Chapter 7 Union-Find Structures

Merging galaxies, NGC 2207 and IC 2163. Combined image from NASA's Spitzer Space Telescope and Hubble Space Telescope. 2006. U.S. government image. NASA/JPL-Caltech/STSci/Vassar.

Contents

Social networking research studies how relationships between various people can influence behavior. Thus, a critical part of this research involves understanding how such relationships define groups or communities within a social network. Formally, given a set, S, of people, we can define a ***social network*** for S by creating an element, x, for each person and then create a set, E, of ***edges*** or ***ties*** between pairs of people that have a certain kind of relationship. For example, in a friendship network, ties would be defined by pairs of friends, in a sexual-relations network, ties would be defined by pairs of people who were physically intimate, and in an enemies network, ties would be defined by pairs of enemies. For the sake of being concrete, let us consider a friendship network.

Social networking researchers are often interested in identifying ***connected components*** in a friendship network. A connected component is a subset, T, of people from S that satisfies the following properties:

- Every person in T is related through friendship, that is, for any x and y in T, either x and y are friends or there is a chain of friendship, such as through a friend of a friend of a friend, that connects x and y.
- No one in T is friends with anyone outside of T.

Such community identifications are useful, since they allow researchers to study the degree to which being connected through friendship can influence behavior, such as how people vote or shop or whether they are more likely to gain weight or lose weight. Thus, from an algorithmic perspective, it is useful to have an efficient data structure that can be used to identify the connected components in a social network. The types of data structures we discuss in this chapter, ***union-find structures***, fit this bill perfectly.

In the context of a social network, a union-find structure gives us a way of maintaining a collection of disjoint sets of people, so as to support the following three operations:

- Make a set, which initially contains just a single person, x, and has that person's name, "X," as the name of the set
- Union two sets, A and B, together, naming the result as being one of "A" or "B," so that everyone that was in A or B is now identified as belonging to this union
- Find the name of the set containing a particular person, x.

Given such a data structure, we can determine all the connected components in a social network simply by making a set for each person in the network and doing a union operation for every pair of friends, x and y, that belong to different sets at the point we consider their edge, (x, y). As we study in this chapter, this algorithm has a surprising "almost" linear-time behavior.

7.1 Union-Find and Its Applications

A ***partition*** or ***union-find*** structure is a data structure supporting a collection of disjoint sets. We define the methods for this structure assuming we have a constant-time way to access a node associated with an item, e. For instance, items could themselves be nodes or we could maintain some kind of lookup table or map for finding the node associated with an item, e, in constant time. Given such an ability, the methods include the following:

makeSet(e): Create a singleton set containing the element e and name this set "e".

union(A, B): Update A and B to create $A \cup B$, naming the result as "A" or "B".

find(e): Return the name of the set containing the element e.

We refer to an implementation supporting these methods as a ***union-find structure***.

7.1.1 Connected Components

Consider again the connected components problem mentioned above. That is, suppose we are given a social network, N, defined by a set, S, of people, and a set, E, of edges defining relationships between pairs of people, in no particular order, and we are asked to find all the connected components for N. (See Figure 7.1.)

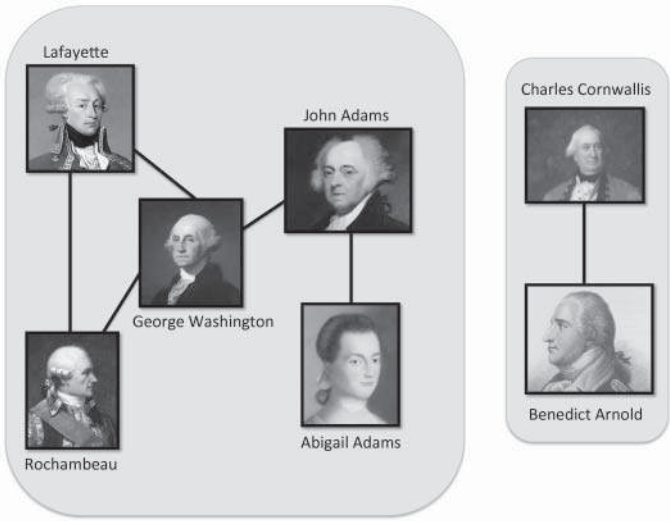


Figure 7.1: Connected components in a friendship network of some of the key figures in the American Revolutionary War. All images are in the public domain.

A Connected Components Algorithm

We give a pseudocode description of an algorithm for solving the connected components problem using a union-find structure in Algorithm 7.2. The output from this algorithm is an identification, for each person x in S, of the connected component to which x belongs. For instance, for the social network shown in Figure 7.1, the output would identify Lafayette, Rochambeau, George Washington, John Adams, and Abigail Adams as belonging to the "Washington" connected component, and Charles Cornwallis and Benedict Arnold as belonging to the "Cornwallis" connected component.

