Lecture 15: Disjoint Set Union-Find

Reading: Weiss, Chapt 8 through Sect 8.6.

Equivalence relations: An *equivalence relation* over some set S is a relation that satisfies the following properties for all elements of $a, b, c \in S$.

Reflexive: $a \equiv a$.

Symmetric: $a \equiv b$ then $b \equiv a$

Transitive: $a \equiv b$ and $b \equiv c$ then $a \equiv c$.

Equivalence relations arise in numerous applications. An example includes any sort of "grouping" operation, where every object belongs to some group (perhaps in a group by itself) and no object belongs to more than one group. More formally these groups are called *equivalent classes* and the subdivision of the set into such classes is called a *partition*. For example, suppose we are maintaining a bidirectional communication network. The ability to communicate is an equivalence relation, since if machine a can communicate with machine b, and machine b can communicate with machine c, then machine a can communicate with machine c (e.g. by sending messages through b). Now suppose that a new link is created between two groups, which previously were unable to communicate. This has the effect of *merging* two equivalence classes into one class.

We discuss a data structure that can be used for maintaining equivalence partitions with two operations: (1) *union*, merging to groups together, and (2) *find*, determining which group an element belongs to. This data structure should *not* be thought of as a general purpose data structure for storing sets. In particular, it cannot perform many important set operations, such as splitting two sets, or computing set operations such as intersection and complementation. And its structure is tailored to handle just these two operations. However, there are many applications for which this structure is useful. As we shall see, the data structure is simple and amazingly efficient.

Union-Find ADT: We assume that we have an underlying finite set of elements S. We want to maintain a *partition* of the set. In addition to the constructor, the (abstract) data structure supports the following operations.

`Set s = find(Element x)`: Return an *set identifier* of the set s that contains the element x. A set identifier is simply a special value (of unspecified type) with the property that `find(x) == find(y)` if and only if x and y are in the same set.

`Set r = union(Set s, Set t)`: Merge two sets named s and t into a single set r containing their union. We assume that s, t and r are given as set identifiers. This operations destroys the sets s and t.

Note that there are *no key values* used here. The arguments to the find and union operations are pointers to objects stored in the data structure. The constructor for the data structure is given the elements in the set S and produces a structure in which every element $x \in S$ is in a singleton set $\{x\}$ containing just x.

Inverted Tree Implementation: We will derive our implementation of a data structure for the union-find ADT by starting with a simple structure based on a forest of *inverted trees*. You think of an inverted tree as a multiway tree in which we only store parent links (no child or

sibling links). The root's parent pointer is null. There is no limit on how many children a node can have. The sets are represented by storing the elements of each set in separate tree.

In this implementation a *set identifier* is simply a pointer to the root of an inverted tree. Thus the type `Set` is just an alias for the type `Element` (which is perhaps not very good practice, but is very convenient). We will assume that each element x of the set has a pointer `x.parent`, which points to its parent in this tree. To perform the operation `find(x)`, we walk along parent links and return the root of the tree. This root element is set identifier for the set containing x. Notice that this satisfies the above requirement, since two elements are in the same set if and only if they have the same root. We call this `find1()`. Later we will propose an improvement

The Find Operation (First version)

```
Set find1(Element x) {
    while (x.parent != null) x = x.parent;  // follow chain to root
    return x;                               // return the root
}
```

For example, suppose that $S = \{a, b, c, \ldots, m\}$ and the current partition is:

$$\{a, f, g, h, k, l\}, \{b\}, \{c, d, e, m\}, \{i, j\}.$$

This might be stored in an inverted tree as shown in the following figure. The operation `find(k)` would return a pointer to node `g`.

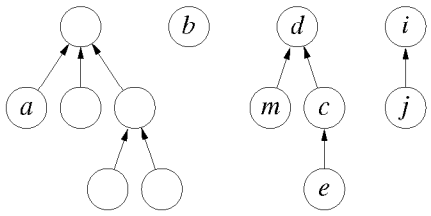

Figure 42: Union-find Tree Example.

Note that there is no particular order to how the individual trees are structured, as long as they contain the proper elements. (Again recall that unlike existing data structures that we have discussed in which there are key values, here the arguments are pointers to the nodes of the inverted tree. Thus, there is never a question of "what" is in the data structure, the issue is "which" tree are you in.) Initially each element is in its own set. To do this we just set all parent pointers to null.

