CHAPTER 8

The Disjoint Sets Class

In this chapter, we describe an efficient data structure to solve the equivalence problem. The data structure is simple to implement. Each routine requires only a few lines of code, and a simple array can be used. The implementation is also extremely fast, requiring constant average time per operation. This data structure is also very interesting from a theoretical point of view, because its analysis is extremely difficult; the functional form of the worst case is unlike any we have yet seen. For the disjoint sets data structure, we will . . .

- 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.
- See a simple application.

8.1 Equivalence Relations

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

An **equivalence relation** is a relation R that satisfies three properties:

1. (*Reflexive*) $a\,R\,a$, for all $a \in S$.
2. (*Symmetric*) $a\,R\,b$ if and only if $b\,R\,a$.
3. (*Transitive*) $a\,R\,b$ and $b\,R\,c$ implies that $a\,R\,c$.

We will consider several examples.

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

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 $\sim$, the natural problem is to decide, for any a and b, if $a \sim b$. If the relation is stored as a two-dimensional array of Boolean variables, then, of course, this can be done in constant time. The problem is that the relation is usually not explicitly, but rather implicitly, defined.

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 $a_1 \sim a_2$, $a_3 \sim a_4$, $a_5 \sim a_1$, $a_4 \sim a_2$ implies that all pairs are related. We would like to be able to infer this quickly.

The **equivalence class** of an element $a \in S$ 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 $a \sim b$, we need only to check whether a and b are in the same equivalence class. This provides our strategy to solve the equivalence problem.

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 $S_i \cap S_j = \emptyset$; 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 want to add the relation $a \sim b$, then we first see if a and b are already related. This is done by performing `find`s 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 $\cup$ is to create a new set $S_k = S_i \cup S_j$, 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.

This algorithm is *dynamic* because, during the course of the **algorithm**, the sets can change via the `union` operation. The algorithm must also operate **online**: When a `find` is performed, it must give an answer before continuing. Another possibility would be an **offline** algorithm. Such an algorithm would be allowed to see the entire sequence of `union`s and `find`s. The answer it provides for each `find` must still be consistent with all the `union`s that were performed up until the `find`, but the algorithm can give all its answers after it has seen *all* the questions. The difference is similar to taking a written exam (which is generally offline—you only have to give the answers before time expires) or an oral exam (which is online, because you must answer the current question before proceeding to the next question).

Notice that we do not perform any operations comparing the relative values of elements but merely require knowledge of their location. For this reason, we can assume that all the elements have been numbered sequentially from 0 to $N - 1$ and that the numbering can

[1] `union` is a (little-used) reserved word in C++. We use it throughout in describing the union/find algorithm, but when we write code, the member function will be named `unionSets`.

be determined easily by some hashing scheme. Thus, initially we have $S_i = \{i\}$ for $i = 0$ through $N - 1$.[2]

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. We will see an application later.

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 is to j. Unfortunately, this scan takes $\Theta(N)$. Thus, a sequence of $N - 1$ `union`s (the maximum, since then everything is in one set) would take $\Theta(N^2)$ time. If there are $\Omega(N^2)$ `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 `find`s, 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 $\Theta(N^2)$ equivalence class updates over the course of the algorithm.

If we also keep track of the size of each equivalence class, and when performing `union`s 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 `find`s and up to $N - 1$ `union`s takes at most $O(M + N \log N)$ time.

In the remainder of this chapter, we will examine a solution to the union/find problem that makes `union`s easy but `find`s hard. Even so, the running time for any sequence of at most M `find`s and up to $N - 1$ `union`s will be only a little more than $O(M + N)$.

8.3 Basic Data Structure

Recall that the problem does not require that a `find` operation return any specific name, just that `find`s 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. (Recall that a collection of trees is known as a **forest**.) Initially, each set contains one element. The trees we will use are not necessarily binary trees, but their representation is

[2] This reflects the fact that array indices start at 0.

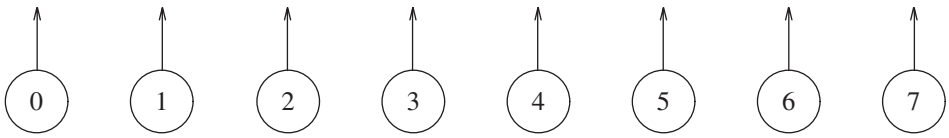


Figure 8.1 Eight elements, initially in different sets

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 $0 \leq i < 8$. As with binary 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.

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

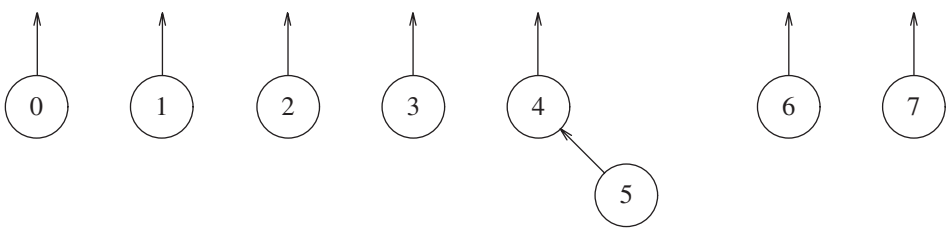


Figure 8.2 After `union(4,5)`

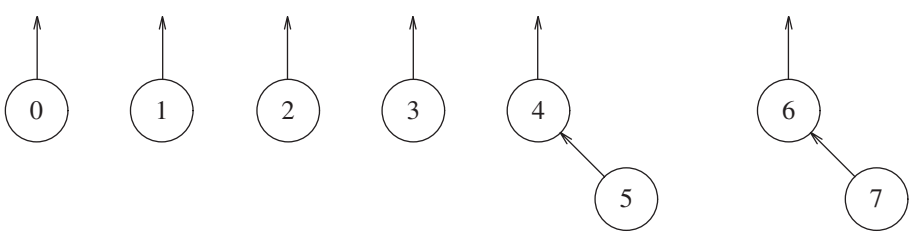


Figure 8.3 After `union(6,7)`

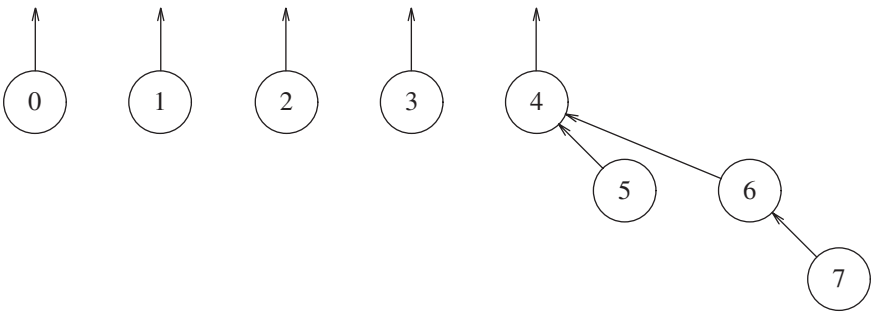


Figure 8.4 After `union(4,6)`

–1	–1	–1	–1	–1	4	4	6
0	1	2	3	4	5	6	7

Figure 8.5 Implicit representation of previous tree

time of a `find` is $\Theta(N)$. Typically, the running time is computed for a *sequence* of M intermixed instructions. In this case, M consecutive operations could take $\Theta(MN)$ 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, `union`s are performed on the roots of the trees. Sometimes the operation is performed by passing any two elements and having the `union` perform two `find`s to determine the roots. In previously seen data structures, `find` has always been an accessor, and thus a `const` member function. Section 8.5 describes a mutator version that is more efficient. Both versions can

```
1   class DisjSets
2   {
3     public:
4       explicit DisjSets( int numElements );
5
6       int find( int x ) const;
7       int find( int x );
8       void unionSets( int root1, int root2 );
9
10    private:
11      vector<int> s;
12  };
```