Algorithm UFConnectedComponents(S, E):
Input: A set, S, of n people and a set, E, of m pairs of people from S defining pairwise relationships
Output: An identification, for each x in S, of the connected component containing x

```
for each x in S do
    makeSet(x)
for each (x, y) in E do
    if find(x) ≠ find(y) then
        union(find(x), find(y))
for each x in S do
    Output "Person x belongs to connected component" find(x)
```

Algorithm 7.2: A connected components algorithm using union and find.

This algorithm's efficiency depends on how we implement the union-find structure, of course. If performing a sequence of m union and find operations, starting with n singleton sets created with the makeSet method, takes $O(t(n, m))$ time, then the running time of the UFConnectedComponents algorithm can be characterized as being $O(t(n, n + m))$, since we do two find operations for each edge in E and then one find operation for each of the n members of S. Using the implementations described in this chapter, we can achieve a running time that is either $O((n+m) \log n)$, using a list-based implementation, or "almost" $O(n+m)$, using a tree-based implementation.

As we explore in Chapter 13, if we are given the set E in sorted order (say, according to a lexicographic ordering), then we can actually design a connected components algorithm that runs in $O(n+m)$ time. But the UFConnectedComponents algorithm is the fastest way we know of for solving the connected components problem if we cannot make any assumptions about the ordering of E and we only get to scan through it once.

7.1.2 Maze Construction and Percolation Theory

Another application of union-find structures, which might at first seem to be purely for entertainment purposes, is for constructing mazes. In this application, we consider a ***maze*** to be a two-dimensional visual puzzle, defined by cells, which can be traversed, and walls, which are barriers. The goal is to find a path from a start location to a finish location that traverses connected cells in the maze without crossing any walls. (See Figure 7.3.)

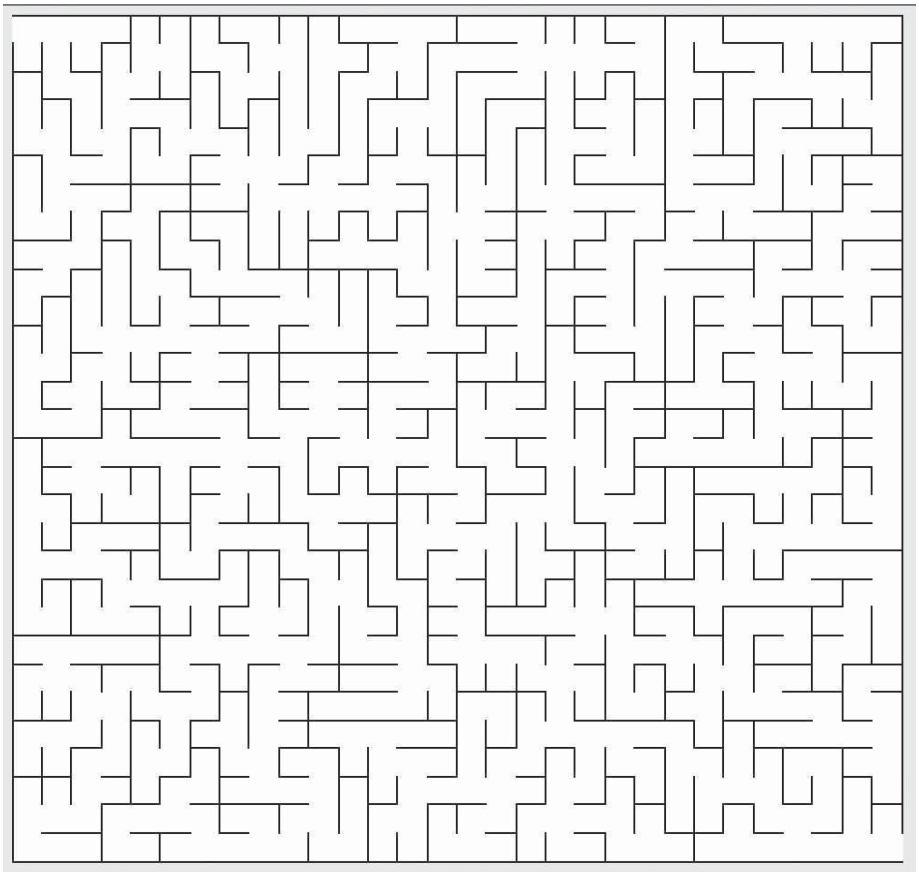


Figure 7.3: A maze defined from a 30×30 square grid.

