Disjoint Set

- A bunch of elements
- Dynamic set
- New elements added to set
  - Each can be only one set = disjoint sets
- Identify a set by an element in the set: representative

- Needs to be same element while set is not modified
- Doesn't matter which element it is
Analysis

Assume \( n \) Make-set operations occur first.

- Can be at most \( n - 1 \) unions.
- Each union reduces number of sets by 1.
- \( mn \) number of Make-set, Union, and Find-set operations.
- \( n \) number of Make-set operations.
- \( n \) number of Make-set operations.

Analysis

Operations

- No delete
- No other operations
- Returns a pointer to the representative of the set containing \( x \).
- Assume set are disjoint beforehand.
- Unions the sets represented by \( x \) and \( y \).
- \( x \) cannot already be in some other set.
- Creates a new set whose only member is \( x \).
- Make-set(\( x \)).
Code

```c
if FIND-SET(u) != FIND-SET(v)
for each edge (u', v') ∈ E
MAKE-SET(u')

for each vertex v ∈ V
CONNECTED-COMPONENTS(G)
```

Application of Disjoint Sets

• Connected components in a Graph

![Connected components in a Graph](image)

Collection of disjoint sets

(a) A graph with four connected components: f(a; b; c; d)g, f(e; f; g)g, f(h; i)g, and f(j)g.

(b) The collection of disjoint sets after processing each edge.
Overview

- Disjoint Forests

Disjoint-Set Operations

- Linked-List Representation

Proof

Question 21.1-2

Show that after all edges are processed by Connected-Components, two vertices are in the same connected component if and only if they are in the same set.

Proof
Time Analysis

- Make-Set(x) and Find-Set(x) - O(1) time
- Union
  - O(1) worst case
  - But must change children of second list to point to first list
  - Union
    - O(1)

Make-Set(x) and Find-Set(x)

Linked-List Representation

(a)

(b)
The result of Union(x; y), which appends the linked list containing y to the linked list containing x. The representative of the resulting set is f. The set object for y's list, S2, is destroyed.

Figure 21.2 (a) Linked-list representation of two sets. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers head and tail to the first and last objects, respectively.
Running time is same as number of times an element's pointer is changed to point to another set. Each time it is changed, put into a set at least twice as large. So most changes are at least \( \lg n \).

**Proof**

Theorem 21.1

Using linked-list and weighted-union heuristic, a sequence of \( m \) Make-set, \( u \) Union, and \( w \) Find-set, \( u \) of which are Make-set, will take \( O(u \lg u + u \lg w) \) time.

- What is running time of non-unions?
- What is maximum number of unions?
- Just need to prove that \( u - 1 \) unions take \( O(u \lg u) \) time.

- Make-set, \( u \) Union, and \( w \) Find-set, \( u \) of which are Make-set, \( w \) Find-set, \( u \) of which are Make-set, \( w \) Find-set, \( u \) of which are Make-set.

**Weighted-Union Heuristic**

- Improvement:
  - Check which list is shorter: append it onto longer list.
  - Worst case since appending a longer list onto a shorter list.
Faster Implementation: Disjoint-set Forests

- Use rooted trees (Chapter 11)
- Each node contains a member, each tree represents a set
- Just use parent pointers
- Forest for the sets

Overview

- Disjoint-set Operations
- Linked-list Representation
- Disjoint Forests
Heuristics

- Path Compression
  - During find-set traversal to root of tree, make each node point to root
  - Don't update rank: rank is an upper bound anyway, and no way to do it

Heuristics to Improve Running Time

- Union by rank
  - Similar to weighted-union heuristic for linked lists

- Make root of tree with fewer nodes the child node
  - How can we keep track of number of nodes without adding extra time?
  - Rank: upper bound on height of node (max number of edges to get to a leaf)
  - When unioning, just root's rank might need to be updated

Heuristics to Improve Running Time
The FIND-SET procedure with path compression is quite simple:

1. \( \text{FIND-SET}(x) \)
2. \( d' = \text{FIND-SET}(x) \)
3. \( d' \neq x \) \( \Rightarrow \)
4. \( \text{FIND-SET}(x) \)
5. \( \text{return} d' \)
6. \( \text{MARK} \)
7. \( \text{MARK} \)

Where are two heuristics?
+ Tail-end recursion?
- How does Find-Set work?
- When is rank changed?
+ When might rank just be upper bound?
- When are two heuristics?