Figure 8.6 Disjoint sets class interface

```
1  /**
2   * Construct the disjoint sets object.
3   * numElements is the initial number of disjoint sets.
4   */
5  DisjSets::DisjSets( int numElements ) : s{ numElements, - 1 }
6  {
7  }
```

Figure 8.7 Disjoint sets initialization routine

```
1  /**
2   * Union two disjoint sets.
3   * For simplicity, we assume root1 and root2 are distinct
4   * and represent set names.
5   * root1 is the root of set 1.
6   * root2 is the root of set 2.
7   */
8  void DisjSets::unionSets( int root1, int root2 )
9  {
10     s[ root2 ] = root1;
11 }
```

Figure 8.8 `union` (not the best way)

```
1  /**
2   * Perform a find.
3   * Error checks omitted again for simplicity.
4   * Return the set containing x.
5   */
6  int DisjSets::find( int x ) const
7  {
8      if( s[ x ] < 0 )
9          return x;
10     else
11         return find( s[ x ] );
12 }
```

Figure 8.9 A simple disjoint sets `find` algorithm

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 $5 \cdot 4 = 20$ equally likely results of the next `union` (as any two different trees can be `union`ed).

Of course, the implication of this model is that there is only a $\frac{2}{5}$ chance that the next `union` will involve the large tree. Another model might say that all `union`s 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 $\frac{8}{11}$ 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; $\Theta(M)$, $\Theta(M \log N)$, and $\Theta(MN)$ 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. Fortunately, there are several ways of easily ensuring that this running time does not occur.

8.4 Smart Union Algorithms

The `union`s above were performed rather arbitrarily, by making the second tree a subtree of 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 `union`s 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 `union`s 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 `union`s and is obtained if all `union`s are between equal-sized trees (the worst-case trees are binomial trees, discussed in Chapter 6).

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

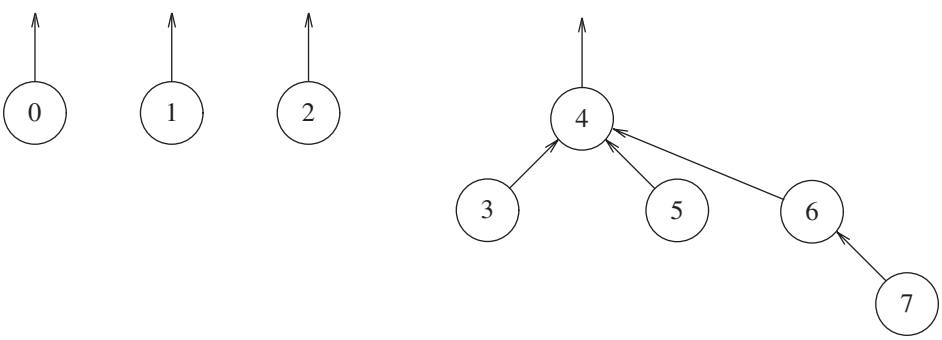


Figure 8.10 Result of union-by-size

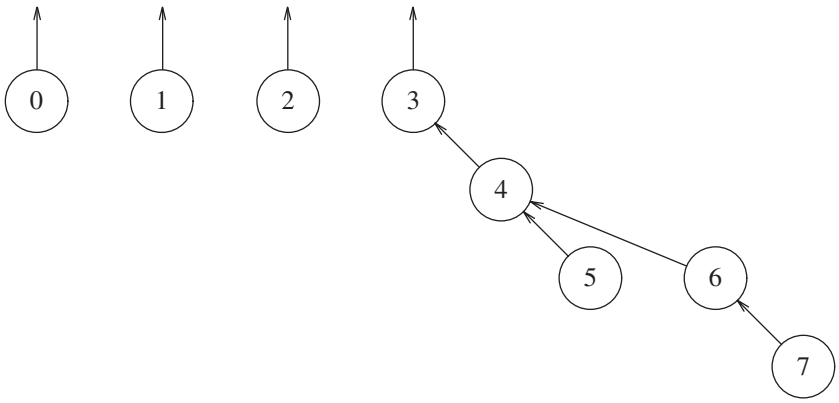


Figure 8.11 Result of an arbitrary `union`

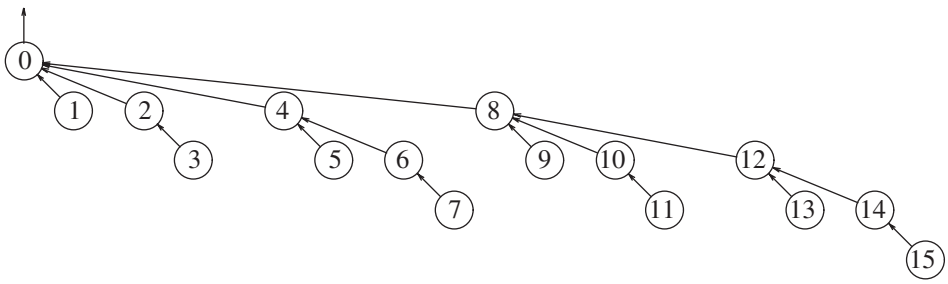
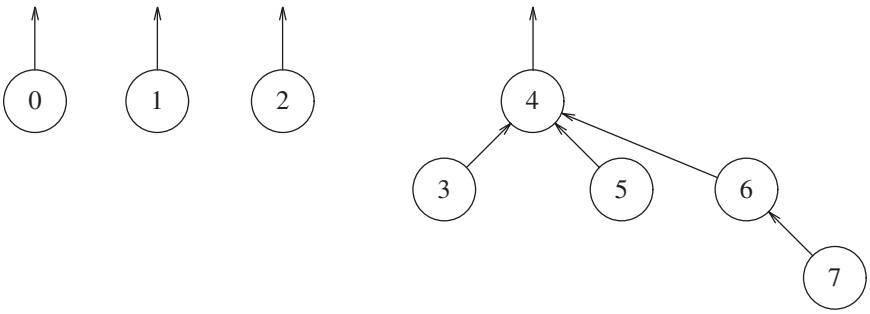


Figure 8.12 Worst-case tree for $N = 16$

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 `union`s 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.** We keep track of the height, instead of the size, of each tree and perform `union`s 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.



–1	–1	–1	4	–5	4	4	6
0	1	2	3	4	5	6	7

–1	–1	–1	4	–3	4	4	6
0	1	2	3	4	5	6	7

Figure 8.13 Forest with implicit representation for union-by-size and union-by-height

```
1   /**
2    * Union two disjoint sets.
3    * For simplicity, we assume root1 and root2 are distinct
4    * and represent set names.
5    * root1 is the root of set 1.
6    * root2 is the root of set 2.
7    */
8   void DisjSets::unionSets( int root1, int root2 )
9   {
10      if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
11          s[ root1 ] = root2;        // Make root2 new root
12      else
13      {
14          if( s[ root1 ] == s[ root2 ] )
15              --s[ root1 ];          // Update height if same
16          s[ root2 ] = root1;        // Make root1 new root
17      }
18  }
```

Figure 8.14 Code for union-by-height (rank)

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 enqueue the union, the worst case occurs. If there are many more `find`s than `union`s, 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 `union`s. 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. This occurs recursively to every node on the path to the root, so this implements path compression.

