Illinois Institute of Technology
Department of Computer Science

Lecture 15: March 7, 2016

CS 430 Introduction to Algorithms
Spring Semester, 2016

Union/Find

It is useful in many applications to have a structure that handles groups of disjoint sets (chapter 21 of CLRS). In particular, we support the following three operations:

- Make-Set: Creates a new set consisting of a single element. Since the sets are disjoint, we assume that this element is not already contained in any set.
- Union: Forms the union of two existing sets into one new set.
- Find-Set: Returns a pointer to the representative of the set containing an element.

The notion of the *representative* of a set deserves further comment. To identify a set, we return a pointer to any element in the set. The only constraint we make on the element chosen is that if the set does not change between calls to Find-Set, we must return the same representative.

We analyze the algorithms implementing these operations in terms of n, the number of Make-Set operations, and m, the total number of Make-Set, Union, and Find-Set operations. Clearly, then, $n \leq m$.

Linked-list representation

One simple approach is to represent a set as a linked list. Each element contains two pointers, one to the next element (as in a simple linked list) and one to the head of the list. The head serves as the set representative.

In this representation, the implementations of Make-Set and Find-Set are simple and require constant time. Union is slightly more involved. We first attach one linked list, x, to the end of the other, y, but we then must update the pointers to the head for every element of x to point to the head of y. As is shown in section 21.2 of CLRS, this yields an amortized time bound of $\Theta(n)$ per operation.

Union by size

Notice that the Union operation is symmetric: Union(x, y) is functionally equivalent to Union(y, x). If $|x| > |y|$, though, it is clearly beneficial to add y to the tail of x rather than adding x to the tail of y, as we then have fewer pointers to update.

In fact, we can quantify the improvement. Consider a single element x. How often can x change its head pointer? It is originally the only element in a set, and any larger set will "swallow" x. But at that point it is in a set of size at least 2, and it can be "swallowed" only by another set of size at least 2. Then it is in a set of size at least 4, and so on. In short, each time x changes its head pointer, the set containing x must double in size. Since a set can double in size at most $\log n$ times, x changes its head pointer at most $\log n$

times. Thus the total time required to update n objects is $O(n \log n)$, giving an amortized cost of $O(\log n)$ per operation.

Disjoint-set forests

We consider a different representation for disjoint sets, namely a collection of trees. In a sense, these trees are "upside-down" trees—each node points to its parent rather than to its children. The root of the tree is the set representative.

A simple implementation yields similar results to those of the linked-list representation. Make-Set creates a singleton tree. Find-Set follows parent pointers until the root is located. Union makes the root of one tree point to the root of the other.

By introducing two simple heuristics, we can guarantee remarkable bounds on the running time. The first heuristic, *union by rank*, is similar to the union-by-size heuristic we considered earlier. The *rank* of a node is an upper bound on the height of the node. The rank of a singleton set, as created by Make-Set, is 0, and Find-Set does not affect the rank. When we construct the Union of two sets, we make the root of higher rank the parent of the root of lower rank or, if the ranks are equal, we choose one arbitrarily and increment its rank.

The second heuristic, *path compression*, in essence flattens the section of a tree traversed by Find-Set and makes all elements in the path from the node we are finding to the root point directly to the root.

Ackermann's function and its inverse

Ackermann's function is defined as follows:[1]

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

We can begin to compute a table of values of Ackermann's function, but we cannot get very far before the values grow too tremendously to even consider:

	$j=0$	$j=1$	$j=2$	$j=3$	$\cdots$
$k=0$	1	2	3	4	$\cdots$
$k=1$	1	3	5	7	$\cdots$
$k=2$	1	7	23	63	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

We define $\alpha(n) = \min\{k | A_k(1) \geq n\}$. It is clear by looking at the above table that for any reasonable value of n, $\alpha(n) \leq 4$ since $A_4(1)$ is far greater than the estimated number of atoms in the observable universe (approximately 10^{80}).

Why is this relevant to union/find? It turns out that the Find-Set operation with union by rank and path compression runs in $O(m\alpha(n))$ time for m operations, yielding an amortized cost of $O(\alpha(n))$ time per operation. Although this is theoretically greater than $O(1)$, as a practical matter it may be treated as a constant.

[1] This definition of Ackermann's function is nonstandard, but it is the one given in CLRS and the one used in the analysis of union/find.

Relaxing the bound (optional)

Rather than deriving the $\alpha(n)$ bound, we derive a weaker bound that is easier to obtain: $O(\lg^* n)$. Define *tower*(k) as a "tower" of k exponentiated 2's. The inverse of the tower function is the $\lg^*$ function. Although $\lg^*$ grows more quickly than α, by extending the second row of the table slightly, it becomes clear that, for any reasonable value of n, $\lg^* n \leq 7$. As a practical matter, a bound of $O(\lg^* n)$ is not significantly worse than a bound of $O(\alpha(n))$.