One challenge for people who like mazes is to find interesting mazes to solve. Thus, it is useful to have an automated way to construct mazes that have exactly one solution such that finding that solution is nontrivial. There are several possibilities for constructing such mazes, of course, but one that has turned out to be able to consistently construct interesting, solvable mazes is the method given in Algorithm 7.4.

Algorithm MazeGenerator(G, E):

Input: A grid, G, consisting of n cells and a set, E, of m "walls," each of which divides two cells, x and y, such that the walls in E initially separate and isolate all the cells in G

Output: A subset, R of E, such that removing the edges in R from E creates a maze defined on G by the remaining walls

while R has fewer than $n - 1$ edges **do**
  Choose an edge, (x, y), in E uniformly at random from among those previously unchosen
  if find$(x) \neq$ find(y) **then**
    union(find(x), find(y))
    Add the edge (x, y) to R
return R

Algorithm 7.4: A randomized algorithm for constructing mazes.

Given a grid, G, of n cells, and a set, E, of m walls separating pairs of cells in G, this maze construction algorithm considers the walls in E in random order. For each wall in this ordering that joins two previously separated connected components of cells, we add that wall to the set we are going to remove and we union the two connected components it separated. Since we repeat this step until the set of removed walls is equal to $n - 1$, we guarantee that the final maze has a single connected component. Thus, it has a solution and it is likely to have lots of passages where a puzzle solver will reach a dead end. Indeed, the maze in Figure 7.3 was constructed using this algorithm.

Given a random ordering of the edges in E, the running time of this algorithm is $O(t(n, m))$, where $O(t(n, m))$ is the running time for performing m union and find operations on an initial set of n singleton sets.

Although this problem is motivated by a recreation, it is related to the science of ***percolation theory***, which is the study of how liquids permeate porous materials. For instance, a porous material might be modeled as a three-dimensional $n \times n \times n$ grid of cells. The barriers separating adjacent pairs of cells might then be removed virtually with some probability p and remain with probability $1 - p$. The scientific question to answer for this virtual material would then be whether a liquid poured on top of the grid will make it to the bottom—that is, whether any top cell is connected to some bottom cell. Such a question can be answered by connecting every top cell to a special "super-top" cell and connecting every bottom cell to a special "super-bottom" cell, performing a union operation for every pair of connected cells, and then asking if the super-top and super-bottom cell are in the same set. Repeating this experiment for several instances of the three-dimensional grid and for different values of p can then provide insights into how liquids permeate porous materials. Thus, the same thinking behind maze construction can be used to answer scientific questions, and both can be performed using union-find structures.

7.2 A List-Based Implementation

A simple implementation of a union-find structure is to use a collection of linked lists, one for each set, where the list for a set A contains a head node, which stores

- the size of A
- the name of A
- a pointer to the first and last nodes of a linked list containing pointers to all the elements of A.

Each node of the linked list for A stores a pointer to an element belonging to that set and a pointer to the head node for A. (See Figure 7.5.)

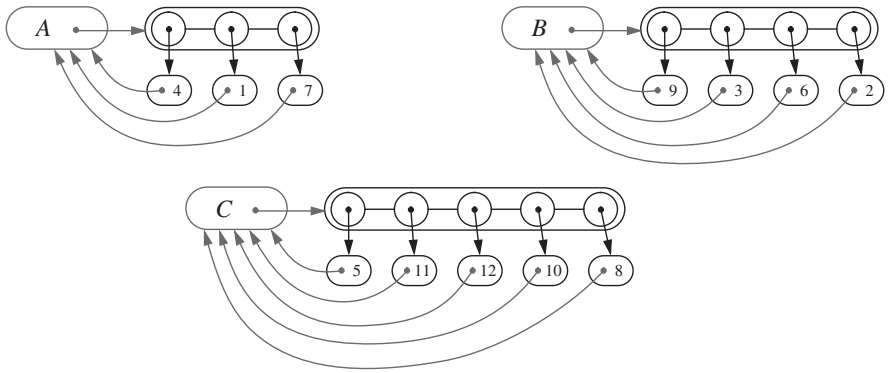


Figure 7.5: A list-based implementation of a union-find structure consisting of three sets: $A = \{1, 4, 7\}$, $B = \{2, 3, 6, 9\}$, and $C = \{5, 8, 10, 11, 12\}$.