When `union`s 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

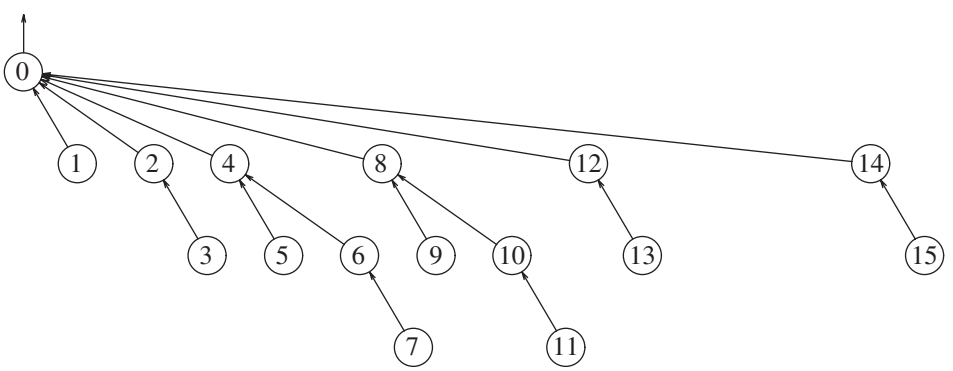


Figure 8.15 An example of path compression

```
1  /**
2   * Perform a find with path compression.
3   * Error checks omitted again for simplicity.
4   * Return the set containing x.
5   */
6  int DisjSets::find( int x )
7  {
8      if( s[ x ] < 0 )
9          return x;
10     else
11         return s[ x ] = find( s[ x ] );
12 }
```

Figure 8.16 Code for disjoint sets `find` with path compression

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. What we will show in the next section is that with either union heuristic, path compression significantly reduces the worst-case running time.

8.6 Worst Case for Union-by-Rank and Path Compression

When both heuristics are used, the algorithm is almost linear in the worst case. Specifically, the time required in the worst case is $\Theta(M\alpha(M, N))$ (provided $M \geq N$), where $\alpha(M, N)$ is an incredibly slowly growing function that for all intents and purposes is at most 5 for any problem instance. However, $\alpha(M, N)$ is not a constant, so the running time is not linear.

In the remainder of this section, we first look at some very slow-growing functions, and then in Sections 8.6.2 to 8.6.4, we establish a bound on the worst case for a sequence of at most $N - 1$ unions and M find operations in an N-element universe in which union is by rank and finds use path compression. The same bound holds if union-by-rank is replaced with union-by-size.

8.6.1 Slowly Growing Functions

Consider the recurrence

$$T(N) = \begin{cases} 0 & N \leq 1 \\ T(\lfloor f(N) \rfloor) + 1 & N > 1 \end{cases} \tag{8.1}$$

In this equation, $T(N)$ represents the number of times, starting at N, that we must iteratively apply $f(N)$ until we reach 1 (or less). We assume that $f(N)$ is a nicely defined function that reduces N. Call the solution to the equation $f^*(N)$.

We have already encountered this recurrence when analyzing binary search. There, $f(N) = N/2$; each step halves N. We know that this can happen at most $\log N$ times until N reaches 1; hence we have $f^*(N) = \log N$ (we ignore low-order terms, etc.). Observe that in this case, $f^*(N)$ is much less than $f(N)$.

Figure 8.17 shows the solution for $T(N)$ for various $f(N)$. In our case, we are most interested in $f(N) = \log N$. The solution $T(N) = \log^* N$ is known as the **iterated logarithm**. The iterated logarithm, which represents the number of times the logarithm needs to be iteratively applied until we reach one, is a very slowly growing function. Observe that $\log^* 2 = 1$, $\log^* 4 = 2$, $\log^* 16 = 3$, $\log^* 65536 = 4$, and $\log^* 2^{65536} = 5$. But keep in mind that 2^{65536} is a 20,000-digit number. So while $\log^* N$ is a growing function, for all intents and purposes, it is at most 5. But we can still produce even more slowly growing functions. For instance, if $f(N) = \log^* N$, then $T(N) = \log^{**} N$. In fact, we can add stars at will to produce functions that grow slower and slower.

8.6.2 An Analysis by Recursive Decomposition

We now establish a tight bound on the running time of a sequence of $M = \Omega(N)$ union/find operations. The `unions` and `finds` may occur in any order, but `unions` are done by rank and `finds` are done with path compression.

$f(N)$	$f^*(N)$
$N-1$	$N-1$
$N-2$	$N/2$
$N-c$	N/c
$N/2$	$\log N$
N/c	$\log_c N$
$\sqrt{N}$	$\log\log N$
$\log N$	$\log^* N$
$\log^* N$	$\log^{**} N$
$\log^{**} N$	$\log^{***} N$

Figure 8.17 Different values of the iterated function

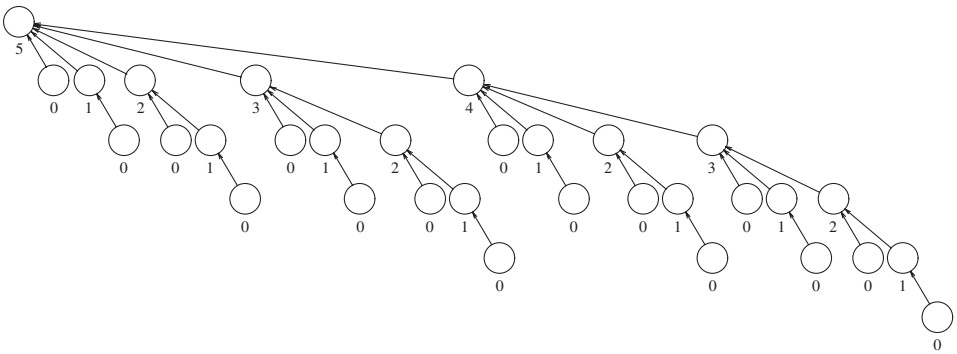


Figure 8.18 A large disjoint set tree (numbers below nodes are ranks)

We begin by establishing two lemmas concerning the properties of the ranks. Figure 8.18 gives a visual picture of both lemmas.

Lemma 8.1

When executing a sequence of `union` instructions, a node of rank $r > 0$ must have at least one child of rank $0, 1, \ldots, r - 1$.

Proof

By induction. The basis $r = 1$ is clearly true. When a node grows from rank $r - 1$ to rank r, it obtains a child of rank $r - 1$. By the inductive hypothesis, it already has children of ranks $0, 1, \ldots, r - 2$, thus establishing the lemma.

The next lemma seems somewhat obvious but is used implicitly in the analysis.

Lemma 8.2

At any point in the union/find algorithm, the ranks of the nodes on a path from the leaf to a root increase monotonically.

Proof

The lemma is obvious if there is no path compression. If, after path compression, some node v is a descendant of w, then clearly v must have been a descendant of w when only `unions` were considered. Hence the rank of v is less than the rank of w.

Suppose we have two algorithms, A and B. Algorithm A works and computes all the answers correctly, but algorithm B does not compute correctly, or even produce useful answers. Suppose, however, that every step in algorithm A can be mapped to an equivalent step in algorithm B. Then it is easy to see that the running time for algorithm B describes the running time for algorithm A exactly.

We can use this idea to analyze the running time of the disjoint sets data structure. We will describe an algorithm B, whose running time is exactly the same as the disjoint sets structure, and then algorithm C, whose running time is exactly the same as algorithm B. Thus any bound for algorithm C will be a bound for the disjoint sets data structure.

Partial Path Compression

Algorithm *A* is our standard sequence of union-by-rank and find with path compression operations. We design an algorithm *B* that will perform the exact same sequence of path compression operations as algorithm *A*. In algorithm *B*, we perform all the unions *prior* to any find. Then each find operation in algorithm *A* is replaced by a **partial find** operation in algorithm *B*. A partial find operation specifies the search item and the node up to which the path compression is performed. The node that will be used is the node that would have been the root at the time the matching find was performed in algorithm A.

