Design and Analysis of Algorithms February 20, 2015
Massachusetts Institute of Technology 6.046J/18.410J
Profs. Erik Demaine, Srini Devadas and Nancy Lynch Recitation 3

Union-Find and Amortization

1 Introduction

A *union-find data structure*, also known as a *disjoint-set data structure*, is a data structure that can keep track of a collection of pairwise disjoint sets $\mathcal{S} = \{S_1, S_2, \ldots, S_r\}$ containing a total of n elements. Each set S_i has a single, arbitrarily chosen element that can serve as a representative for the entire set, denoted as $rep[S_i]$.

Specifically, we wish to support the following operations:

- MAKE-SET(x), which adds a new set $\{x\}$ to $\mathcal{S}$ with $rep[\{x\}] = x$.
- FIND-SET(x), which determines which set $S_x \in \mathcal{S}$ contains x and returns $rep[S_x]$.
- UNION(x, y), which replaces S_x and S_y with $S_x \cup S_y$ in $\mathcal{S}$ for any x, y in distinct sets S_x, S_y.

Furthermore, we want to make these as efficient as possible. We will go through the process of creating a data structure that, in an amortized sense, performs spectacularly.

1.1 Motivation

Union-find data structure is used in many different algorithms. A natural use case of union find is keeping track of connected component in an undirected graph where nodes and edges could be added dynamically. We start with the initial connected components as S_i's. As we add a node v along with its edges $E = \{(v, v_j)\}$, we either

1. Call MAKE-SET if $E = \emptyset$ to make a new component.
2. Call FIND-SET(v_j) to find the component node v will be connected to, and call UNION to connect all those components connected through v.

Another use case of this data structure is in Kruskal's algorithm which you will see in future lectures.

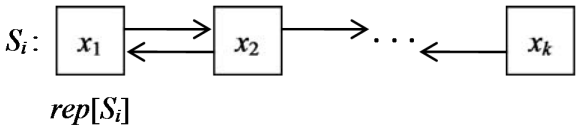

Figure 1: A simple doubly linked list.

2 Starting Out

2.1 Linked List Solution

A naïve way to solve this problem is to represent each set with a doubly linked list like so:

We may also have a data structure, such as a hash table mapping elements to pointers, that allows us to access each linked list node in constant time.

We define $rep[S_i]$ to be the element at the head of the list representing S_i.

Here are the algorithms for each operation:

- MAKE-SET(x) initializes x as a lone node. This takes $\Theta(1)$ time in the worst case.
- FIND-SET(x) walks left in the list containing x until it reaches the front of the list. This takes $\Theta(n)$ time in the worst case.
- UNION(x, y) walks to the tail of S_x and to the head of S_y and concatenates the two lists together. This takes $\Theta(n)$ time in the worst case.

2.2 Augmenting the Linked List

We can improve the behavior of our linked list solution by augmenting the linked lists such that each node has a pointer to its representative and that we also keep track of the tail of the list and the number of elements in the list at any time, which we will call its *weight*.

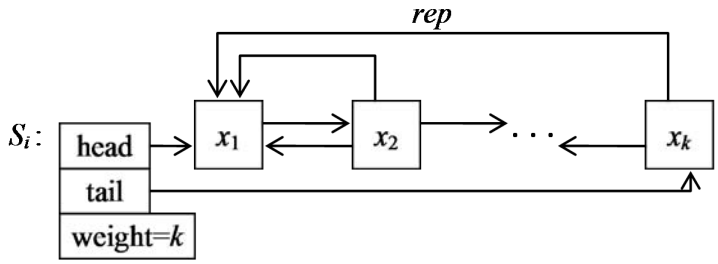

Figure 2: An augmented linked list.

Now, FIND-SET(x) can run in $\Theta(1)$ time.

We also change the behavior of UNION(x, y) to concatenate the lists containing x and y, updating the rep pointers for all the elements in y.

However, this could still require $\Theta(n)$ time in the worst case! Imagine if we called MAKE-SET for every integer between 1 and n and then called UNION$(n-1, n)$, UNION$(n-2, n-1)$, ..., UNION$(1, 2)$, where at the ith union we modify a list of length i. The total cost of all the calls to UNION is $1 + 2 + \ldots + (n-1) = \Theta(n^2)$.

3 First Improvement: Smaller into Larger

(This part is updated and covered in Quiz 1 review session.)

However, we can solve this by forcing UNION to merge the smaller list into the larger list. If we do this, the above scenario will only modify a list of length 1 every time, resulting in $\Theta(n)$ running time for these $n-1$ UNIONs.

Claim. With the first improvement, the amortized running time of n calls to MAKE-SET followed by n calls to UNION is $O(n \lg n)$.

