Disjoint Sets (Ch. 21)

A disjoint-set data structure
- maintains a collection of disjoint subsets \( C = s_1, s_2, \ldots, s_m \)
  where each \( s_i \) is identified by a distinguished element.
- set of underlying elements \( U = \{1, 2, \ldots, n\} \)

Operations on \( C \):
- \textbf{make-set}(x): \( x \in U \), creates set \( \{x\} \)
- \textbf{union}(x, y): \( x, y \in U \) and are ids of their resp. sets, \( s_x \) and \( s_y \), replaces sets \( s_x \) and \( s_y \) with a set that is \( s_x \cup s_y \)
  and returns the id of the new set.
- \textbf{find}(x): \( x \in U \), returns the id of the set containing \( x \).

Data Structures for Disjoint Sets

Comment 1: The make-set operation is normally used during the initialization of a particular algorithm.

Comment 2: We assume there is a pointer to each \( x \in U \) (so we never have to look for a particular element). Thus the problems we’re trying to solve are how to manage the sets (unions) and how to find the id of the set containing a particular element (finds).

Finding Connected Components

![Diagram of connected components](image)

After step 1, we have the 7 sets:
\( \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\} \)

After step 2, we have 2 sets:
\( \{a, b, d, f\}, \{e, c, g\} \)

which represent the two connected components of \( G \)

Linked List Representation of Sets

Claim: Amortized (average) worst-case cost per operation is \( O(n) \)

Example worst-case sequence of \( 2n - 1 \) operations:

\( S = \text{M-S}(1), \text{M-S}(2), \ldots, \text{M-S}(n), U(1, 2), U(2, 3), U(3, 4), \ldots, U(n-1, n) \)

- total cost for \( n \) make-sets = \( n \)
- total cost for \( n-1 \) unions = \( 1 + 2 + \ldots + n-1 = O(n^2) \)
- average cost per operation = \( O(n^2)/2n = O(n) \)

Rooted Tree Representation of Sets

Idea: Organize elements of each set as a tree with id = root element, and a pointer from every child to its parent (recall we have pointers to each element).

![Diagram of rooted tree](image)

\textbf{find}(x):
- start at \( x \) (using pointer provided) and follow pointers up to the root.
- return id of root.

w-c running time is \( O(n) \), \( n = |U| \)

\textbf{union}(x, y):
- \( x \) and \( y \) are ids (roots of trees).
- make \( x \) a child of \( y \) and return \( y \)

Running time is \( O(1) \)

Weighted Union Implementation (Union by Rank)

Idea: Add weight (rank) field to each node holding the number of nodes in subtree rooted at this node (only care about weight field of roots).

\textbf{find}(x):
- as before with simple implementation, running time \( O(\log n) \)

\textbf{union}(x, y):
- \( x \) and \( y \) are ids (roots of trees).
- make node \( x \) or \( y \) with smaller weight the child of the other

Running time is \( O(1) \)
Weighted Union

**Theorem**: Any k-node tree created by k-1 weighted unions has height \( \log k \) (assume we start with singleton sets). That is, trees stay “short”.

**Proof**: By induction on k.

- **Basis**: \( k = 1 \), height = 0 = \( \log 1 \)
- **Inductive Hypothesis**: Assume true for all \( i < k \).
- **Inductive Step**: Show true for \( k \). Suppose the last operation performed was union(x,y) and that \( m = \text{wt}(x) \leq \text{wt}(y) = k - m \), so that \( m \leq k/2 \).

Show \( h = \max(h_x + 1, h_y) \leq \log k \)
- \( h_x + 1 = \log m + 1 \leq \log(k/2) + 1 = \log k + 1 + 1 = \log k \)
- \( h_y \leq \log(k - m) \leq \log k \)

Path Compression Implementation

Idea: extend the idea of weighted union (i.e., unions still weighted), but on a find(x) operation, make every node on the path from x to the root (the node with the set id) a child of the root.

So find(x) still has worst-case time of \( O(\log n) \), but subsequent finds for nodes that used to be ancestors of x (or subsequent finds for x itself) will now be very fast.

Path Compression

**Theorem**: Let S be any sequence of \( O(n) \) unions and finds. Then the worst-case time to do S with weighted union and path compression is \( O(n \log^* n) \)

\( \log^* n \) is “almost” constant

\( \log^* n \) is the number of times we have to take the log of a number to reach 1:
- \( \log^* 2 = 1 \)
- \( \log^* 3 \) and \( \log^* 4 = 2 \)
- \( \log^* 16 = 3 \)
- \( \log^* 65536 = 4 \)

Exercise: Union/Find Data Structure

for \( i = 1 \) to 16
- `make-set(i)`
for \( i = 1 \) to 15
- `union(find(i), find(i+1))`
for \( i = 1 \) to 16
- `find(i)`

1. Draw the final data structure that results from applying the above sequence of operations without weighted unions or path compression.
2. Draw the final data structure that results from applying the above sequence of operations with weighted unions but not path compression.
3. Draw the final data structure that results from applying the above sequence of operations using both weighted unions and path compression.