Lecture 16

Disjoint-Set Data Structures

Supplemental reading in CLRS: Chapter 21 (§21.4 is optional)

When implementing Kruskal's algorithm in Lecture 4, we built up a minimum spanning tree T by adding in one edge at a time. Along the way, we needed to keep track of the connected components of T; this was achieved using a disjoint-set data structure. In this lecture we explore disjoint-set data structures in more detail.

Recall from Lecture 4 that a **disjoint-set data structure** is a data structure representing a dynamic collection of sets $\mathbf{S} = \{S_1, \ldots, S_r\}$. Given an element u, we denote by S_u the set containing u. We will equip each set S_i with a representative element $\text{rep}[S_i]$.[1] This way, checking whether two elements u and v are in the same set amounts to checking whether $\text{rep}[S_u] = \text{rep}[S_v]$. The disjoint-set data structure supports the following operations:

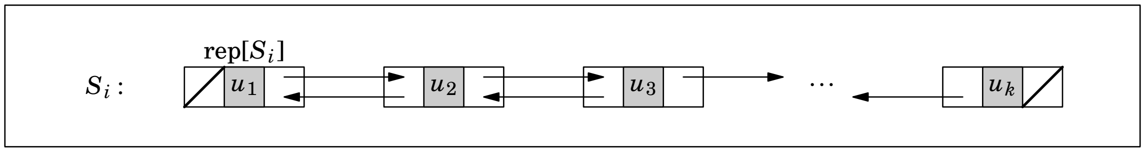
- MAKE-SET(u): Creates a new set containing the single element u.
  - u must not belong to any already existing set
  - of course, u will be the representative element initially
- FIND-SET(u): Returns the representative $\text{rep}[S_u]$.
- UNION(u,v): Replaces S_u and S_v with $S_u \cup S_v$ in $\mathbf{S}$. Updates representative elements as appropriate.

In Lecture 4, we looked at two different implementations of disjoint sets: doubly-linked lists and trees. In this lecture we'll improve each of these two implementations, ultimately obtaining a very efficient tree-based solution.

16.1 Linked-List Implementation

Recall the simple linked-list implementation of disjoint sets that we saw in Lecture 4:

[1] This is not the only way to do things, but it's just as good as any other way. In any case, we'll need to have some symbol representing each set S_i (to serve as the return value of FIND-SET); the choice of what kind of symbol to use is essentially a choice of notation. We have chosen to use a "representative element" of each set rather than create a new symbol.



MAKE-SET(u)	–	initialize as a lone node	$\Theta(1)$
FIND-SET(u)	–	walk left from u until you reach the head of S_u	$\Theta(n)$ worst-case
UNION(u,v)	–	walk right (towards the tail) from u and left (towards the head) from v. Reassign pointers so that the tail of S_u and the head of S_v become neighbors. The representative is updated automatically.	$\Theta(n)$ worst-case

Do we really need to do all that walking? We could save ourselves some time by augmenting each element u with a pointer to rep[S_u], and augmenting rep[S_u] itself with a pointer to the tail. That way, the running time of FIND-SET would be $O(1)$ and all the walking we formerly had to do in UNION would become unnecessary. However, to maintain the new data fields, UNION(u,v) would have to walk through S_v and update each element's "head" field to point to rep[S_u]. Thus, UNION would still take $O(n)$ time in the worst case.

Perhaps amortization can give us a tighter bound on the running time of UNION? At first glance, it doesn't seem to help much. As an example, start with the sets $\{1\}, \{2\}, \ldots, \{n\}$. Perform UNION(2,1), followed by UNION(3,1), and so on, until finally UNION(n,1), so that $S_1 = \{1, \ldots, n\}$. In each call to UNION, we have to walk to the tail of S_1, which is continually growing. The ith call to UNION has to walk through $i-1$ elements, so that the total running time of the $n-1$ UNION operations is $\sum_{i=1}^{n-1}(i-1) = \Theta\left(n^2\right)$. Thus, the amortized cost per call to UNION is $\Theta(n)$.

