**The Disjoint Sets Class**

we describe an efficient data structure to solve the equivalence problem. The implementation is also extremely fast, requiring constant average time per operation Topics of coverage are *. . .*

\_ Show how it can be implemented with minimal coding effort.

\_ Greatly increase its speed, using just two simple observations.

\_ Analyze the running time of a fast implementation.

**8.1 Equivalence Relations**

a. A **relation** *R* is defined on a set *S* if for every pair of elements (*a*, *b*), *a*, *b* ∈ *S*, *aR b* is either true or false. If *a R b* is true, then we say that *a* is related to *b*.

b. An **equivalence relation** is a relation *R* that satisfies three properties: 1. (*Reflexive*) *a R a*, for all *a* ∈ *S*.

2. (*Symmetric*) *a R b* if and only if *bR a*.

3. (*Transitive*) *aR b* and *bR c* implies that *aR c*.

c. **examples.**

The ≤ relationship is not an equivalence relationship. Although it is reflexive, since *a* ≤ *a*, and transitive, since *a* ≤ *b* and *b* ≤ *c* implies *a* ≤ *c*, it is not symmetric, since *a* ≤ *b* does not imply *b* ≤ *a*.

**Electrical connectivity,** where all connections are by metal wires, is an equivalence relation. The relation is clearly reflexive, as any component is connected to itself. If *a* is electrically connected to *b*, then *b* must be electrically connected to *a*, so the relation is symmetric. Finally, if *a* is connected to *b* and *b* is connected to *c*, then *a* is connected to *c*.

Thus electrical connectivity is an equivalence relation.

Two cities are related if they are in the same country. It is easily verified that this is an equivalence relation. Suppose town *a* is related to *b* if it is possible to travel from *a* to *b* by taking roads. This relation is an equivalence relation if all the roads are two-way.

**8.2 The Dynamic Equivalence Problem**

Given an equivalence relation ∼, the natural problem is to decide, for any *a* and *b*, if *a* ∼*b*. If the relation is stored as a two-dimensional array of Boolean variables, then, of course, this can be done in constant time. As an example, suppose the equivalence relation is defined over the five element set {*a*1, *a*2, *a*3, *a*4, *a*5}. Then there are 25 pairs of elements, each of which is either related or not. However, the information *a*1 ∼ *a*2, *a*3 ∼ *a*4, *a*5 ∼ *a*1, *a*4 ∼ *a*2 implies that all pairs are related. We would like to be able to infer this quickly.

The **equivalence class** of an element *a* ∈ *S* is the subset of *S* that contains all the elements that are related to *a*. Notice that the equivalence classes form a partition of *S*: Every member of *S* appears in exactly one equivalence class. To decide if *a* ∼ *b*, we need only to check whether *a* and *b* are in the same equivalence class. The input is initially a collection of *N* sets, each with one element.

This initial representation is that all relations (except reflexive relations) are false. Each set has a different element, so that *Si* ∩ *Sj* = ∅; this makes the sets **disjoint**.

There are two permissible operations. The first is find, which returns the name of the set (that is, the equivalence class) containing a given element. The second operation adds relations. If we

1

want to add the relation *a* ∼ *b*, then we first see if *a* and *b* are already related. This is done by performing finds on both *a* and *b* and checking whether they are in the same equivalence class. If they are not, then we apply union.1 This operation merges the two equivalence classes containing *a* and *b* into a new equivalence class. From a set point of view, the result of ∪is to create a new set *Sk* = *Si* ∪*Sj*, destroying the originals and preserving the disjointness of all the sets.

The algorithm to do this is frequently known as the disjoint set **union/find algorithm** for this reason.

Our second observation is that the name of the set returned by find is actually fairly arbitrary. All that really matters is that find(a)==find(b) is true if and only if a and b are in the same set.

These operations are important in many graph theory problems and also in compilers which process equivalence (or type) declarations.

There are two strategies to solve this problem. One ensures that the find instruction can be executed in constant worst-case time, and the other ensures that the union instruction can be executed in constant worst-case time. It has recently been shown that both cannot be done simultaneously in constant worst-case time.

We will now briefly discuss the first approach.

For the find operation to be fast, we could maintain, in an array, the name of the equivalence class for each element. Then find is just a simple *O*(1) lookup. Suppose we want to perform union(a,b). Suppose that a is in equivalence class *i* and b is in equivalence class *j*. Then we scan down the array, changing all *i*s to *j*. Unfortunately, this scan takes *\_*(*N*). Thus, a sequence of *N* −

1 unions (the maximum, since then everything is in one set) would take *\_*(*N*2) time. If there are *\_*(*N*2) find operations, this performance is fine, since the total running time would then amount to *O*(1) for each union or find operation over the course of the algorithm. If there are fewer finds, this bound is not acceptable.

