DISJOINT SET DATA STRUCTURES

CS340

Disjoint-set data structures

- Also known as "**union find**."
- **Disjoint** = "no elements in common"
- Maintain collection $S = \{S_1, \ldots, S_k\}$ of disjoint dynamic (changing over time) sets.
- Each set is identified by a **representative**, which is some member of the set.
- Doesn't matter which member is the representative, as long as if we ask for the representative twice without modifying the set, we get the same answer both times.

Operations supported

- MAKE-SET(x)
  - Makes a new set whose only member (and representative) is x.
- UNION(x,y)
  - Unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. The representative is any member of $S_x \cup S_y$.
- FIND-SET(x)
  - Returns a pointer to the representative of the (unique) set containing x.

Running Times

- Analyzed in terms of:
  - n, the number of MAKE-SET operations,
  - m, the total number of MAKE-SET, UNION, and FIND-SET operations.
- Each UNION(x) reduces the number of sets by 1.
  - max number of UNION operations is n-1

Amortized Time Complexity

- Take the time complexity for the complete set of operations, and divide by the number of operations.
- For a disjoint set problem, total time to do 2n-1 (n Make-Sets and n-1 Unions) operations = $\Theta(n^2)$
- Amortized time per operation = $\Theta(n)$

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n-1$

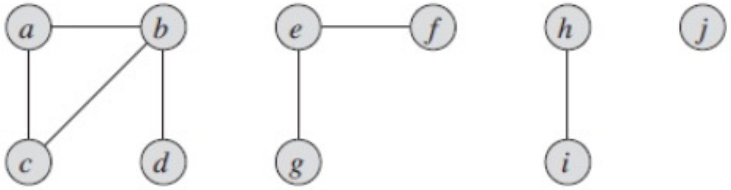
Determining Connected Components

- Why not use DFS?

```
CONNECTED-COMPONENTS(G)
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)
```

```
SAME-COMPONENT(u, v)
if FIND-SET(u) == FIND-SET(v)
    return TRUE
else return FALSE
```

Connected Components



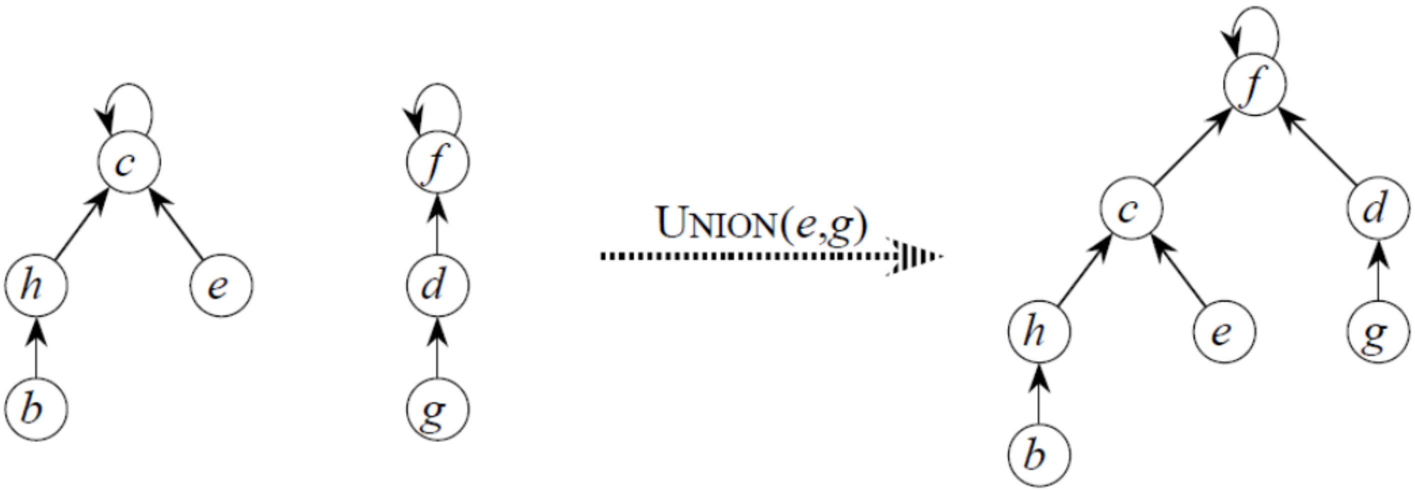
(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(b)

Disjoint Set Forest

- Create a forest of trees.
  - 1 tree per set. Root is representative.
  - Each node points only to its parent.



Disjoint set forest

- MAKE-SET
  - make a single-node tree.
- UNION
  - make one root a child of the other.
- FIND-SET
  - follow pointers to the root.

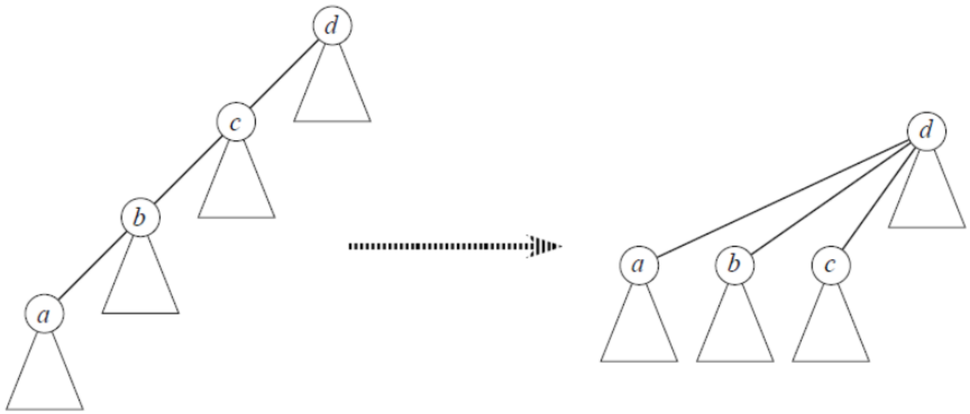
- Sounds good, but if our trees end up looking like linked lists, we haven't gained anything.

Forest heuristics

- Union by rank
  - Make the root with the smaller rank into a child of the root with the larger rank.
  - Rank is an upper bound on the height of a node.
  - Size (number of nodes) of the trees is irrelevant.

Forest Heuristics

- Path compression
  - Find path = nodes visited during FIND-SET on the trip to the root.
  - Make all nodes on the find path direct children of root.
  - We want the tree to be as short and bushy as possible.



Each node has two attributes, *p* (parent) and *rank*.

Forest implementation with heuristics

```
MAKE-SET(x)
    x.p = x
    x.rank = 0

UNION(x, y)
    LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
    if x.rank > y.rank
        y.p = x
    else x.p = y
        if x.rank == y.rank
            y.rank = y.rank + 1
```

```
FIND-SET(x)
    if x ≠ x.p
        x.p = FIND-SET(x.p)
    return x.p
```

notice how cleverly x.p=FIND-SET(x.p) accomplishes path compression!

FIND-SET makes a pass up to find the root, and a pass down as recursion unwinds to update each node on find path to point directly to root.

Running times

- If use both **union by rank** and **path compression**, $O(m\,\alpha(n))$.
- α is a very-slow growing function
- Running time is just barely superlinear