Figure 8.19 shows that algorithm *A* and algorithm *B* will get equivalent trees (forests) at the end, and it is easy to see that the exact same amount of parent changes are performed by algorithm *A*'s finds, compared to algorithm *B*'s partial finds. But algorithm *B* should be simpler to analyze, since we have removed the mixing of unions and finds from the equation. The basic quantity to analyze is the number of parent changes that can occur in any sequence of partial finds, since all but the top two nodes in any find with path compression will obtain new parents.

A Recursive Decomposition

What we would like to do next is to divide each tree into two halves: a top half and a bottom half. We would then like to ensure that the number of partial find operations in the top half plus the number of partial find operations in the bottom half is exactly the same as the total number of partial find operations. We would then like to write a formula for the total path compression cost in the tree in terms of the path compression cost in the top half plus the path compression cost in the bottom half. Without specifying how we decide which nodes are in the top half, and which nodes are in the bottom half, we can look at Figures 8.20, 8.21, and 8.22, to see how most of what we want to do can work immediately.

In Figure 8.20, the partial find resides entirely in the bottom half. Thus one partial find in the bottom half corresponds to one original partial find, and the charges can be recursively assigned to the bottom half.

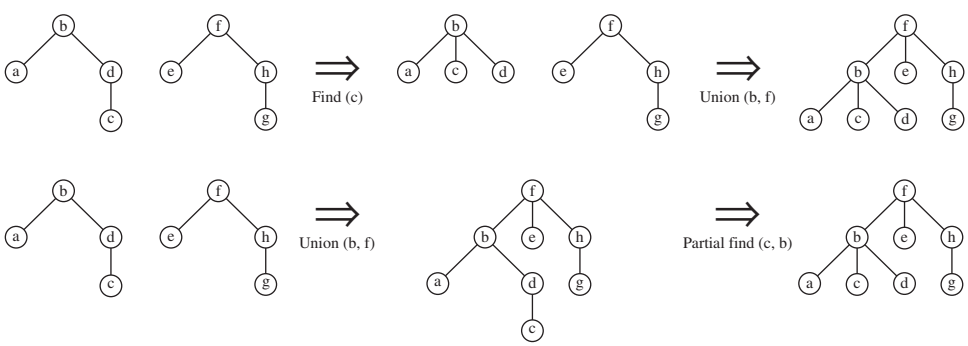


Figure 8.19 Sequences of union and find operations replaced with equivalent cost of union and partial find operations

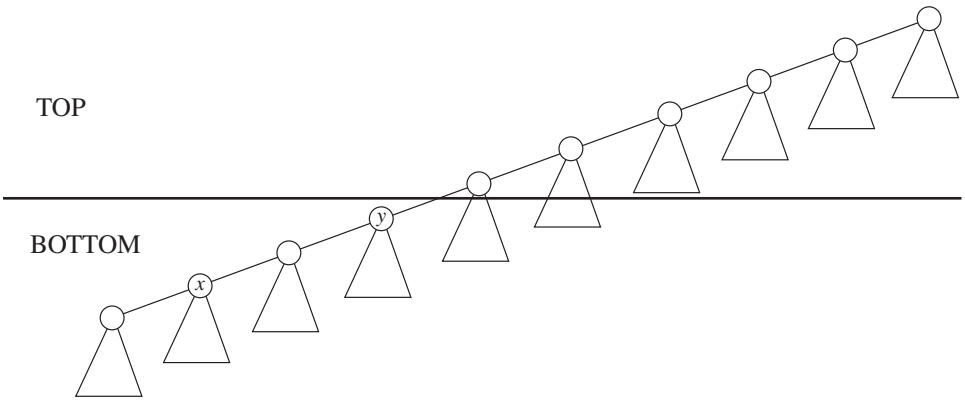


Figure 8.20 Recursive decomposition, case 1: Partial find is entirely in bottom

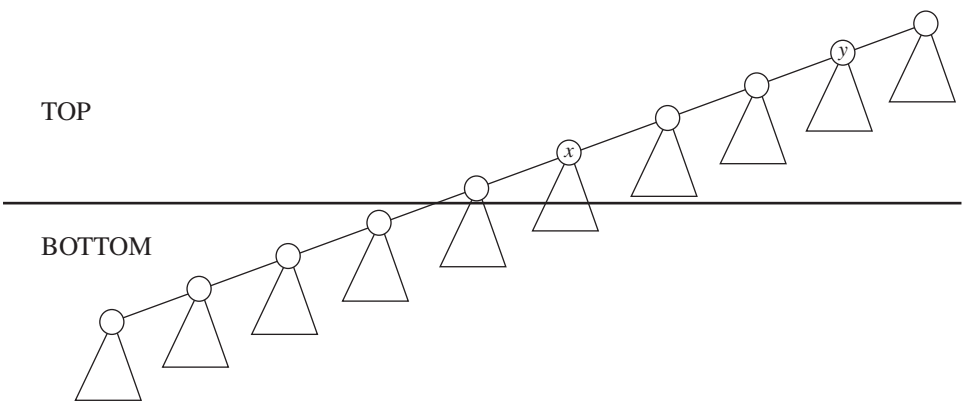


Figure 8.21 Recursive decomposition, case 2: Partial find is entirely in top

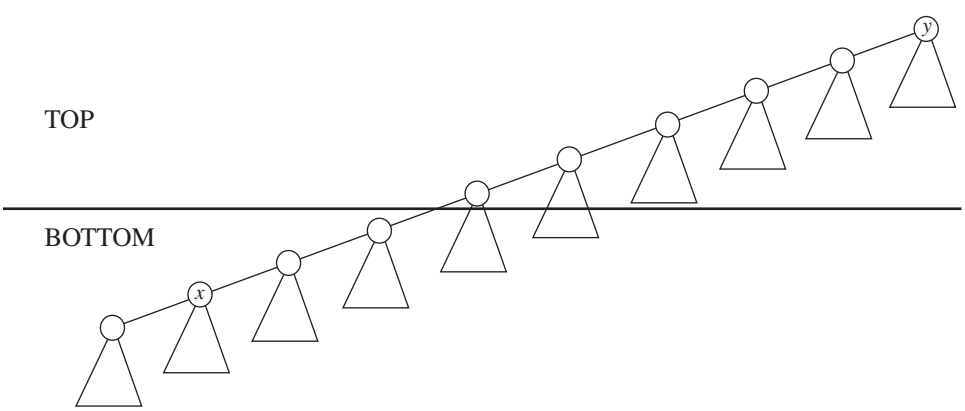


Figure 8.22 Recursive decomposition, case 3: Partial find goes from bottom to top

In Figure 8.21, the partial find resides entirely in the top half. Thus one partial find in the top half corresponds to one original partial find, and the charges can be recursively assigned to the top half.

However, we run into lots of trouble when we reach Figure 8.22. Here x is in the bottom half, and y is in the top half. The path compression would require that all nodes from x to y's child acquire y as its parent. For nodes in the top half, that is no problem, but for nodes in the bottom half this is a deal breaker: Any recursive charges to the bottom have to keep everything in the bottom. So as Figure 8.23 shows, we can perform the path compression on the top, but while some nodes in the bottom will need new parents, it is not clear what to do, because the new parents for those bottom nodes cannot be top nodes, and the new parents cannot be other bottom nodes.

The only option is to make a loop where these nodes' parents are themselves and make sure these parent changes are correctly charged in our accounting. Although this is a new algorithm because it can no longer be used to generate an identical tree, we don't need identical trees; we only need to be sure that each original partial find can be mapped into a new partial find operation and that the charges are identical. Figure 8.24 shows what the new tree will look like, and so the big remaining issue is the accounting.