One idea is to keep all the elements that are in the same equivalence class in a linked list. This saves time when updating, because we do not have to search through the entire array. This by itself does not reduce the asymptotic running time, because it is still possible to perform *\_*(*N*2) equivalence class updates over the course of the algorithm. If we also keep track of the size of each equivalence class, and when performing unions we change the name of the smaller equivalence class to the larger, then the total time spent for *N* − 1 merges is *O*(*N* log*N*). The reason for this is that each element can have its equivalence class changed at most log*N* times, since every time its class is changed, its new equivalence class is at least twice as large as its old. Using this strategy, any sequence of *M* finds and up to *N* − 1 unions takes at most *O*(*M* + *N* log*N*) time.

**8.3 Basic Data Structure**

a find operation do not return any specific name, just that finds on two elements return the same answer if and only if they are in the same set. One idea might be to use a tree to represent each set, since each element in a tree has the same root. Thus, the root can be used to name the set. We will represent each set by a tree. Initially, each set contains one element. The trees we will use are not necessarily binary trees, but their representation is easy, because the only information we will need is a parent link. The name of a set is given

by the node at the root. Since only the name of the parent is required, we can assume that this tree is stored implicitly in an array: Each entry s[i] in the array represents the parent of element *i*. If *i* is a root, then s[i] = −1. In the forest in Figure 8.1, s[i] = −1 for 0 ≤ *i <* 8. As with binary

2

heaps, we will draw the trees explicitly, with the understanding that an array is being used. Figure 8.1 shows the explicit representation. We will draw the root’s parent link vertically for convenience. To perform a union of two sets, we merge the two trees by making the parent link of one tree’s root link to the root node of the other tree. It should be clear that this operation takes constant time. Figures 8.2, 8.3, and 8.4 represent the forest after each of union(4,5), union(6,7), union(4,6), where we have adopted the convention that the new root after the union(x,y) is x. The implicit representation of the last forest is shown in Figure 8.5.

**Figure 8.1** Eight elements, initially in different sets

A find(x) on element x is performed by returning the root of the tree containing x. The time to perform this operation is proportional to the depth of the node representing x, assuming, of course, that we can find the node representing x in constant time. Using the strategy above, it is possible to create a tree of depth *N* − 1, so the worst-case running

3

time of a find is *\_*(*N*). Typically, the running time is computed for a *sequence* of *M* intermixedinstructions. In this case, *M* consecutive operations could take *\_*(*MN*) time in the worst case. The code in Figures 8.6 through 8.9 represents an implementation of the basic algorithm,

assuming that error checks have already been performed. In our routine, unions are performed on the roots of the trees. Sometimes the operation is performed by passing any two elements and having the union perform two finds to determine the roots. In previously seen data structures, find has always been an accessor, and thus a const member function. Both versions can

4

be supported simultaneously. The mutator is always called, unless the controlling object is unmodifiable. The average-case analysis is quite hard to do. The least of the problems is that the

answer depends on how to define *average* (with respect to the union operation). For instance, in the forest in Figure 8.4, we could say that since there are five trees, there are 5·4 = 20 equally likely results of the next union (as any two different trees can be unioned).

5

**Figure 8.9** A simple disjoint sets find algorithm Of course, the implication of this model is that there is only a 25 chance that the next union will involve the large tree. Another model might say that all unions between any two *elements* in different trees are equally likely, so a larger tree is more likely to be involved in the next union than a smaller tree. In the example above, there is an 8 11 chance that the large tree is involved in the next union, since (ignoring symmetries) there are 6 ways in which to merge two elements in {0, 1, 2, 3}, and 16 ways to merge an element in {4, 5, 6, 7} with an element in {0, 1, 2, 3}. There are still more models and no general agreement on which is the best. The average running time depends on the model; *\_*(*M*), *\_*(*M*log*N*), and *\_*(*MN*) bounds have actually been shown for three different models, although the latter bound is thought to be more realistic. Quadratic running time for a sequence of operations is generally unacceptable.

