0} and {1}
uf.union(2, 3)  # Merge {2} and {3}
uf.union(0, 2)  # Merge {0,1} and {2,3}

# Check connectivity
print(uf.connected(1, 3))  # True (both in {0,1,2,3})
print(uf.connected(1, 4))  # False (4 is separate)
print(uf.count)            # 2 sets: {0,1,2,3}, {4}
```

Problem: This naive implementation can be slow!

- Worst case: Linear chain of elements
- `find` operation: O(n) time
- `union` operation: O(n) time

Problem: Linear Chains

Worst Case Scenario:



Linear chain: find(4) requires 4 steps

Consequences:

- Sequential unions can create long chains
- Each `find` operation walks entire chain
- Multiple finds: O(n) each time
- Overall complexity: O(n) per operation

Solution: We need optimizations!

Path Compression

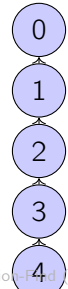
Path Compression Idea

Optimization 1: Path Compression

Key Insight:

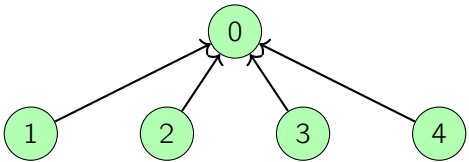
- During `find`, we walk from x to root
- Why not make every node on path point directly to root?
- Future `find` operations become faster!

Before find(4):



Path Compression Effect

After find(4) with path compression:



All nodes now point directly to root!

Benefits:

- Tree becomes flatter
- Future `find` operations: O(1)
- Amortized complexity improves significantly

Path Compression Implementation

Recursive Implementation:

```python
def find(self, x):
    """Find with path compression"""
    if self.parent[x] != x:
        # Recursively find root and compress path
        self.parent[x] = self.find(self.parent[x])
    return self.parent[x]
```

Iterative Implementation:

```python
def find_iterative(self, x):
    """Iterative find with path compression"""
    root = x
    # Find root
    while self.parent[root] != root:
        root = self.parent[root]

    # Compress path: point all nodes to root
```

Path Halving Optimization

Path Halving: Point every other node to its grandparent (one-pass)

```
def find_path_halving(self, x):
    """Point every other node to its grandparent"""
    while self.parent[x] != x:
        self.parent[x] = self.parent[self.parent[x]]
        x = self.parent[x]
    return x
```

Advantages:

- Single pass (no recursion or two passes)
- Simpler and more efficient
- Still achieves excellent amortized complexity
- Popular in competitive programming

Union by Rank

Union by Rank Idea

Optimization 2: Union by Rank

Problem with Naive Union:

- Always attach first tree to second
- Can create unbalanced trees
- Results in long chains

Solution: Union by Rank

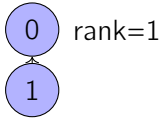
- **Rank:** Upper bound on tree height
- **Rule:** Attach smaller rank tree under larger rank tree
- **Benefit:** Keeps trees balanced
- Trees grow logarithmically

Rank Update Rules:

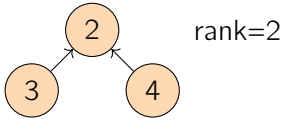
- If ranks differ: Attach smaller under larger, no rank change
- If ranks equal: Attach either way, increase winner's rank by 1

Union by Rank Example

Tree A (rank 1):



Tree B (rank 2):



After union(0, 2): Attach A under B (smaller rank under larger)

Union by Rank Implementation

```python
class UnionFindRank:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n  # Initial rank is 0
        self.count = n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x == root_y:
            return False

        # Attach smaller rank tree under larger rank tree
```

Union by Size

Union by Size

Alternative: Union by Size

Idea:

- Track number of elements in each tree
- Attach smaller tree under larger tree
- Also keeps trees balanced

Comparison with Union by Rank:

- **Rank:** Upper bound on height (less accurate with path compression)
- **Size:** Exact count of elements (always accurate)
- Both achieve $O(\alpha(n))$ complexity with path compression
- Size is more intuitive and provides useful information

Additional Benefits of Size:

- Can query component size: `get_size(x)`
- Track largest component: `max_size`

Union by Size Implementation

```python
class UnionFindSize:
    def __init__(self, n):
        self.parent = list(range(n))
        self.size = [1] * n  # Initial size is 1
        self.count = n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x == root_y:
            return False

        # Attach smaller tree under larger tree
```

Complexity Analysis

The Ackermann Function

Ackermann Function A(m, n): Grows extremely fast