In order to prove this bound, we partition the range $[0 \ldots \lg n]$ of possible ranks for a node x into blocks by placing $\operatorname{rank}(x)$ in block $\lg^* (\operatorname{rank}(x))$. Then

$$\lambda(x) = \begin{cases} 3 & \text{if } x \text{ is a root or } \operatorname{rank}(x) = 0, \\ 2 & \text{if } x \text{ is not a root and } \lfloor \lg \operatorname{rank}(x) \rfloor = \lfloor \lg \operatorname{rank}(\operatorname{parent}(x)) \rfloor, \\ 1 & \text{otherwise,} \end{cases}$$

and

$$\delta(x) = \begin{cases} 0 & \text{if } \lambda(x) = 3, \\ \operatorname{rank}(\operatorname{parent}(x)) - \operatorname{rank}(x) & \text{if } \lambda(x) = 2, \\ \lfloor \lg \operatorname{rank}(\operatorname{parent}(x)) \rfloor & \text{if } \lambda(x) = 1 \text{ and} \\ \quad - \lfloor \lg \operatorname{rank}(x) \rfloor & \quad \text{block(rank(parent(x)))} \\ & \quad\quad = \text{block(rank(x)),} \\ \operatorname{rank}(x) & \text{otherwise.} \end{cases}$$

Intuitively, $\delta(x)$ counts the difference between the rank of x and the rank of its parent. As this difference grows, we switch to counting the difference in their logs. $\lambda(x)$ records this difference, with decreasing $\lambda(x)$ generally corresponding to increasing difference.

Using these definitions, we define the potential of a single node as

$$\phi(x) = \lambda(x) \operatorname{rank}(x) - \delta(x).$$

The potential of the whole structure is simply the sum of the potentials for each node:

$$\Phi = \sum_{x \in \text{node}} \phi(x).$$

Note that $\lambda(x) \operatorname{rank}(x) \geq \delta(x)$, so that $\phi(x) \geq 0$ for all nodes x.

Now we can begin the analysis of the operations. For the operation *makeset*(x) we have an actual cost of $O(1)$. The node's rank is initialized to zero, and hence $\delta(x)$ is 0, so there is no change in potential and hence the amortized cost of *makeset*(x) is $O(1)$.

Consider a *find* operation that starts at a node x_0 and follows the path $x_0, x_1, \ldots, x_l$, where $\operatorname{parent}(x_i) = x_{i+1}$ for $i < l$. The actual cost of the *find* operation is $O(l)$. Of course the path compression does not effect x_l or x_{l-1}. Since there are only $\lg^* n$ blocks, there are at most $\lg^* n$ nodes whose ranks are the last in their block on the find path. Therefore, there are at most $\lg^* n$ nodes $\hat{x}$ where $\operatorname{block}(\operatorname{rank}(\hat{x})) \neq \operatorname{block}(\operatorname{rank}(\operatorname{parent}(\hat{x})))$ and $\lambda(\hat{x}) = 1$. The relationship between changes in λ and δ guarantees that the potential of any "last node in block" node cannot increase, making the amortized cost of updating the parent pointers of these nodes $O(1)$ each, or $O(\lg^* n)$ total.

Now examine what happens to the potential of the other nodes on the find path. If $\lambda(x) = 3$, then x is either the root or a leaf and its potential doesn't change. If this is not the case, then either $\lambda(x) = 2$ or

$\lambda(x) = 1$. If $\lambda(x)$ has the same value before and after the path compression then $\delta(x)$ increases because x get a higher-rank parent and hence $\phi(x)$ decreases. This follows from the monotonicity of the ranks up a find path. This decrease pays for the work of adjusting x's parent pointer.

Because $\lambda(x) = 1$ represents a greater difference between the rank of x and the rank of parent(x) than $\lambda(x) = 2$, it is impossible for $\lambda(x) = 1$ before path compression and $\lambda(x) = 2$ afterwards. Thus, if $\lambda(x)$ changes, it must be true that $\lambda(x) = 2$ before the *find* and the path compression causes $\lambda(x)$ to decrease to 1. In this case we may have a decrease in $\delta(x)$. However, the decrease can never exceed rank(x) since $\text{rank}(\text{parent}(x)) - \text{rank}(x) < \text{rank}(x)$ when $\lambda(x) = 2$. So even if $\delta(x)$ decreased to zero, the decrease in $\lambda(x)$ still causes $\phi(x)$ to decrease. Again, this decrease in potential can be used to pay for updating the parent pointers.

Since the change in potential pays for all the updates except possibly for the $O(\lg^* n)$ to update the pointers for the "last block" nodes, this is the amortized cost of the *find* operation.

The *link*(x, y) operation can change the potential of the new child. The actual cost of this operation is $O(1)$. Without loss of generality, suppose the *link* makes x the new child of y. This causes either $\lambda(x)$ to decrease or $\delta(x)$ to increase, so that $\phi(x)$ decreases. The *link* can also cause rank(y) to increase by one; this would increase its potential by $O(1)$. The change in rank might also cause a change in potential in the other children of y. The discussion about the change in potential via path compression in the *find* operation implies that the potential of these nodes cannot increase. Hence the change in potential resulting from a *link* operation is at most $O(1)$ and thus the amortized cost is $O(1)$.

Since the largest amortized cost of a single operation is $O(\lg^* n)$, the amortized cost of a sequence of m operations on a structure with n elements is $O(m \lg^* n)$.