Thus, we can perform operation find(e) in $O(1)$ time simply by following the pointer from the node for e to the head node and returning the name of the set identified by that node. Likewise, makeSet also takes $O(1)$ time, since it involves the creation of a new head node and a linked list containing a single element. The operation union(A, B) is not a simple constant-time procedure, however, since it requires that we join the two linked lists for A and B into one list and update the head pointers for all the nodes in one of these two lists (to now point to the head node of the other list). In order to save some time, let us choose to implement this operation by always changing the head pointers for whichever list, for A or B, has the smaller size (breaking ties arbitrarily). Hence, this implementation of the operation union(A, B) takes time $O(\min(|A|, |B|))$, which is $O(n)$ in the worst-case, because, in the worst case, $|A| = |B| = n/2$, where n is the number of singleton sets we start with. Nevertheless, as we show below, an amortized analysis reveals that this implementation is much better than first appears from this worst-case analysis.

Details for a List-Based Implementation for Disjoint Sets

We give pseudocode descriptions for implementing the fundamental methods of the disjoint sets data structures using linked lists in Algorithm 7.6.

Algorithm makeSet():

for each singleton element, x **do**
create a linked-list header node, u,
u.name $\leftarrow$ "x"
add x to the list u
x.head $\leftarrow u$

Algorithm find(x):

return x.head

Algorithm union(u, v):

if the set u is smaller than v **then**
for each element, x, in the set u **do**
remove x from u and add it v
x.head $\leftarrow v$
else
for each element, x, in the set v **do**
remove x from v and add it u
x.head $\leftarrow u$

Algorithm 7.6: The fundamental disjoint-set methods for an implementation based on linked lists.

Analysis of the List-Based Implementation

The above linked list implementation is simple, and it might at first appear to be inefficient, but, as the following theorem shows, this implementation is actually not that slow.

Theorem 7.1: *Performing a sequence, σ, of m* union *and* find *operations, starting from n singleton sets, using the above list-based implementation of a union-find structure, takes $O(n \log n + m)$ time.*

Proof: We use the accounting method to analyze the time for us to perform all the operations in σ. We assume that 1 cyber-dollar can pay for the constant amount

of time it takes to perform a find operation, or to update a head pointer during a union operation.

In the case of a find operation, we charge the operation itself 1 cyber-dollar, and, in the case of a union operation, we charge 1 cyber-dollar to each node for which we change its head pointer. Note that we charge nothing to the union operations themselves. Also, observe that there can be at most $n - 1$ union operations before all the singleton sets have been merged into one. Finally, note that the total charges to find operations can be at most $O(m)$, since m is the number of operations in σ.

Consider, then, the number of charges made to nodes on behalf of union operations. The important observation is that each time we update the head pointer for some node, the size of the new set at least doubles. This property is due to the fact that we always link the smaller set into the larger one, with ties broken arbitrarily. Thus, any node can have its head pointer changed at most $\log n$ times; hence, each such node can be charged at most $O(\log n)$ cyber-dollars, since we assume that the partition starts with n singleton sets. Thus, the total amount of cyber-dollars charged to all the nodes in this implementation of a union-find structure is $O(n \log n)$. ■

The amortized running time of an operation in a series of makeSet, union, and find operations, is the total time taken for the series divided by the number of operations. Note, in addition, that, for the sake of analysis of a sequence of union, find, and makeSet operations, we can assume without loss of generality that all the makeSet operations come first. We conclude from the above theorem that, for a list-based implementation of a union-find structure, as described above, the amortized running time of each union operation is $O(\log n)$ and the amortized running time for each makeSet and find operation is $O(1)$. Thus, we can summarize the performance of our simple list-based implementation as follows.

Theorem 7.2: *Using a list-based implementation of a union-find structure, in a series of* makeSet, union, *and* find *operations, involving a total of n initially singleton sets, the amortized running of each* union *operation is $O(\log n)$ and the amortized running time for each* makeSet *and* find *operation is $O(1)$.*

We would like to stress that in this list-based implementation of a union-find structure, the running time of the union operations is the computational bottleneck, since the running time of this method is proportional to the size of the smaller set involved in the union. In the next section, we describe a tree-based implementation of a union-find structure where find operations are the bottleneck, but the amortized time performance for each operation is much better than $O(\log n)$.

7.3 A Tree-Based Implementation

An alternative data structure for implementing a union-find structure, starting from n singleton sets, is to use a collection of trees to store the elements in sets, where each tree is associated with a different set. In particular, we implement a tree T with a linked data structure, where each node u of T stores an element of the set associated with T, and a parent pointer to the parent node of u. If u is the root, then its parent pointer points to itself. As in the list-based implementation, we assume that either we have a constant-time way of accessing the node associated with an element from the element itself or that the nodes of in our data structure serve as the elements in our partition. Also, in this implementation, we identify each set with the root of its associated tree. (See Figure 7.7.)