Proof. We first use the aggregate method. Suppose the cost of moving a rep pointer is 1. Monitor some element x and the set S_x that contains it. After MAKE-SET(x), we have $weight[S_x] = 1$. When we call UNION(x, y), one of the following will happen:

- If $weight[S_x] > weight[S_y]$, then $rep[x]$ stays unchanged, so we need not pay anything on x's behalf, and $weight[S_x]$ will only increase.
- If $weight[S_x] \leq weight[S_y]$, we pay 1 to update $rep[x]$, and $weight[S_x]$ at least doubles.

S_x can double at most $\lg n$ times, so we update $rep[x]$ at most $\lg n$ times. Therefore across all the n elements, we get an amortized running time of $O(n \lg n)$. □

Proof. The proof is very similar using the accounting method or the charging method. When we call UNION(x, y), suppose $|S_x| < |S_y|$ without loss of generality. We will charge every element $i \in S_x$ because their pointers $rep[i]$'s are updated. Similarly, each element i can be charged at most $\lg n$ times, because $|S_i|$ at least doubles.

If we use the accounting method, each MAKE-SET(x) stores $\lg n$ coins into the bank for use in future UNION. □

Proof. Now we would like use the potential method. The potential method is usually the "inverse" of the accounting method. The potential function can be viewed as the balance in the bank account. In this case, we define the potential function to be

$$\Phi = \sum_i \lg n - \lg |S_i|$$

where the sum is taken over every element i in the structure (n is an upper bound on the total number of elements).

When we call MAKE-SET(x), $|S_x| = 1$, so the amortized cost is $O(1) + \Delta\Phi = O(\lg n)$. This means each MAKE-SET(x) stores $\lg n$ coins into the bank.

When we call UNION(x, y), again suppose $|S_x| < |S_y|$. The actual cost is $|S_x|$. $\Delta\Phi < -|S_x|$, because every element in $i \in S_x$ now has $|S_i|$ doubled. This means we withdraw $|S_x|$ coins from the bank to pay for this UNION operation. So the amortized cost of UNION is $|S_x| + \Delta\Phi < 0$. □

4 Forest of Trees Representation

One interesting observation we can make is that we don't really care about the links between nodes in the linked list; we only care about the rep pointers. Essentially, a list can be represented as a forest of trees, where $rep[x]$ will be the root of the tree that contains x:

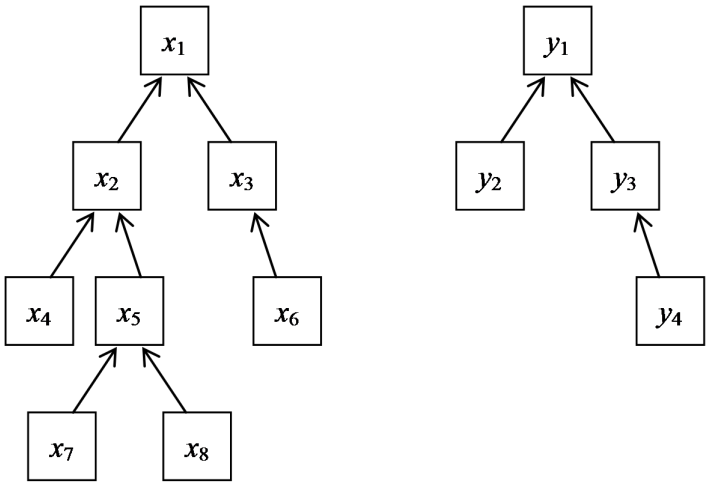

Figure 3: A forest of trees representation of a disjoint set data structure.

The algorithms will now be as follows:

- MAKE-SET(x) initializes x as a lone node. This takes $\Theta(1)$ time in the worst case.
- FIND-SET(x) climbs the tree containing x to the root. This takes $\Theta(height)$ time.
- UNION(x, y) climbs to the roots of the trees containing x and y and merges sets the parent of $rep[y]$ to $rep[x]$. This takes $\Theta(height)$ time.

4.1 Adapting the First Improvement

Our first trick can be modified to fit the forest of trees representation by merging the tree with the smaller height into the tree with the bigger height. One can then show that the height of a tree will still be $O(\lg n)$.

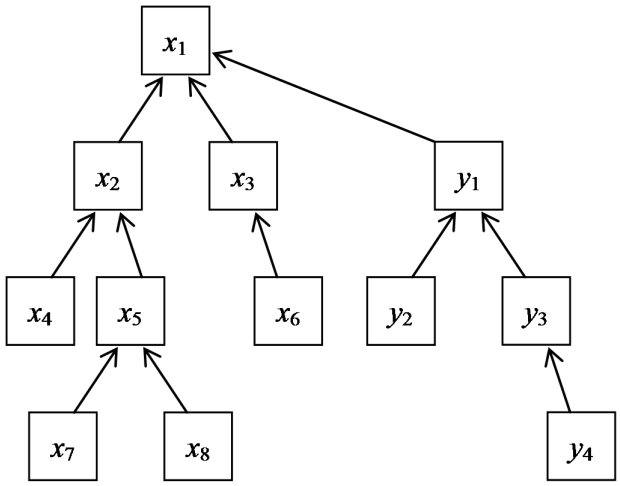

Figure 4: The data structure after calling UNION(x_1, y_1).

