

Data Structures for Disjoint Sets

Adnan Aziz

UT Austin

Based on CLRS, Ch 21.

In several applications, need to group n distinct elements into a collection of disjoint sets.

- Example — electrically equivalent nets on a PCB

Operations:

- return which set an element belongs to
- form the union of sets corresponding to two elements

Formally: disjoint set data structure maintains a collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ of disjoint dynamic sets

- identify each set by a “representative” member
 - usually don’t care which one is representative; just want it to remain the same if no operations are performed on the set

Let x, y be elements; want to support the following operations:

1. MAKE-SET(x) — return a new set whose only member is x
 - very important: x better not be present in any other current set — our assumption is that we are working with disjoint sets
2. FIND-SET(x) — returns a pointer to representative of set containing x
3. UNION(x, y) — unites sets containing x and y

Want **efficient** algorithms

- will measure runtimes of operations in terms of
 - n — the number of make-set operations, and

- m — the total number of make-set, union, and find-set operations

Trivial implementation — linked lists

- each set corresponds to a linked list
 - each node in list has element, next pointer, and pointer to head of list
 - first object in each list is the set's representative
- perform $\text{UNION}(x, y)$ by appending x 's list onto the end of y 's list
 - need to update the representative field for **each** node in x 's list

Example — Figure 21.2

Easy to see make-set and find-set take $O(1)$ time.

Fact — can have a situation where m operations take $\Theta(m^2)$ time

- $\text{MAKE-SET}(x_1), \text{MAKE-SET}(x_2), \dots, \text{MAKE-SET}(x_n), \text{UNION}(x_1, x_2), \text{UNION}(x_2, x_3), \dots, \text{UNION}(x_{n-1}, x_n)$
 - keep having to update lots of representatives!

So the average cost of an operation is $\Theta(m)$.

The problem above is that we are appending a long list onto a short list

- idea — keep track of length of list (easy, fast)
 - always append smaller list onto longer list (break ties arbitrarily)
 - * called the “weighted-union” heuristic

At first glance, this isn't very good — a single union can take $\Omega(m)$ time

- when both sets have m members

However if you start from scratch (i.e., no sets), and perform m make-set, union, and find operations, can prove that runtime is $O(m + n \cdot \log n)$.

An even better approach: represent sets using rooted trees.

- each node contains one element, each tree represents one set as in Figure 21.4(a)
- root of tree contains representative, and is own parent

1. MAKE-SET — create tree with one node
2. FIND-SET — chase pointers to parent (path to parent called the “find path”)
3. UNION — set root of one tree to point to root of other

Straightforward implementation of this idea: $n - 1$ unions can lead to a tree consisting of a linear chain of n nodes

- bad for find operation

Can improve performance dramatically by using the following two ideas:

1. make root of tree with fewer nodes point to root of tree with more nodes (need book-keeping; actually use “rank” estimates — see book for details)
2. update nodes on find path during find (Figure 21.5)

Can prove that performing a sequence of m operations (of which n are make-sets) takes time $O(m \cdot \alpha(m, n))$, where $\alpha(m, n)$ is an incredibly slowly growing function (less than or equal to 6 for any values of n, m that you can think of).