Looking at Figure 8.24, we see that the path compression charges from x to y can be split into three parts. First, there is the path compression from z (the first top node on the upward path) to y. Clearly those charges are already accounted for recursively. Then there is the charge from the topmost-bottom node w to z. But that is only one unit, and there can be at most one of those per partial find operation. In fact, we can do a little better: There can be at most one of those per partial find operation on the top half. But how do we account for the parent changes on the path from x to w? One idea would be to argue that those changes would be exactly the same cost as if there were a partial find from x to w. But there is a big problem with that argument: It converts an original partial find into a partial find on the top *plus* a partial find on the bottom, which means the number of operations,

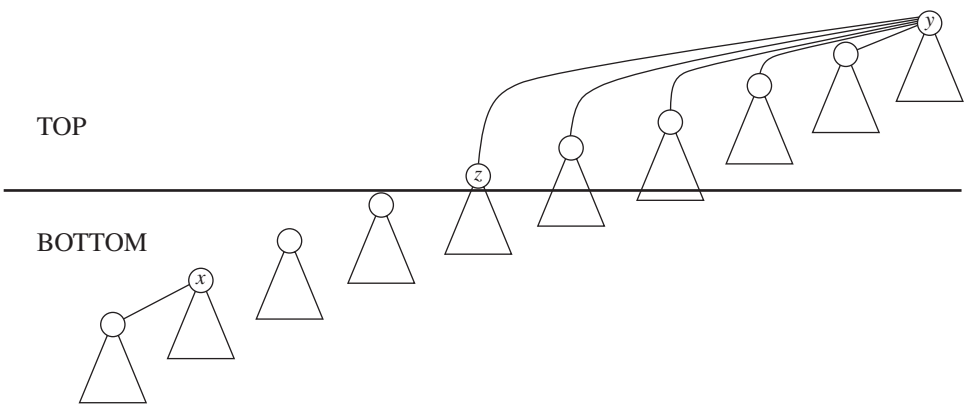


Figure 8.23 Recursive decomposition, case 3: Path compression can be performed on the top nodes, but the bottom nodes must get new parents; the parents cannot be top parents, and they cannot be other bottom nodes

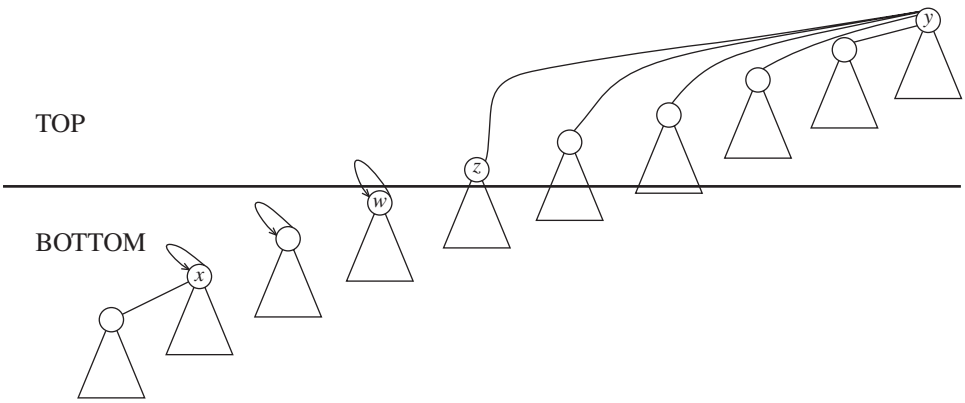


Figure 8.24 Recursive decomposition, case 3: The bottom node new parents are the nodes themselves

M, would no longer be the same. Fortunately, there is a simpler argument: Since each node on the bottom can have its parent set to itself only once, the number of charges are limited by the number of nodes on the bottom whose parents are also in the bottom (i.e., w is excluded).

There is one important detail that we must verify. Can we get in trouble on a subsequent partial find given that our reformulation detaches the nodes between x and w from the path to y? The answer is no. In the original partial find, suppose any of the nodes between x and w are involved in a subsequent original partial find. In that case, it will be with one of y's ancestors, and when that happens, any of those nodes will be the topmost "bottom node" in our reformulation. Thus on the subsequent partial find, the original partial find's parent change will have a corresponding one unit charge in our reformulation.

We can now proceed with the analysis. Let M be the total number of original partial find operations. Let M_t be the total number of partial find operations performed exclusively on the top half, and let M_b be the total number of partial find operations performed exclusively on the bottom half. Let N be the total number of nodes. Let N_t be the total number of top-half nodes, let N_b be the total number of bottom-half nodes, and let N_{nrb} be the total number of non-root bottom nodes (i.e., the number of bottom nodes whose parents are also bottom nodes prior to any partial finds).

Lemma 8.3
$M = M_t + M_b$.

Proof
In cases 1 and 3, each original partial find operation is replaced by a partial find on the top half, and in case 2, it is replaced by a partial find on the bottom half. Thus each partial find is replaced by exactly one partial find operation on one of the halves.

Our basic idea is that we are going to partition the nodes so that all nodes with rank s or lower are in the bottom, and the remaining nodes are in the top. The choice of s will be made later in the proof. The next lemma shows that we can provide a recursive formula

for the number of parent changes by splitting the charges into the top and bottom groups. One of the key ideas is that a recursive formula is written not only in terms of M and N, which would be obvious, but also in terms of the maximum rank in the group.

Lemma 8.4

Let $C(M, N, r)$ be the number of parent changes for a sequence of M finds with path compression on N items whose maximum rank is r. Suppose we partition so that all nodes with rank at s or lower are in the bottom, and the remaining nodes are in the top. Assuming appropriate initial conditions,

$$C(M, N, r) < C(M_t, N_t, r) + C(M_b, N_b, s) + M_t + N_{nrb}$$

Proof

The path compression that is performed in each of the three cases is covered by $C(M_t, N_t, r) + C(M_b, N_b, s)$. Node w in case 3 is accounted for by M_t. Finally, all the other bottom nodes on the path are non-root nodes that can have their parent set to themselves at most once in the entire sequence of compressions. They are accounted for by N_{nrb}.

If union-by-rank is used, then by Lemma 8.1, every top node has children of ranks $0, 1, \ldots, s$ prior to the commencement of the partial find operations. Each of those children are definitely root nodes in the bottom (their parent is a top node). So for each top node, $s+2$ nodes (the $s+1$ children plus the top node itself) are definitely not included in N_{nrb}. Thus, we can refomulate Lemma 8.4 as follows:

Lemma 8.5

Let $C(M, N, r)$ be the number of parent changes for a sequence of M finds with path compression on N items whose maximum rank is r. Suppose we partition so that all nodes with rank at s or lower are in the bottom, and the remaining nodes are in the top. Assuming appropriate initial conditions,

$$C(M, N, r) < C(M_t, N_t, r) + C(M_b, N_b, s) + M_t + N - (s+2)N_t$$

Proof

Substitute $N_{nrb} < N - (s+2)N_t$ into Lemma 8.4.

If we look at Lemma 8.5, we see that $C(M, N, r)$ is recursively defined in terms of two smaller instances. Our basic goal at this point is to remove one of these instances by providing a bound for it. What we would like to do is to remove $C(M_t, N_t, r)$. Why? Because, if we do so, what is left is $C(M_b, N_b, s)$. In that case, we have a recursive formula in which r is reduced to s. If s is small enough, we can make use of a variation of Equation (8.1), namely, that the solution to

$$T(N) = \begin{cases} 0 & N \leq 1 \\ T(\lfloor f(N) \rfloor) + M & N > 1 \end{cases} \tag{8.2}$$

is $O(Mf^*(N))$. So, let's start with a simple bound for $C(M, N, r)$:

Theorem 8.1
$C(M, N, r) < M + N \log r$.