However, you may have noticed that we could have performed essentially the same operations more efficiently by instead calling UNION(1,2) followed by UNION(1,3), and so on, until finally UNION(1,n). This way, we never have to perform the costly operation of walking to the tail of S_1; we only ever have to walk to the tails of one-element sets. Thus the running time for this smarter sequence of $n-1$ UNION operations is $\Theta(n)$, and the amortized cost per operation is $O(1)$.

The lesson learned from this analysis is that, when performing a UNION operation, it is best to always merge the smaller set into the larger set, i.e., the representative element of the combined set should be chosen equal to the representative of the larger constituent—that way, the least possible amount of walking has to occur. To do this efficiently, we ought to augment each S_i with a "size" field, which we'll call $S_i.weight$ (see Figure 16.1).

It turns out that the "smaller into larger" strategy gives a significant improvement in the amortized worst-case running time of the UNION operation. We'll show that the total running time of any sequence of UNION operations on a disjoint-set data structure with n elements (i.e., in which MAKE-SET is called n times) is $O(n \lg n)$. Thus, the running time of m operations, n of which are MAKE-SET operations, is

$$O\left(m + n \lg n\right).$$

To start, focus on a single element u. We'll show that the total amount of time spent updating u's "head" pointer is $O(\lg n)$; thus, the total time spent on all UNION operations is $O(n \lg n)$. When u is added to the structure via a call to MAKE-SET, we have $S_u.weight = 1$. Then, every time S_u merges with another set S_v, one of the following happens:

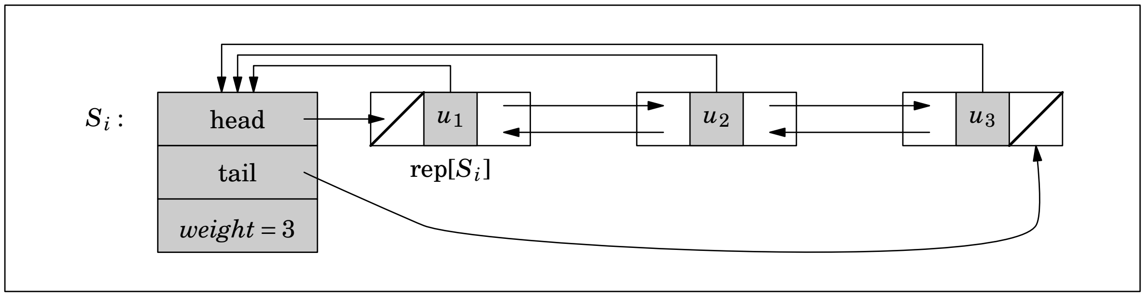


Figure 16.1. A linked list augmented with data fields for the head, the tail, and the size (weight).

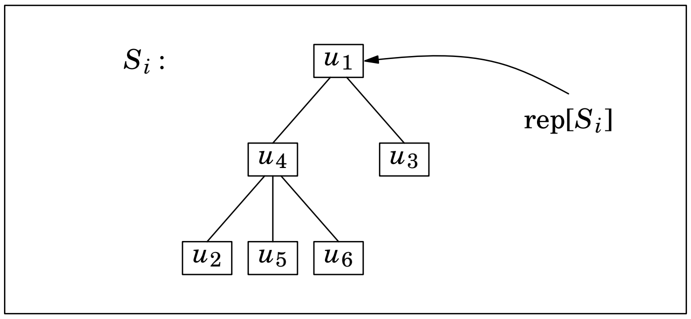
- $S_u.weight > S_v.weight$. Then no update to u's "head" pointer is needed.
- $S_v.weight \geq S_u.weight$. Then, we update u's "head" pointer. Also, in this case, the value of $S_u.weight$ at least doubles.

Because $S_u.weight$ at least doubles every time we update u's "head" pointer, and because $S_u.weight$ can only be at most n, it follows that the total number of times we update u's "head" pointer is at most $\lg n$. Thus, as above, the total cost of all UNION operations is $O(n \lg n)$ and the total cost of any sequence of m operations is $O(m + n \lg n)$.