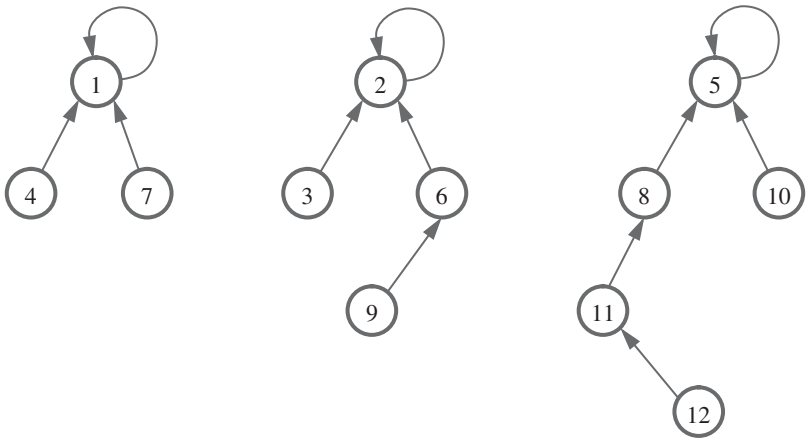
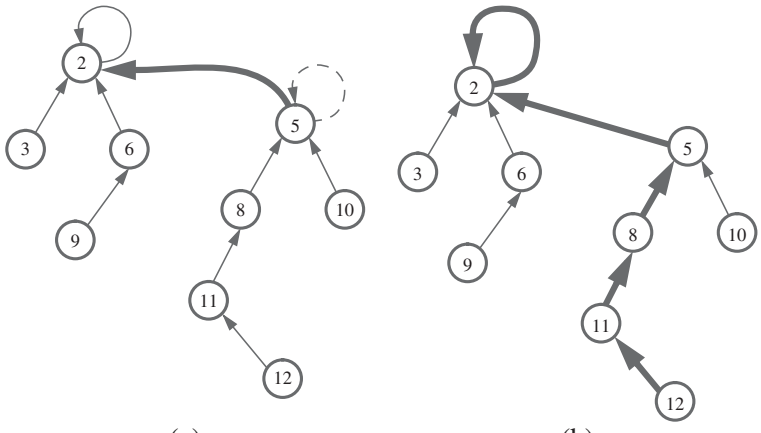


Figure 7.7: A tree-based implementation of a union-find structure for three disjoint sets: $A = \{1, 4, 7\}$, $B = \{2, 3, 6, 9\}$, and $C = \{5, 8, 10, 11, 12\}$.

With this data structure, the operation union is performed by making one of the two trees a subtree of the other (Figure 7.8a), which can be done in $O(1)$ time by setting the parent pointer of the root of one tree to point to the root of the other tree. Operation find for an element e is performed by walking up to the root of the tree containing e (Figure 7.8b), which takes $O(n)$ time in the worst case.

Note that this representation of a tree is a specialized data structure used to implement a union-find structure, and it is not meant to be a realization of a general tree data structure. Indeed, this representation has only parent links, and does not provide a way to access the children of a given node.

At first, this implementation may seem to be no better than the list-based implementation of a union-find structure, but we add the following simple heuristics to make it run faster.



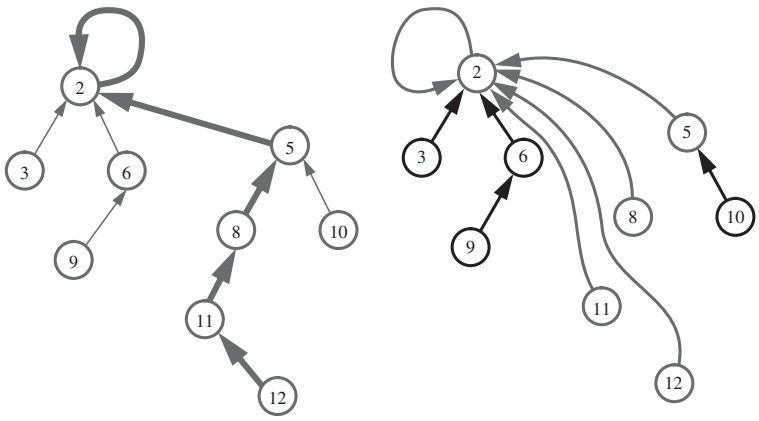
(a) (b)

Figure 7.8: Tree-based implementation of a partition: (a) operation union(A, B); (b) operation find(ℓ), where ℓ denotes the node for element 12.