```
def A(m, n):
    if m == 0:
        return n + 1
    if n == 0:
        return A(m - 1, 1)
    return A(m - 1, A(m, n - 1))
```

Growth Rate:

- A(0, n) = n + 1
- A(1, n) = n + 2
- A(2, n) = 2n + 3
- A(3, n) = 2^{n+3} - 3
- A(4, n) = $2^{2^{2^{\cdots}}}$ (n+3 times) - 3

A(4, 2) is already larger than the number of atoms in the universe!

Inverse Ackermann Function

Inverse Ackermann α(n): Grows extremely slowly

α(n) = minimum m such that A(m, m) $\geq$ n

Values of α(n):

- $\alpha(1) = 1$
- $\alpha(3) = 2$
- $\alpha(7) = 3$
- $\alpha(2047) = 4$
- $\alpha(2^{2048} - 1) = 5$

Practical Implication:

- For all practical n (even $n = 10^{80}$), $\alpha(n) \leq 5$
- Effectively constant time!
- For $n = 10^9$, $\alpha(n) \approx 4$

Complexity Comparison

Operation	Naive	Path Comp	Union Rank	Both
Find	O(n)	O(log n)*	O(log n)	O(α(n))*
Union	O(n)	O(log n)*	O(log n)	O(α(n))*
Connected	O(n)	O(log n)*	O(log n)	O(α(n))*
Space	O(n)	O(n)	O(n)	O(n)

* Amortized complexity

Key Insights:

- Path compression alone: O(log n) amortized
- Union by rank alone: O(log n) worst case
- Both together: O(α(n)) amortized $\approx$ O(1)
- Space is always O(n)

Practical Performance

Example: 10^9 **operations on** 10^9 **elements**

Implementation	Complexity	Operations
Naive	O(n)	10^{18}
Path Compression	O(log n)	$\sim 3 \times 10^{10}$
Union by Rank	O(log n)	$\sim 3 \times 10^{10}$
Both Optimized	$O(\alpha(n))$	$\sim 4 \times 10^9$

Speedup with both optimizations:

- vs Naive: $\sim$ 250 million times faster!
- vs Single optimization: $\sim$ 7.5 times faster
- Practically constant time per operation

Applications

Kruskal's Minimum Spanning Tree

Problem: Find minimum spanning tree of weighted graph

```python
def kruskal_mst(n, edges):
    """
    Find MST using Kruskal's algorithm
    n: number of vertices
    edges: list of (weight, u, v)
    """
    # Sort edges by weight
    edges.sort()

    uf = UnionFind(n)
    mst = []
    total_weight = 0

    for weight, u, v in edges:
        # If u and v not connected, add edge
        if uf.union(u, v):
            mst.append((u, v, weight))
            total_weight += weight
```

Network Connectivity

```python
class NetworkConnectivity:
    """Check if network is fully connected"""
    def __init__(self, n):
        self.uf = UnionFind(n)

    def add_connection(self, u, v):
        """Add connection between nodes u and v"""
        self.uf.union(u, v)

    def is_connected(self, u, v):
        """Check if u and v can communicate"""
        return self.uf.connected(u, v)

    def is_fully_connected(self):
        """Check if all nodes are connected"""
        return self.uf.count == 1

    def count_components(self):
        """Count number of separate networks"""
```

Number of Islands

Problem: Count islands in 2D grid ('1' = land, '0' = water)

```python
def num_islands(grid):
    """Count islands in 2D grid"""
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    uf = UnionFind(rows * cols)

    def get_id(r, c):
        return r * cols + c

    # Union adjacent land cells
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '1':
                # Check right neighbor
                if c + 1 < cols and grid[r][c + 1] == '1':
                    uf.union(get_id(r, c), get_id(r, c + 1))
```

Cycle Detection

Problem: Detect cycle in undirected graph

```python
def has_cycle(n, edges):
    """Check if undirected graph has cycle"""
    uf = UnionFind(n)

    for u, v in edges:
        # If u and v already connected, adding edge creates cycle
        if uf.connected(u, v):
            return True
        uf.union(u, v)

    return False

# Example: Triangle graph
edges = [(0, 1), (1, 2), (2, 0)]
print(has_cycle(3, edges))  # True
```

Why it works:

Friend Circles

Problem: Count number of friend circles (transitive friendships)

```python
def find_circle_num(is_connected):
    """
    Count number of friend circles
    is_connected[i][j] = 1 if person i and j are friends
    """
    n = len(is_connected)
    uf = UnionFind(n)

    for i in range(n):
        for j in range(i + 1, n):
            if is_connected[i][j] == 1:
                uf.union(i, j)

    return uf.count

# Example
is_connected = [
    [1, 1, 0],  # Person 0 friends with 0,1
```

Redundant Connection

Problem: Find edge that creates cycle in tree

```python
def find_redundant_connection(edges):
    """Find edge that creates cycle in tree"""
    n = len(edges)
    uf = UnionFind(n + 1)

    for u, v in edges:
        if not uf.union(u, v):
            # Union failed: u and v already connected
            # This edge creates a cycle
            return [u, v]

    return []

# Example: Tree edges with one extra
edges = [[1,2], [1,3], [2,3]]
print(find_redundant_connection(edges))  # [2,3]
```

Advanced Patterns

Complete Optimized Implementation