Exercise 16.1. *With this new augmented structure, do we still need the list to be doubly linked? Which pointers can we safely discard?*

16.2 Forest-of-Trees Implementation

In addition to the linked-list implementation, we also saw in Lecture 4 an implementation of the disjoint-set data structure based on trees:



MAKE-SET(u)	–	initialize new tree with root node u	$\Theta(1)$
FIND-SET(u)	–	walk up tree from u to root	$\Theta(\text{height}) = \Theta(\lg n)$ best-case
UNION(u, v)	–	change rep[S_v]'s parent to rep[S_u]	$O(1) + 2T_{\text{FIND-SET}}$

The efficiency of the basic implementation hinges completely on the height of the tree: the shorter the tree, the more efficient the operations. As the implementation currently stands, the trees could

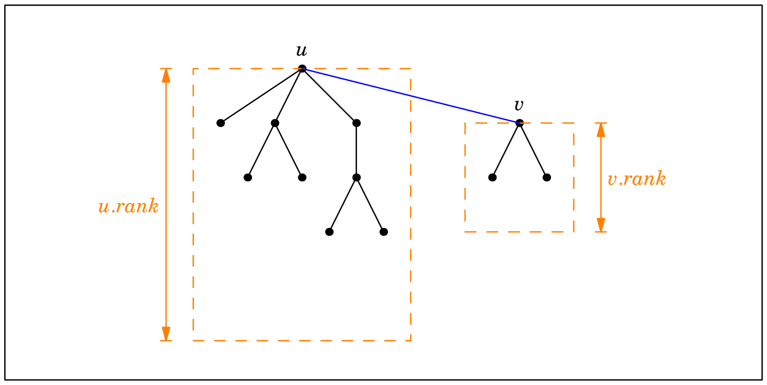


Figure 16.2. Union by rank attempts to always merge the shorter tree into the taller tree, using rank as an estimate (always an overestimate) of height.

be unbalanced and FIND-SET could take as long as $\Theta(n)$ in the worst case. However, this behavior can be dramatically improved, as we will see below.

16.2.1 Union by rank

When we call UNION(u,v), rep$[S_v]$ becomes a child of rep$[S_u]$. Merging S_v into S_u in this way results in a tree of height

$$\max\{height[S_u],\ height[S_v]+1\} \tag{16.1}$$

(why?). Thus, the way to keep our trees short is to always merge the shorter tree into the taller tree. (This is analogous to the "smaller into larger" strategy in the linked-list implementation.) To help us do this, we will introduce a new data field called *rank*. If u is the root of a tree (i.e., if $u = \text{rep}[S_u]$), then $u.rank$ will be an *upper bound* on the height of S_u.[2] In light of (16.1), the pseudocode for UNION will be as follows (see Figure 16.2):

```
Algorithm: UNION($\tilde{u}$,$\tilde{v}$)
1 $u \leftarrow$ FIND-SET($\tilde{u}$)
2 $v \leftarrow$ FIND-SET($\tilde{v}$)
3 if $u.rank = v.rank$ then
4     $u.rank \leftarrow u.rank + 1$
5     $v.parent \leftarrow u$
6 else if $u.rank > v.rank$ then
7     $v.parent \leftarrow u$
8 else
9     $u.parent \leftarrow v$
```

The following lemma shows that UNION preserves the fact that rank is an upper bound on height.

Lemma 16.1. *Suppose initially the following hold:*

- $S_u \neq S_v$

[2] If union by rank is the only improvement we use, then $u.rank$ will actually be the exact height of S_u. But in general, we wish to allow other improvements (such as path compression) to decrease the height of S_u without having to worry about updating ranks. In such cases, the upper bound provided by $u.rank$ may not be tight.