Union-by-Size: Store with each node v the size of the subtree rooted at v, denoted by $n(v)$. In a union, we now make the tree of the smaller set a subtree of the other tree, and update the size field of the root of the resulting tree.

Path Compression: In a find operation, for each node v that the find visits, reset the parent pointer from v to point to the root. (See Figure 7.9.)

These heuristics increase the running time of an operation by a constant factor, but as we show in the section that follows, they significantly improve the amortized running time for performing a sequence of union and find operations.



(a) (b)

Figure 7.9: Path-compression heuristic: (a) path traversed by operation find on element 12; (b) restructured tree.

We give pseudocode for implementing the fundamental methods of the disjoint sets data structures using the tree-based implementation in Algorithm 7.10.

Algorithm makeSet():
  for each singleton element, x **do**
    x.parent $\leftarrow x$
    x.size $\leftarrow 1$

Algorithm union(x, y):
  if x.size $< y$.size **then**
    x.parent $\leftarrow y$
    y.size $\leftarrow y$.size $+ x$.size
  else
    y.parent $\leftarrow x$
    x.size $\leftarrow x$.size $+ y$.size

Algorithm find(x):
  $r \leftarrow x$
  while r.parent $\neq r$ **do** // find the root
    $r \leftarrow r$.parent
  $z \leftarrow x$
  while z.parent $\neq z$ **do** // path compression
    $w \leftarrow z$
    $z \leftarrow z$.parent
    w.parent $\leftarrow r$

Algorithm 7.10: The fundamental disjoint-set methods for an implementation based on trees defined by "parent" pointers for every element. Note: the x and y parameters for the union method must both be roots of their respective trees.

The above tree-based implementation has a very interesting analysis, which we give in the next section. Before giving that analysis, however, let us first give a simple analysis for the above implementation.

Theorem 7.3: *Performing a sequence, σ, of m* union *and* find *operations, starting from n singleton sets, using the above tree-based implementation of a union-find structure, takes $O(n + m \log n)$ time.*

Proof: There are at most $O(n)$ union operations, and each runs in $O(1)$ time; hence, the total time for doing all the union operations is $O(n)$. We only change a node's parent pointer when we union its set into a set that is at least as large as it. Thus, each time we go from a node to its parent, in doing a find operation, the size of the set rooted at that node must at least double. Therefore, the longest sequence of parent pointers that we can march through in doing a find is $O(\log n)$. ■

7.3.1 Analyzing the Tree-Based Implementation

Our analysis of the tree-based implementation of the union-find operations is based on the use of a very slow-growing function, $\alpha(n)$, which is the inverse of a very fast-growing function, known as the ***Ackermann function***.

The Ackermann Function

The Ackermann function is named after Wilhelm Ackermann, a German mathematician who was a student of the well-known mathematician David Hilbert. There are actually several versions of the Ackermann function, including one originally proposed by Wilhelm Ackermann, each of which is more or less equivalent with the others. Each version of the Ackermann function defines a very fast-growing function, which has the interesting property that it cannot be computed using only Pascal-style for-loops (as it also requires the use of while-loops). We are not primarily interested in this structural property of the Ackermann function, however, which was Ackermann's original motivation for proposing it. Instead, we are interested in the Ackermann function because it is useful for analyzing the tree-based implementation of union-find operations.

The version of the Ackermann function we use is based on an indexed function, A_i, which is defined as follows, for integers $x \geq 0$ and $i > 0$:

$$\begin{aligned} A_0(x) &= x + 1 \\ A_{i+1}(x) &= A_i^{(x)}(x), \end{aligned}$$

where $f^{(k)}$ denotes the k-fold composition of the function f with itself. That is,

$$\begin{aligned} f^{(0)}(x) &= x \\ f^{(k)}(x) &= f(f^{(k-1)}(x)). \end{aligned}$$

So, in other words, $A_{i+1}(x)$ involves making x applications of the A_i function on itself, starting with x. This indexed function actually defines a progression of functions, with each function growing much faster than the previous one:

- $A_0(x) = x + 1$, which is the increment-by-one function
- $A_1(x) = 2x$, which is the multiply-by-two function
- $A_2(x) = x2^x \geq 2^x$, which is the power-of-two function
- $A_3(x) \geq 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (with x number of 2's), which is the tower-of-twos function
- $A_4(x)$ is greater than or equal to the tower-of-tower-of-twos function
- and so on.