Proof
We start with Lemma 8.5:

$$C(M, N, r) < C(M_t, N_t, r) + C(M_b, N_b, s) + M_t + N - (s + 2)N_t \tag{8.3}$$

Observe that in the top half, there are only nodes of rank $s+1, s+2, \ldots, r$, and thus no node can have its parent change more than $(r-s-2)$ times. This yields a trivial bound of $N_t(r-s-2)$ for $C(M_t, N_t, r)$. Thus,

$$C(M, N, r) < N_t(r - s - 2) + C(M_b, N_b, s) + M_t + N - (s + 2)N_t \tag{8.4}$$

Combining terms,

$$C(M, N, r) < N_t(r - 2s - 4) + C(M_b, N_b, s) + M_t + N \tag{8.5}$$

Select $s = \lfloor r/2 \rfloor$. Then $r - 2s - 4 < 0$, so

$$C(M, N, r) < C(M_b, N_b, \lfloor r/2 \rfloor) + M_t + N \tag{8.6}$$

Equivalently, since according to Lemma 8.3, $M = M_b + M_t$ (the proof falls apart without this),

$$C(M, N, r) - M < C(M_b, N_b, \lfloor r/2 \rfloor) - M_b + N \tag{8.7}$$

Let $D(M, N, r) = C(M, N, r) - M$; then

$$D(M, N, r) < D(M_b, N_b, \lfloor r/2 \rfloor) + N \tag{8.8}$$

which implies $D(M, N, r) < N \log r$. This yields $C(M, N, r) < M + N \log r$.

Theorem 8.2
Any sequence of $N - 1$ unions and M finds with path compression makes at most $M + N \log \log N$ parent changes during the finds.

Proof
The bound is immediate from Theorem 8.1 since $r \leq \log N$.

8.6.3 An $O(M \log^* N)$ Bound

The bound in Theorem 8.2 is pretty good, but with a little work, we can do even better. Recall, that a central idea of the recursive decomposition is choosing s to be as small as possible. But to do this, the other terms must also be small, and as s gets smaller, we would expect $C(M_t, N_t, r)$ to get larger. But the bound for $C(M_t, N_t, r)$ used a primitive estimate, and Theorem 8.1 itself can now be used to give a better estimate for this term. Since the $C(M_t, N_t, r)$ estimate will now be lower, we will be able to use a lower s.

Theorem 8.3
$C(M, N, r) < 2M + N \log^* r$.

Proof

From Lemma 8.5 we have,

$$C(M, N, r) < C(M_t, N_t, r) + C(M_b, N_b, s) + M_t + N - (s + 2)N_t \tag{8.9}$$

and by Theorem 8.1, $C(M_t, N_t, r) < M_t + N_t \log r$. Thus,

$$C(M, N, r) < M_t + N_t \log r + C(M_b, N_b, s) + M_t + N - (s + 2)N_t \tag{8.10}$$

Rearranging and combining terms yields

$$C(M, N, r) < C(M_b, N_b, s) + 2M_t + N - (s - \log r + 2)N_t \tag{8.11}$$

So choose $s = \lfloor \log r \rfloor$. Clearly, this choice implies that $(s - \log r + 2) > 0$, and thus we obtain

$$C(M, N, r) < C(M_b, N_b, \lfloor \log r \rfloor) + 2M_t + N \tag{8.12}$$

Rearranging as in Theorem 8.1, we obtain

$$C(M, N, r) - 2M < C(M_b, N_b, \lfloor \log r \rfloor) - 2M_b + N \tag{8.13}$$

This time, let $D(M, N, r) = C(M, N, r) - 2M$; then

$$D(M, N, r) < D(M_b, N_b, \lfloor \log r \rfloor) + N \tag{8.14}$$

which implies $D(M, N, r) < N \log^* r$. This yields $C(M, N, r) < 2M + N \log^* r$.

8.6.4 An $O(M \alpha(M, N))$ Bound

Not surprisingly, we can now use Theorem 8.3 to improve Theorem 8.3:

Theorem 8.4

$C(M, N, r) < 3M + N \log^{**} r$.

Proof

Following the steps in the proof of Theorem 8.3, we have

$$C(M, N, r) < C(M_t, N_t, r) + C(M_b, N_b, s) + M_t + N - (s + 2)N_t \tag{8.15}$$

and by Theorem 8.3, $C(M_t, N_t, r) < 2M_t + N_t \log^* r$. Thus,

$$C(M, N, r) < 2M_t + N_t \log^* r + C(M_b, N_b, s) + M_t + N - (s + 2)N_t \tag{8.16}$$

Rearranging and combining terms yields

$$C(M, N, r) < C(M_b, N_b, s) + 3M_t + N - (s - \log^* r + 2)N_t \tag{8.17}$$

So choose $s = \log^* r$ to obtain

$$C(M, N, r) < C(M_b, N_b, \log^* r) + 3M_t + N \tag{8.18}$$

Rearranging as in Theorems 8.1 and 8.3, we obtain

$$C(M,N,r) - 3M < C(M_b, N_b, \log^* r) - 3M_b + N \tag{8.19}$$

This time, let $D(M,N,r) = C(M,N,r) - 3M$; then

$$D(M,N,r) < D(M_b, N_b, \log^* r) + N \tag{8.20}$$

which implies $D(M,N,r) < N \log^{**} r$. This yields $C(M,N,r) < 3M + N \log^{**} r$.

Needless to say, we could continue this ad infinitim. Thus with a bit of math, we get a progression of bounds:

$$\begin{aligned}
C(M,N,r) &< 2M + N \log^* r \\
C(M,N,r) &< 3M + N \log^{**} r \\
C(M,N,r) &< 4M + N \log^{***} r \\
C(M,N,r) &< 5M + N \log^{****} r \\
C(M,N,r) &< 6M + N \log^{*****} r
\end{aligned}$$

Each of these bounds would seem to be better than the previous since, after all, the more *s the slower $\log^{**\cdots**} r$ grows. However, this ignores the fact that while $\log^{*****} r$ is smaller than $\log^{****} r$, the $6M$ term is NOT smaller than the $5M$ term.

Thus what we would like to do is to optimize the number of *s that are used.

Define $\alpha(M,N)$ to represent the optimal number of *s that will be used. Specifically,

$$\alpha(M,N) = min \left\{ i \geq 1 \,\middle|\, \log^{\overbrace{****}^{i \text{ times}}} (\log N) \leq (M/N) \right\}$$

Then, the running time of the union/find algorithm can be bounded by $O(M\alpha(M,N))$.

Theorem 8.5

Any sequence of $N-1$ unions and M finds with path compression makes at most

$$(i+1)M + N \log^{\overbrace{****}^{i \text{ times}}} (\log N)$$

parent changes during the finds.

Proof

This follows from the above discussion and the fact that $r \leq \log N$.

Theorem 8.6

Any sequence of $N-1$ unions and M finds with path compression makes at most $M\alpha(M,N) + 2M$ parent changes during the finds.

Proof

In Theorem 8.5, choose i to be $\alpha(M,N)$; thus, we obtain a bound of $(i+1)M + N(M/N)$, or $M\alpha(M,N) + 2M$.

8.7 An Application

An example of the use of the union/find data structure is the generation of mazes, such as the one shown in Figure 8.25. In Figure 8.25, the starting point is the top-left corner, and the ending point is the bottom-right corner. We can view the maze as a 50-by-88 rectangle of cells in which the top-left cell is connected to the bottom-right cell, and cells are separated from their neighboring cells via walls.