A union is straightforward to implement. To perform the union of two sets, e.g. to take the union of the set containing $\{b\}$ with the set $\{i, j\}$ we just link the root of one tree into the root of the other tree. But here is where it pays to be smart. If we link the root of $\{i, j\}$ into $\{b\}$, the height of the resulting tree is 2, whereas if we do it the other way the height of the tree is only 1. (Recall that *height* of a tree is the maximum number of edges from any leaf to the root.) It is best to keep the tree's height small, because in doing so we make the find's run faster.

In order to perform union's intelligently, we maintain an extra piece of information, which records the height of each tree. For historic reasons we call this height the *rank* of the tree. We assume that the rank is stored as a field in each element. The intelligent rule for doing

unions is to link the root of the set of smaller rank as a child of the root of the set of larger rank.

Observe will only need the rank information for each tree root, and each tree root has no need for a parent pointer. So in a really tricky implementation, the ranks and parent pointers can share the same storage field, provided that you have some way of distinguishing them. For example, in our text, parent pointers (actually indices) are stored with positive integers and ranks are stored as negative integers. We will not worry about being so clever.

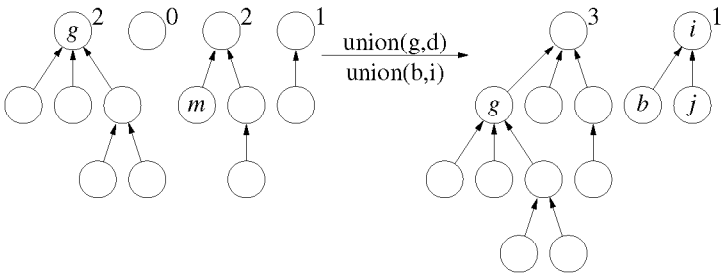

Figure 43: Union-find with ranks.

The code for union operation is shown in the following figure. When updating the ranks there are two cases. If the trees have the same ranks, then the rank of the result is one larger than this value. If not, then the rank of the result is the same as the rank of the higher ranked tree. (You should convince yourself of this.)

Union operation

```
Set union(Set s, Set t) {
    if (s.rank > t.rank) {      // s has strictly higher rank
        t.parent = s            // make s the root (rank does not change)
        return s
    }
    else {                      // t has equal or higher rank
        if (s.rank == t.rank) t.rank++
        s.parent = t            // make t the root
        return t
    }
}
```

Analysis of Running Time: Consider a sequence of m union-find operations, performed on a domain with n total elements. Observe that the running time of the initialization is proportional to n, the number of elements in the set, but this is done only once. Each union takes only constant time, $O(1)$.

In the worst case, find takes time proportional to the height of the tree. The key to the efficiency of this procedure is the following observation, which implies that the tree height is never greater than $\lg m$. (Recall that $\lg m$ denotes the logarithm base 2 of m.)

Lemma: Using the union-find procedures described above any tree with height h has at least 2^h elements.

Proof: Given a union-find tree T, let h denote the height of T, let n denote the number of elements in T, and let u denote the number of unions needed to build T. We prove the lemma by strong induction on the number of unions performed to build the tree. For the

basis (no unions) we have a tree with 1 element of height 0. Since $2^0 = 1$, the basis case is established.

For the induction step, assume that the hypothesis is true for all trees built with strictly fewer than u union operations, and we want to prove the lemma for a union-find tree built with exactly u union operations. Let T be such a tree. Let T' and T'' be the two subtrees that were joined as part of the final union. Let h' and h'' be the heights of T' and T'', respectively, and define n' and n'' similarly. Each of T' and T'' were built with strictly fewer than u union operations. By the induction hypothesis we have

$$n' \geq 2^{h'} \qquad \text{and} \qquad n'' \geq 2^{h''}.$$

Let us assume that T' was made the child of T'' (the other case is symmetrical). This implies that $h' \leq h''$, and $h = \max(h' + 1, h'')$. There are two cases. First, if $h' = h''$ then $h = h' + 1$ and we have

$$n \;=\; n' + n'' \;\geq\; 2^{h'} + 2^{h''} \;=\; 2^{h'+1} \;=\; 2^h.$$

Second, if $h' < h''$ then $h = h''$ and we have

$$n \;=\; n' + n'' \;\geq\; n'' \;\geq\; 2^{h''} \;=\; 2^h.$$

In either case we get the desired result.