We then define the ***Ackermann function*** as

$$\mathcal{A}(x) = A_x(2),$$

which is an incredibly fast-growing function. To get some perspective, note that $\mathcal{A}(3) = 2048$ and $\mathcal{A}(4)$ is greater than or equal to a tower of 2048 twos, which is much larger than the number of subatomic particles in the universe. Likewise, its inverse,

$$\alpha(n) = \min\{x\colon\ \mathcal{A}(x) \geq n\},$$

is an incredibly slow-growing function. Still, even though $\alpha(n)$ is indeed growing as n goes to infinity, for all practical purposes, $\alpha(n) \leq 4$.

An Amortized Analysis

Let us use an amortization argument to analyze the running time of a sequence, σ, of m union and find operations on a partition that initially consists of n single-element sets.

Let U be the tree defined by all the union operations in σ ***without*** our having performed any path compressions. For each node v, let $n(v)$ denote the number of nodes in the subtree of U rooted at v, and define the ***rank*** of v, which we denote as $r(v)$, as follows:

$$r(v) = \lfloor \log n(v) \rfloor + 2.$$

Thus, we immediately get that $n(v) \geq 2^{r(v)-2}$. Also, since there are at most n nodes in U, $r(v) \leq \lfloor \log n \rfloor + 2$, for each node v.

Lemma 7.4: *If node w is the parent of node v, then*

$$r(v) < r(w).$$

Proof: We make v point to w only if the size of w before the union is at least as large as the size of v. Let $n(w)$ denote the size of w before the union and let $n'(w)$ denote the size of w after the union. Thus, after the union we get

$$\begin{aligned} r(v) &= \lfloor \log n(v) \rfloor + 2 \\ &< \lfloor \log n(v) + 1 \rfloor + 2 \\ &= \lfloor \log 2n(v) \rfloor + 2 \\ &\leq \lfloor \log(n(v) + n(w)) \rfloor + 2 \\ &= \lfloor \log n'(w) \rfloor + 2 \\ &\leq r(w). \end{aligned}$$

■

Put another way, this theorem states that ranks are ***strictly*** increasing as we follow parent pointers up the union tree. It also implies the following.

Lemma 7.5: *The number of nodes of rank s, $0 \leq s \leq \lfloor \log n \rfloor + 2$, is at most*

$$\frac{n}{2^{s-2}}.$$

Proof: By the previous theorem, $r(v) < r(w)$, for any node v with parent w, and ranks are strictly increasing as we follow parent pointers up any tree. Thus, if $r(v) = r(w)$ for two nodes v and w, then the nodes counted in $n(v)$ must be separate and distinct from the nodes counted in $n(w)$. By the definition of rank, if a node v is of rank s, then $n(v) \geq 2^{s-2}$. Therefore, since there are at most n nodes total, there can be at most $n/2^{s-2}$ that are of rank s. ■

Let us now consider the time it takes to perform the m union and find operations in the sequence σ. In particular, let us develop an amortized analysis, assuming that it takes 1 cyber-dollar to perform $O(1)$ amount of work during our performance of σ. We have already observed that performing a union takes $O(1)$ time. We therefore charge each union operation 1 cyber-dollar to pay for this. This implies that the total charges to all union operations is $O(n)$, since there can be at most $n - 1$ union operations in all.

Analyzing Path Compression

To characterize the performance of the other operations in σ, let us therefore consider how path compression affects the performance of find operations. As we perform path compressions, for each node v, we may be changing the parent, $p(v)$, of v. Note that, by Lemma 7.4, every time we change $p(v)$ during the execution of σ, we increase the value of $r(p(v))$, the rank of v's parent.

For the sake of our amortized analysis, let us define a ***labeling function***, $L(v)$, for each node v, which changes over the course of the execution of the operations in σ. In particular, at each step t in the sequence σ, define $L(v)$ as follows:

$$L(v) = \text{ the largest } i \text{ for which } r(p(v)) \geq A_i(r(v)).$$

Note that if v has a parent, then $L(v)$ is well-defined and is at least 0, since

$$r(p(v)) \geq r(v) + 1 = A_0(r(v)),$$

because ranks are strictly increasing as we go up the tree U. Also, for $n \geq 5$, the maximum value for $L(v)$ is $\alpha(n) - 1$, since, if $L(v) = i$, then

$$\begin{aligned} n &> \lfloor \log n \rfloor + 2 \\ &\geq r(p(v)) \\ &\geq A_i(r(v)) \\ &\geq A_i(2). \end{aligned}$$

Or, put another way,

$$L(v) < \alpha(n),$$

for all v and t.

The main computational task in performing a find operation is in following parent pointers along a path, P, from a some node u up to the root, z, of the tree containing u at time t. We can account for all of this work by paying 1 cyber-dollar for each parent pointer we traverse. Let v be some node along P. We use two rules for charging a cyber-dollar for following the parent pointer for v:

- If v has an ancestor w in P such that $L(v) = L(w)$, at this point in time, then we charge 1 cyber-dollar to v itself.
- If v has no such ancestor, then we charge 1 cyber-dollar to this find operation.