A simple algorithm to generate the maze is to start with walls everywhere (except for the entrance and exit). We then continually choose a wall randomly, and knock it down if the cells that the wall separates are not already connected to each other. If we repeat this process until the starting and ending cells are connected, then we have a maze. It is actually better to continue knocking down walls until every cell is reachable from every other cell (this generates more false leads in the maze).

We illustrate the algorithm with a 5-by-5 maze. Figure 8.26 shows the initial configuration. We use the union/find data structure to represent sets of cells that are connected to each other. Initially, walls are everywhere, and each cell is in its own equivalence class.

Figure 8.27 shows a later stage of the algorithm, after a few walls have been knocked down. Suppose, at this stage, the wall that connects cells 8 and 13 is randomly targeted. Because 8 and 13 are already connected (they are in the same set), we would not remove the wall, as it would simply trivialize the maze. Suppose that cells 18 and 13 are randomly targeted next. By performing two `find` operations, we see that these are in different sets; thus 18 and 13 are not already connected. Therefore, we knock down the wall that separates them, as shown in Figure 8.28. Notice that as a result of this operation, the sets

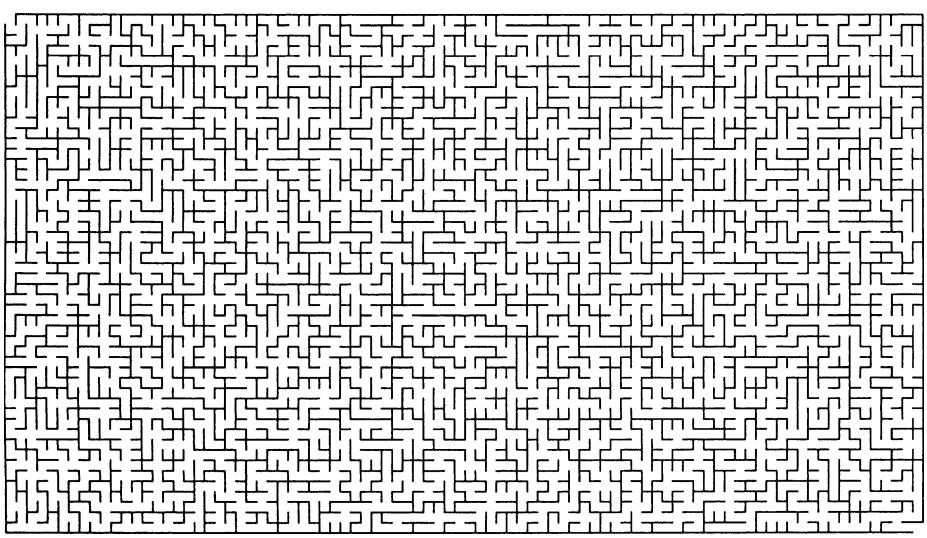
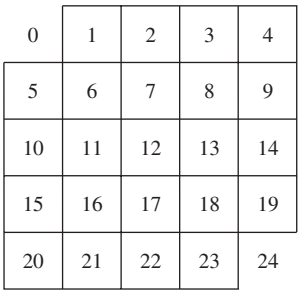


Figure 8.25 A 50-by-88 maze

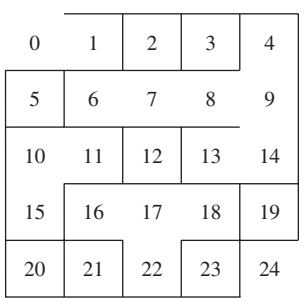


{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21} {22} {23} {24}

Figure 8.26 Initial state: all walls up, all cells in their own set

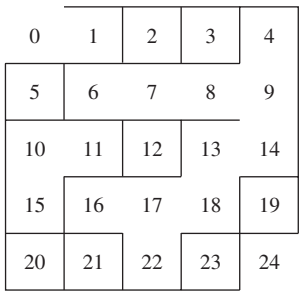
containing 18 and 13 are combined via a `union` operation. This is because everything that was connected to 18 is now connected to everything that was connected to 13. At the end of the algorithm, depicted in Figure 8.29, everything is connected, and we are done.

The running time of the algorithm is dominated by the union/find costs. The size of the union/find universe is equal to the number of cells. The number of `find` operations is proportional to the number of cells, since the number of removed walls is one less than the number of cells, while with care, we see that there are only about twice the number of walls as cells in the first place. Thus, if N is the number of cells, since there are two finds per randomly targeted wall, this gives an estimate of between (roughly) $2N$ and $4N$ `find` operations throughout the algorithm. Therefore, the algorithm's running time can be taken as $O(N \log^* N)$, and this algorithm quickly generates a maze.



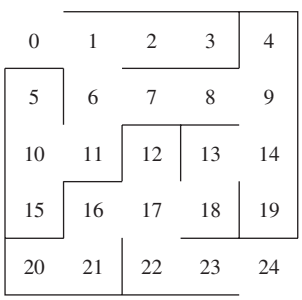
{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14} {5} {10, 11, 15} {12} {16, 17, 18, 22} {19} {20} {21} {23} {24}

Figure 8.27 At some point in the algorithm: Several walls down, sets have merged; if at this point the wall between 8 and 13 is randomly selected, this wall is not knocked down, because 8 and 13 are already connected



{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {5} {10, 11, 15} {12} {19} {20} {21} {23} {24}

Figure 8.28 Wall between squares 18 and 13 is randomly selected in Figure 8.27; this wall is knocked down, because 18 and 13 are not already connected; their sets are merged



{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}

Figure 8.29 Eventually, 24 walls are knocked down; all elements are in the same set

Summary

We have seen a very simple data structure to maintain disjoint sets. When the `union` operation is performed, it does not matter, as far as correctness is concerned, which set retains its name. A valuable lesson that should be learned here is that it can be very important to consider the alternatives when a particular step is not totally specified. The `union` step is flexible; by taking advantage of this, we are able to get a much more efficient algorithm.

Path compression is one of the earliest forms of *self-adjustment,* which we have seen elsewhere (splay trees, skew heaps). Its use is extremely interesting, especially from a theoretical point of view, because it was one of the first examples of a simple algorithm with a not-so-simple worst-case analysis.

Exercises

8.1 Show the result of the following sequence of instructions: `union(1,2)`, `union(3,4)`, `union(3,5)`, `union(1,7)`, `union(3,6)`, `union(8,9)`, `union(1,8)`, `union(3,10)`, `union (3,11)`, `union(3,12)`, `union(3,13)`, `union(14,15)`, `union(16,0)`, `union(14,16)`, `union (1,3)`, `union(1, 14)` when the `union`s are
a. performed arbitrarily
b. performed by height
c. performed by size

8.2 For each of the trees in the previous exercise, perform a `find` with path compression on the deepest node.

8.3 Write a program to determine the effects of path compression and the various `union`ing strategies. Your program should process a long sequence of equivalence operations using all six of the possible strategies.

8.4 Show that if `union`s are performed by height, then the depth of any tree is $O(\log N)$.

8.5 Suppose $f(N)$ is a nicely defined function that reduces N to a smaller integer. What is the solution to the recurrence $T(N) = \frac{N}{f(N)} T(f(N)) + N$ with appropriate initial conditions?

8.6 a. Show that if $M = N^2$, then the running time of M union/find operations is $O(M)$.
b. Show that if $M = N \log N$, then the running time of M union/find operations is $O(M)$.
⋆ c. Suppose $M = \Theta(N \log \log N)$. What is the running time of M union/find operations?
⋆ d. Suppose $M = \Theta(N \log^* N)$. What is the running time of M union/find operations?