- *S_u has height h_1 and S_v has height h_2*
- *$\text{rep}[S_u].rank = r_1$ and $\text{rep}[S_v].rank = r_2$*
- *$h_1 \le r_1$ and $h_2 \le r_2$.*

Suppose we then call UNION(u,v)*, producing a new set $S = S_u \cup S_v$. Let h be the height of S and let $r = \text{rep}[S].rank$. Then $h \le r$.*

Proof. First, suppose $r_1 > r_2$. Then S_v has been merged into S_u and $r = r_1$. By (16.1), we have

$$\begin{aligned} h &= \max\{h_1,\ h_2+1\} \\ &\le \max\{r_1,\ r_2+1\} \\ &= r_1 \\ &= r. \end{aligned}$$

A similar argument shows that $h \le r$ in the case that $r_2 > r_1$. Finally, suppose $r_1 = r_2$. Then S_v has been merged into S_u and $r = r_1 + 1 = r_2 + 1$, so

$$\begin{aligned} h &= \max\{h_1,\ h_2+1\} \\ &\le \max\{r_1,\ r_2+1\} \\ &= r_2 + 1 \\ &= r. \end{aligned}$$ □

It turns out that the rank of a tree with k elements is always at most $\lg k$. Thus, the worst-case performance of a disjoint-set forest with union by rank having n elements is

MAKE-SET	$O(1)$
FIND-SET	$\Theta(\lg n)$
UNION	$\Theta(\lg n)$.

Exercise 16.2. *Amortization does not help this analysis. Given sufficiently large n and given m which is sufficiently large compared to n, produce a sequence of m operations, n of which are* MAKE-SET *operations (so the structure ultimately contains n elements), whose running time is $\Theta(m \lg n)$.*

Exercise 16.3. *Above we claimed that the rank of any tree with k elements is at most $\lg k$. Use induction to prove this claim. (You may assume that* UNION *is the only procedure that modifies ranks. However, you should not assume anything about the height of a tree except that it is less than the rank.) What is the base case?*

16.2.2 Path compression

The easiest kind of tree to walk up is a *flat* tree, where all non-root nodes are direct children of the root (see Figure 16.3). The idea of path compression is that, every time we invoke FIND-SET and walk up the tree, we should reassign parent pointers to make each node we pass a direct child of the root (see Figure 16.4). This locally flattens the tree. With path compression, the pseudocode for FIND-SET is as follows:

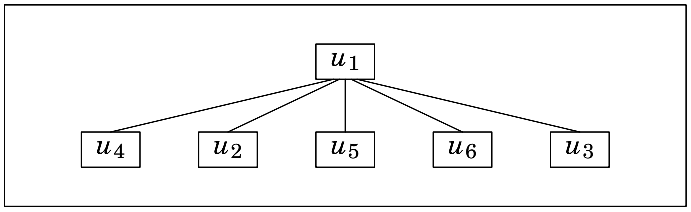


Figure 16.3. In a flat tree, each FIND-SET operation requires us to traverse only one edge.