```python
class UnionFind:
    """Union-Find with all optimizations"""
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.count = n

    def find(self, x):
        """Find with path compression"""
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        """Union by rank"""
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x == root_y:
```

Union-Find with Size Tracking

```python
class UnionFindWithSize:
    """Track size of each component"""
    def __init__(self, n):
        self.parent = list(range(n))
        self.size = [1] * n
        self.count = n
        self.max_size = 1  # Track largest component

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x == root_y:
            return False

        # Always attach smaller to larger
        if self.size[root_x] < self.size[root_y]:
            self.parent[root_x] = root_y
            self.size[root_y] += self.size[root_x]
```

Union-Find with Custom Data

```python
class UnionFindCustom:
    """Union-Find with custom merge function"""
    def __init__(self, n, merge_fn=None):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.data = [None] * n  # Store custom data
        self.merge_fn = merge_fn or (lambda a, b: a)
        self.count = n

    def set_data(self, x, data):
        """Set data for element x"""
        self.data[x] = data

    def get_data(self, x):
        """Get data for component containing x"""
        return self.data[self.find(x)]

    def union(self, x, y):
        root_x = self.find(x)
```

Example: Component Sum Tracking

```python
# Create Union-Find that tracks sum of each component
uf = UnionFindCustom(5, merge_fn=lambda a, b: a + b)

# Initialize with values 0, 1, 2, 3, 4
for i in range(5):
    uf.set_data(i, i)

# Merge components
uf.union(0, 1)  # Component sum: 0+1 = 1
uf.union(2, 3)  # Component sum: 2+3 = 5

print(uf.get_data(0))  # 1 (sum of component {0,1})
print(uf.get_data(2))  # 5 (sum of component {2,3})
print(uf.get_data(4))  # 4 (sum of component {4})

uf.union(0, 2)  # Merge: 1+5 = 6
print(uf.get_data(0))  # 6 (sum of component {0,1,2,3})
```

Summary

Summary: Union-Find Structure

Core Data Structure:

- Parent array representing forest of trees
- Each tree is a disjoint set
- Root of tree is representative of set

Core Operations:

- `find(x)`: Find representative of x's set
- `union(x, y)`: Merge sets containing x and y
- `connected(x, y)`: Check if x and y in same set

Key Optimizations:

- **Path Compression:** Flatten trees during find
- **Union by Rank/Size:** Keep trees balanced
- **Together:** Nearly O(1) operations!

Summary: Complexity

Configuration	Time per Op	Space
Naive	O(n)	O(n)
Path Compression only	O(log n)*	O(n)
Union by Rank only	O(log n)	O(n)
Both Optimizations	$O(\alpha(n))^* \approx O(1)$	O(n)

* Amortized complexity

Practical Performance:

- $\alpha(n) \leq 5$ for all practical n
- Essentially constant time operations
- Linear space overhead
- Extremely efficient in practice

Summary: Applications

Common Applications:

1. Kruskal's MST algorithm
2. Network connectivity problems
3. Percolation theory
4. Image processing / segmentation
5. Dynamic graph connectivity
6. Cycle detection in graphs
7. Finding connected components
8. Social network analysis

When to Use Union-Find:

- Need to group elements into disjoint sets
- Frequent connectivity queries
- Dynamic merging of groups
- Offline processing of relationships

Key Takeaways

1. Union-Find efficiently maintains disjoint sets
2. Simple parent array representation
3. Two key optimizations:
   - Path compression (flatten on find)
   - Union by rank/size (balance trees)
4. Together achieve $O(\alpha(n)) \approx O(1)$ per operation
5. Extremely practical and widely applicable
6. Easy to implement, hard to beat performance
7. Essential for graph algorithms (Kruskal's)
8. Versatile for connectivity problems

Remember: Always use both optimizations in practice!

Practice Problems

LeetCode Problems:

- 200. Number of Islands
- 547. Number of Provinces (Friend Circles)
- 684. Redundant Connection
- 685. Redundant Connection II
- 721. Accounts Merge
- 990. Satisfiability of Equality Equations
- 1319. Number of Operations to Make Network Connected
- 1579. Remove Max Number of Edges to Keep Graph Traversable

Advanced:

- Minimum Spanning Tree problems
- Dynamic connectivity with deletions
- Persistent Union-Find

Further Learning

Topics to Explore:

- Persistent Union-Find (immutable/versioned)
- Union-Find with rollback
- Weighted Union-Find (for shortest paths)
- Link-Cut Trees (more powerful, more complex)
- Applications in competitive programming

Resources:

- "Introduction to Algorithms" (CLRS) - Chapter 21
- "Algorithms" by Sedgewick & Wayne - Chapter 1.5
- Tarjan's original papers on Union-Find analysis
- Competitive programming books (Halim, etc.)

Thank You!

Questions?

Union-Find: Simple, Elegant, Efficient