8.7 Tarjan's original bound for the union/find algorithm defined $\alpha(M, N) = min\{i \geq 1 | (A(i, \lfloor M/N \rfloor) > \log N)\}$, where

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

Here, $A(m, n)$ is one version of the **Ackermann function**. Are the two definitions of α asymptotically equivalent?

8.8 Prove that for the mazes generated by the algorithm in Section 8.7, the path from the starting to ending points is unique.

8.9 Design an algorithm that generates a maze that contains no path from start to finish but has the property that the removal of a **prespecified** wall creates a unique path.

⋆ **8.10** Suppose we want to add an extra operation, `deunion`, which undoes the last `union` operation that has not been already undone.
a. Show that if we do union-by-height and `find`s without path compression, then `deunion` is easy, and a sequence of M `union`, `find`, and `deunion` operations takes $O(M \log N)$ time.
b. Why does path compression make `deunion` hard?

** c. Show how to implement all three operations so that the sequence of M operations takes $O(M \log N / \log \log N)$ time.

* **8.11** Suppose we want to add an extra operation, `remove(x)`, which removes `x` from its current set and places it in its own. Show how to modify the union/find algorithm so that the running time of a sequence of M `union`, `find`, and `remove` operations is $O(M\alpha(M, N))$.

* **8.12** Show that if all of the `union`s precede the `find`s, then the disjoint sets algorithm with path compression requires linear time, even if the `union`s are done arbitrarily.

** **8.13** Prove that if `union`s are done arbitrarily, but path compression is performed on the `find`s, then the worst-case running time is $\Theta(M \log N)$.

* **8.14** Prove that if `union`s are done by size and path compression is performed, the worst-case running time is $O(M\alpha(M, N))$.

8.15 The disjoint set analysis in Section 8.6 can be refined to provide tight bounds for small N.

a. Show that $C(M, N, 0)$ and $C(M, N, 1)$ are both 0.
b. Show that $C(M, N, 2)$ is at most M.
c. Let $r \leq 8$. Choose $s = 2$ and show that $C(M, N, r)$ is at most $M + N$.

8.16 Suppose we implement partial path compression on `find(i)` by making every other node on the path from i to the root link to its grandparent (where this makes sense). This is known as *path halving*.

a. Write a procedure to do this.
b. Prove that if path halving is performed on the `find`s and either union-by-height or union-by-size is used, the worst-case running time is $O(M\alpha(M, N))$.

8.17 Write a program that generates mazes of arbitrary size. If you are using a system with a windowing package, generate a maze similar to that in Figure 8.25. Otherwise describe a textual representation of the maze (for instance, each line of output represents a square and has information about which walls are present) and have your program generate a representation.

References

Various solutions to the union/find problem can be found in [6], [9], and [11]. Hopcroft and Ullman showed an $O(M \log^* N)$ bound using a nonrecursive decomposition. Tarjan [16] obtained the bound $O(M\alpha(M, N))$, where $\alpha(M, N)$ is as defined in Exercise 8.7. A more precise (but asymptotically identical) bound for $M < N$ appears in [2] and [19]. The analysis in Section 8.6 is due to Seidel and Sharir [15]. Various other strategies for path compression and `union`s also achieve the same bound; see [19] for details.

A lower bound showing that under certain restrictions $\Omega(M\alpha(M, N))$ time is required to process M union/find operations was given by Tarjan [17]. Identical bounds under less restrictive conditions have been shown in [7] and [14].

Applications of the union/find data structure appear in [1] and [10]. Certain special cases of the union/find problem can be solved in $O(M)$ time [8]. This reduces the running time of several algorithms, such as [1], graph dominance, and reducibility (see references

in Chapter 9) by a factor of $\alpha(M, N)$. Others, such as [10] and the graph connectivity problem in this chapter, are unaffected. The paper lists 10 examples. Tarjan has used path compression to obtain efficient algorithms for several graph problems [18].

Average-case results for the union/find problem appear in [5], [12], [22], and [3]. Results bounding the running time of any single operation (as opposed to the entire sequence) appear in [4] and [13].

Exercise 8.10 is solved in [21]. A general union/find structure, supporting more operations, is given in [20].

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "On Finding Lowest Common Ancestors in Trees," *SIAM Journal on Computing,* 5 (1976), 115–132.
2. L. Banachowski, "A Complement to Tarjan's Result about the Lower Bound on the Complexity of the Set Union Problem," *Information Processing Letters,* 11 (1980), 59–65.
3. B. Bollobás and I. Simon, "Probabilistic Analysis of Disjoint Set Union Algorithms," *SIAM Journal on Computing,* 22 (1993), 1053–1086.
4. N. Blum, "On the Single-Operation Worst-Case Time Complexity of the Disjoint Set Union Problem," *SIAM Journal on Computing,* 15 (1986), 1021–1024.
5. J. Doyle and R. L. Rivest, "Linear Expected Time of a Simple Union Find Algorithm," *Information Processing Letters,* 5 (1976), 146–148.
6. M. J. Fischer, "Efficiency of Equivalence Algorithms," in *Complexity of Computer Computation* (eds. R. E. Miller and J. W. Thatcher), Plenum Press, New York, 1972, 153–168.
7. M. L. Fredman and M. E. Saks, "The Cell Probe Complexity of Dynamic Data Structures," *Proceedings of the Twenty-first Annual Symposium on Theory of Computing* (1989), 345–354.
8. H. N. Gabow and R. E. Tarjan, "A Linear-Time Algorithm for a Special Case of Disjoint Set Union," *Journal of Computer and System Sciences,* 30 (1985), 209–221.
9. B. A. Galler and M. J. Fischer, "An Improved Equivalence Algorithm," *Communications of the ACM,* 7 (1964), 301–303.
10. J. E. Hopcroft and R. M. Karp, "An Algorithm for Testing the Equivalence of Finite Automata," *Technical Report TR-71-114,* Department of Computer Science, Cornell University, Ithaca, N.Y., 1971.
11. J. E. Hopcroft and J. D. Ullman, "Set Merging Algorithms," *SIAM Journal on Computing,* 2 (1973), 294–303.
12. D. E. Knuth and A. Schonhage, "The Expected Linearity of a Simple Equivalence Algorithm," *Theoretical Computer Science,* 6 (1978), 281–315.
13. J. A. LaPoutre, "New Techniques for the Union-Find Problem," *Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms* (1990), 54–63.
14. J. A. LaPoutre, "Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines," *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing* (1990), 34–44.
15. R. Seidel and M. Sharir, "Top-Down Analysis of Path Compression," *SIAM Journal on Computing*, 34 (2005), 515–525.
16. R. E. Tarjan, "Efficiency of a Good but Not Linear Set Union Algorithm," *Journal of the ACM,* 22 (1975), 215–225.
17. R. E. Tarjan, "A Class of Algorithms Which Require Nonlinear Time to Maintain Disjoint Sets," *Journal of Computer and System Sciences,* 18 (1979), 110–127.

18. R. E. Tarjan, "Applications of Path Compression on Balanced Trees," *Journal of the ACM,* 26 (1979), 690–715.
19. R. E. Tarjan and J. van Leeuwen, "Worst-Case Analysis of Set Union Algorithms," *Journal of the ACM,* 31 (1984), 245–281.
20. M. J. van Kreveld and M. H. Overmars, "Union-Copy Structures and Dynamic Segment Trees," *Journal of the ACM,* 40 (1993), 635–652.
21. J. Westbrook and R. E. Tarjan, "Amortized Analysis of Algorithms for Set Union with Backtracking," *SIAM Journal on Computing,* 18 (1989), 1–11.
22. A. C. Yao, "On the Average Behavior of Set Merging Algorithms," *Proceedings of Eighth Annual ACM Symposium on the Theory of Computation* (1976), 192–195.