Since the unions's take $O(1)$ time each, we have the following.

Theorem: After initialization, any sequence of m union's and find's can be performed in time $O(m \log m)$.

Path Compression: It is possible to apply a very simple heuristic improvement to this data structure which provides a significant improvement in the running time. Here is the intuition. If the user of your data structure repeatedly performs find's on a leaf at a very low level in the tree then each such find takes as much as $O(\log n)$ time. Can we improve on this?

Once we know the result of the find, we can go back and "short-cut" each pointer along the path to point directly to the root of the tree. This only increases the time needed to perform the find by a constant factor, but any subsequent find on this node (or any of its ancestors) will take only $O(1)$ time. The operation of short-cutting the pointers so they all point directly to the root is called *path-compression* and an example is shown below. Notice that only the pointers along the path to the root are altered. We present a slick recursive version below as well. Trace it to see how it works.

Find Operation using Path Compression

```
Set find2(Element x) {
    if (x.parent == null) return x          // return root
    else return x.parent = find2(x.parent)  // find root and update parent
}
```

The running time of find2 is still proportional to the depth of node being found, but observe that each time you spend a lot of time in a find, you flatten the search path. Thus the work you do provides a benefit for later find operations. (This is the sort of thing that we observed in earlier amortized analyses.)

Does the savings really amount to anything? The answer is yes. It was actually believed at one time that if path compression is used, then (after initialization) the running time of the

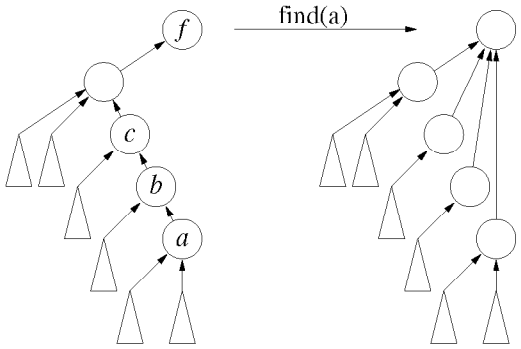

Figure 44: Find using path compression.

algorithm for a sequence of m union and finds was $O(m)$, and hence the amortized cost of each operation is $O(1)$. However, this turned out to be false, but the amortized time is much less than $O(\log m)$.

Analyzing this algorithm is quite tricky. (Our text gives the full analysis if you are interested.) In order to create a bad situation you need to do lots of unions to build up a tree with some real depth. But as soon as you start doing finds on this tree, it very quickly becomes very flat again. In the worst case we need to an immense number of unions to get high costs for the finds.

To give the analysis (which we won't prove) we introduce two new functions, $A(m, n)$ and $\alpha(n)$. The function $A(m, n)$ is called *Ackerman's function*. It is famous for being just about the fastest growing function imaginable.

$$\begin{aligned} A(1, j) &= 2^j \qquad \text{for } j \geq 1, \\ A(i, 1) &= A(i-1, 2) \qquad \text{for } i \geq 2, \\ A(i, j) &= A(i-1, A(i, j-1)) \qquad \text{for } i, j \geq 2. \end{aligned}$$

In spite of its innocuous appearance, this function is a real monster. To get a feeling for how fast this function grows, observe that

$$A(2, j) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}},$$

where the tower of powers of 2 is j high. (Try expanding the recurrence to prove this.) Hence,

$$A(4, 1) = A(3, 2) = A(2, A(3, 1)) = A(2, A(2, 2)) = A(2, 2^{2^2}) = A(2, 16) \approx 10^{80},$$

which is already greater than the estimated number of atoms in the observable universe.

Since Ackerman's function grows so fast, its inverse, called α grows incredibly slowly. Define

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \lg n\}.$$

This definition is somewhat hard to interpret, but the important bottom line is that assuming $\lfloor m/n \rfloor \geq 1$, we have $\alpha(m, n) \leq 4$ as long as m is less than the number of atoms in the universe (which is certainly true for most input sets!) The following result (which we present without proof) shows that an sequence of union-find operations take amortized time $O(\alpha(m, n))$, and so each operation (for all practical purposes) takes amortized constant time.

Theorem: After initialization, any sequence of m union's and find's (using path compression) on an initial set of n elements can be performed in time $O(m\alpha(m,n))$ time. Thus the amortized cost of each union-find operation is $O(\alpha(m,n))$.