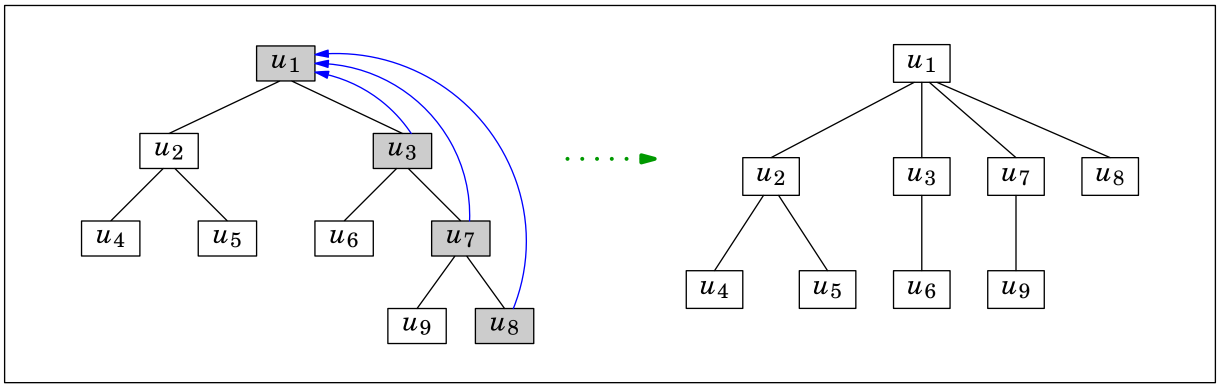


Figure 16.4. With path compression, calling FIND-SET(u_8) will have the side-effect of making u_8 and all of its ancestors direct children of the root.

Algorithm: FIND-SET(u)

```
1 $A \leftarrow \emptyset$
2 $n \leftarrow u$
3 while $n$ is not the root do
4     $A \leftarrow A \cup \{n\}$
5     $n \leftarrow n.parent$
6 for each $x \in A$ do
7     $x.parent \leftarrow n$
8 return $n$
```

What data structure should we use for A? In an ideal world, where n can truly be arbitrarily large, we would probably want A to be a dynamically doubled array of the kind discussed in Lecture 10. In real life, however, some assumptions can be made. For example, if you have less than a petabyte (1024 TB, or 2^{53} bits) of available memory, then the rank (and therefore the height) of any tree is at most $\lg(2^{53}) = 53$, and it would be slightly more efficient to maintain A as a static array of size 53 (with an end-marking sentinel value, perhaps).

It can be shown that, with path compression (but not union by rank), the running time of any sequence of n MAKE-SET operations, f FIND-SET operations, and up to $n-1$ UNION operations is

$$\Theta\left(n + f\left(1 + \log_{2+f/n} n\right)\right).$$

16.2.3 Both improvements together

The punch-line of this lecture is that, taken together, union by rank and path compression produce a spectacularly efficient implementation of the disjoint-set data structure.

Theorem 16.2. *On a disjoint-set forest with union by rank and path compression, any sequence of m operations, n of which are* MAKE-SET *operations, has worst-case running time*

$$\Theta\big(m\alpha(n)\big),$$

where α is the inverse Ackermann function. Thus, the amortized worst-case running time of each operation is $\Theta(\alpha(n))$. If one makes the approximation $\alpha(n) = O(1)$, which is valid for literally all conceivable purposes, then the operations on a disjoint-set forest have $O(1)$ amortized running time.

The proof of this theorem is in §21.4 of CLRS. You can read it if you like; it is not essential. You might also be interested to know that, in a 1989 paper, Fredman and Saks proved that $\Theta(\alpha(n))$ is the fastest possible amortized running time per operation for any implementation of the disjoint-set data structure.

The *inverse Ackermann function α* is defined by

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

where $(k, j) \mapsto A_k(j)$ is the **Ackermann function**. Because the Ackermann function is an extremely rapidly growing function, the inverse Ackermann function α is an extremely slowly growing function (though it is true that $\lim_{n\to\infty} \alpha(n) = \infty$).

The Ackermann function A (at least, one version of it) is defined by

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) \quad \text{(that is, } A_{k-1} \text{ iterated } j+1 \text{ times)} & \text{for } k \geq 1. \end{cases}$$

Some sample values of the Ackermann function are

$$\begin{aligned} A_1(1) &= 3 & A_1(j) &= 2j+1 \\ A_2(1) &= 7 & A_2(j) &= 2^{j+1}(j+1) \\ A_3(1) &= 2047 & & \\ A_4(1) &\gg 10^{80}. & & \end{aligned}$$

By current estimates, 10^{80} is roughly the same order of magnitude as the number of particles in the observable universe. Thus, even if you are a theoretical computer scientist or mathematician, you will still most likely never end up considering a number n so large that $\alpha(n) > 4$.

Exercise 16.4. *Write down a sequence of operations on the disjoint-set forest with union by rank and path compression (including* MAKE-SET *operations) which cause a taller tree to be merged into a shorter tree. Why do we allow this to happen?*