**8.4 Smart Union Algorithms**

The unions above were performed rather arbitrarily, by making the second tree a subtree of

6

the first. A simple improvement is always to make the smaller tree a subtree of the larger, breaking ties by any method; we call this approach **union-by-size**. The three unions in the preceding example were all ties, and so we can consider that they were performed by size. If the next operation were union(3,4), then the forest in Figure 8.10 would form. Had the size heuristic not been used, a deeper tree would have been formed (Fig. 8.11). We can prove that if unions are done by size, the depth of any node is never more than log*N*. To see this, note that a node is initially at depth 0. When its depth increases as a result of a union, it is placed in a tree that is at least twice as large as before. Thus, its depth can be increased at most log*N* times. (We used this argument in the quick-find algorithm at the end of Section 8.2.) This implies that the running time for a find operation is *O*(log*N*), and a sequence of *M* operations takes *O*(*M*log*N*). The tree in Figure 8.12 shows the worst tree possible after 16 unions and is obtained if all unions are between equal-sized trees. To implement this strategy, we need to keep track of the size of each tree. Since we are really just using an array, we can have the array entry of each root contain the *negative* of

the size of its tree. Thus, initially the array representation of the tree is all −1s. When a union is performed, check the sizes; the new size is the sum of the old. Thus, union-by-size is not

at all difficult to implement and requires no extra space. It is also fast, on average. For virtually all reasonable models, it has been shown that a sequence of *M* operations requires *O*(*M*) average time if union-by-size is used. This is because when random unions are performed, generally very small (usually one-element) sets are merged with large sets throughout the algorithm. An alternative implementation, which also guarantees that all the trees will have depth at most *O*(log*N*), is **union-by-height.**

7



We keep track of the height, instead of the size, of each tree and perform unions by making the shallow tree a subtree of the deeper tree. This is an easy algorithm, since the height of a tree increases only when two equally deep trees are joined (and then the height goes up by one). Thus, union-by-height is a trivial modification of union-by-size. Since heights of zero would not be negative, we actually store the negative of height, minus an additional 1. Initially, all entries are −1. Figure 8.13 shows a forest and its implicit representation for both union-by-size and union-by-height. The code in Figure 8.14 implements union-by-height.

8

**8.5 Path Compression**

The union/find algorithm, as described so far, is quite acceptable for most cases. It is very simple and linear on average for a sequence of *M* instructions (under all models). However, the worst case of *O*(*M*log*N*) can occur fairly easily and naturally. For instance, if we put all the sets on a queue and repeatedly dequeue the first two sets and enqueuer the union, the worst case occurs. If there are many more finds than unions, this running time is worse than that of the quick-find algorithm. Moreover, it should be clear that there are probably no more improvements possible for the union algorithm. This is based on the observation that any method to perform the unions will yield the same worst-case trees, since it must break ties arbitrarily. Therefore, the only way to speed the algorithm up, without reworking the data structure entirely, is to do something clever on the find operation. The clever operation is known as **path compression**. Path compression is performed during a find operation and is independent of the strategy used to perform unions. Suppose the operation is find(x). Then the effect of path compression is that *every* node on the path from x to the root has its parent changed to the root. Figure 8.15 shows the effect of path compression after find(14) on the generic worst tree of Figure 8.12. The effect of path compression is that with an extra two link changes, nodes 12 and 13 are now one position closer to the root and nodes 14 and 15 are now two positions closer. Thus, the fast future accesses on these nodes will pay (we hope) for the extra work to do the path compression. As the code in Figure 8.16 shows, path compression is a trivial change to the basic find algorithm. The only change to the find routine (besides the fact that it is no longer a const member function) is that s[x] is made equal to the value returned by find; thus, after the root of the set is found recursively, x’s parent link references it.

9

This occurs recursively to every node on the path to the root, so this implements path compression. When unions are done arbitrarily, path compression is a good idea, because there is an abundance of deep nodes and these are brought near the root by path compression. It has been proven that when path compression is done in this case, a sequence of M



operations requires at most *O*(*M*log*N*) time. It is still an open problem to determine what the average-case behavior is in this situation. Path compression is perfectly compatible with union-by-size, and thus both routines can be implemented at the same time. Since doing union by-size by itself is expected to execute a sequence of *M* operations in linear time, it is not clear that the extra pass involved in path compression is worthwhile on average. Indeed, this problem is still open. However, as we shall see later, the combination of path compression and a smart union rule guarantees a very efficient algorithm in all cases. Path compression is not entirely compatible with union-by-height, because path compression can change the heights of the trees. It is not at all clear how to recompute them efficiently. The answer is do not! Then the heights stored for each tree become estimated heights (sometimes known as **ranks**), but it turns out that **union-by-rank** (which is what this has now become) is just as efficient in theory as union-by size. Furthermore, heights are updated less often than sizes. As with union-by-size, it is not clear whether path compression is worthwhile on average.

10