5 Second Improvement: Path Compression

Let's now improve FIND-SET. When we climb the tree, we learn the representatives of every intermediate node we pass. Therefore we should redirect the rep pointers so a future call to FIND-SET won't have to do the same computations multiple times.

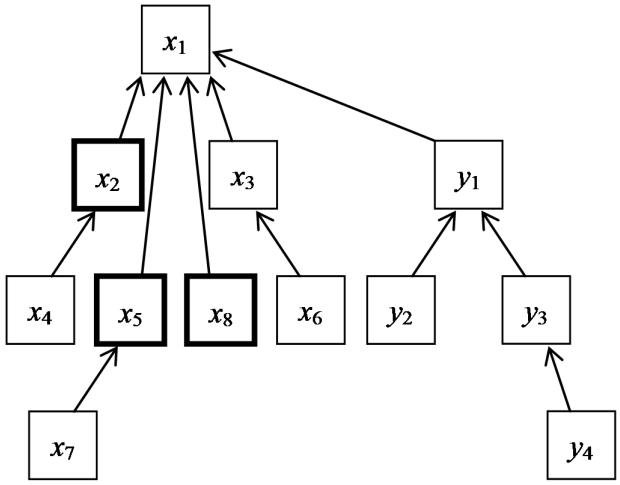

Figure 5: The data structure after calling FIND-SET(x_8).

Claim. Let n be the total number of elements we're keeping track of. With the second improvement alone, the amortized running time of m operations (including FIND-SET) is $O(m \lg n)$.

Proof. Amortization by potential function. Let us define $weight[x_i]$ to be the number of elements in the subtree rooted at x_i.

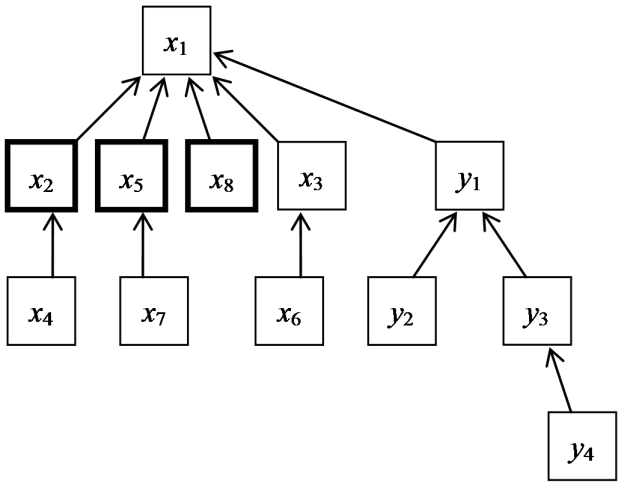

Figure 6: The same tree as in Figure 5, but redrawn. Notice that the x_i's are much more compact than Figure 4.

Let $\Phi(x_1, \ldots, x_n) = \sum_i \lg weight[x_i]$. If UNION$(x_i, x_j)$ attaches x_j's subtree's root as a child of x_i's subtree's root, it only increases the weight of the root of S_{x_i}. The increase is at most $\lg n$, because there are only n elements at most. The weights of all the other elements stay unchanged.

Now consider each step from child c to an ancestor p made by FIND-SET(x_i) moves c's subtree out of p's subtree. If, at any particular step, $weight[c] \geq \frac{1}{2} weight[p]$, then the potential decreases by at least 1, which pays for the move. Furthermore, there can be at most $\lg n$ steps for which $weight[c] < \frac{1}{2} weight[p]$, since each one decreases the size of the tree we're looking in by more than half. □

6 Why Don't We Do Both?

If we do both improvements to the tree representation, we get spectacular behavior where the amortized cost of each operation is almost constant!

6.1 CLRS-Ackermann Function

We define a function, $A_k(j)$, with a similar structure to the well-known Ackermann function, as follows:

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{j+1}(j) & \text{if } k \geq 1 \end{cases}$$

Because you're repeatedly iterating A_{k-1} many times when $k \geq 1$, this function explodes very quickly:

$$\begin{aligned} A_0(1) &= 2 \\ A_1(1) &= 3 \\ A_2(1) &= 7 \\ A_3(1) &= 2047 \\ A_4(1) &> \left.2^{2^{\cdot^{\cdot^{\cdot^{2^{2047}}}}}}\right\} 2048 \end{aligned}$$

Now consider its "inverse" as follows:

$$\alpha(n) = \min\{k : A_k(1) \geq n\}$$

While not technically constant, this value is pretty darn close, even for very large values of n. For practical purposes, you can easily assume that $\alpha(n) \leq 4$.

6.2 How Spectacular?

Theorem. With both improvements improvement, the amortized running time of m operations is $O(m\alpha(n))$.

The proof is very long and very tricky. If you're interested, check out Section 21.4 in CLRS.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015