Thus, the maximum number of cyber-dollars any find can get charged is bounded by the number of distinct $L(v)$ values for nodes on the path P, which is less than $\alpha(n)$, as we observed above. The total amount of cyber-dollars charged to all the find operations is therefore $O(m\,\alpha(n))$.

Let us next consider all the charges that are made to a vertex v over the course of performing the find operations in σ. For such a charge to occur at time t, then v must have an ancestor w such that $L(v) = L(w) = i$, for some i. So, at time t,

$$r(p(v)) \geq A_i(r(v))$$

and

$$r(p(w)) \geq A_i(r(w)).$$

Suppose, in particular, that

$$r(p(v)) \geq A_i^{(k)}(r(v)),$$

where $k \geq 1$. Recall that z denotes the last vertex on the path P. Then at time t,

$$\begin{aligned} r(z) &\geq r(p(w)) \\ &\geq A_i(r(w)) \\ &\geq A_i(r(p(v))) \\ &\geq A_i(A_i^{(k)}(r(v))) \\ &\geq A_i^{(k+1)}(r(v)). \end{aligned}$$

Therefore, since z becomes the new parent of v at time $t+1$, because of path compression, we have at time $t+1$,

$$r(p(v)) \geq A_i^{(k+1)}(r(v)).$$

This implies that at most $r(v)$ charges can be made to v before

$$\begin{aligned} r(p(v)) &\geq A_i^{(r(v))}(r(v)) \\ &= A_{i+1}(r(v)), \end{aligned}$$

at which point $L(v) = i+1$. Thus, after at most $r(v)$ cyber-dollars are charged against v, $L(v)$ increases by at least 1. Since $L(v)$ can increase at most $\alpha(n) - 1$ times, this means that there can be at most $r(v)$ times $\alpha(n)$ cyber-dollars charged

to v in total. Combining this fact with Lemma 7.5, there are at most

$$s\,\alpha(n)\frac{n}{2^{s-2}} = n\,\alpha(n)\frac{s}{2^{s-2}}$$

cyber-dollars charged to all the vertices of rank s. Summing over all possible ranks, the total number of cyber-dollars charged to all nodes is at most

$$\begin{aligned}\sum_{s=0}^{\lfloor \log n \rfloor + 2} n\,\alpha(n)\frac{s}{2^{s-2}} &\leq \sum_{s=0}^{\infty} n\,\alpha(n)\frac{s}{2^{s-2}} \\ &= n\,\alpha(n)\sum_{s=0}^{\infty}\frac{s}{2^{s-2}} \\ &\leq 8n\,\alpha(n),\end{aligned}$$

by a well-known summation bound (see Theorem A.15 in the appendix). That is, the total charges made to all nodes is $O(n\,\alpha(n))$, which gives us the following:

Theorem 7.6: *In a sequence σ of m* union *and* find *operations performed using union-by-size and path compression, starting with a collection of n single-element sets, the total time to perform the operations in σ is $O((n+m)\alpha(n))$.*

7.4 Exercises

Reinforcement

R-7.1 Suppose we have a social network with members A, B, C, D, E, F, and G, and the set of friendship ties,

$$\{(A,B),(B,C),(C,A),(D,E),(F,G)\}.$$

What are the connected components?

R-7.2 Suppose a social network, N, contains n people, m edges, and c connected components. What is the exact number of times that each of the methods, makeSet, union, and find, are called in computing the connected components for N using Algorithm 7.2?

R-7.3 How many walls were erased to construct the maze in Figure 7.3, not counting the start and finish walls?

R-7.4 For the sake of analysis, if we have a sequence of union, find, and makeSet operations, why can we can assume without loss of generality that all the makeSet operations come first?

R-7.5 One additional feature of the list-based implementation of a union-find structure is that it allows for the contents of any set in a partition to be listed in time proportional to the size of the set. Describe how this can be done.

R-7.6 Suppose we have 20 singleton sets, numbered 0 through 19, and we call the operation union(find(i),find($i+5$)), for $i = 0, 1, 2, \ldots, 14$. Draw a picture of a list-based representation of the sets that result.

R-7.7 Suppose we have 20 singleton sets, numbered 0 through 19, and we call the operation union(find(i),find($i+5$)), for $i = 0, 1, 2, \ldots, 14$. Draw a picture of a tree-based representation of the sets that result, assuming we don't implement the union-by-size and path compression heuristics.

R-7.8 Answer the previous exercise assuming that we implement both the union-by-size and path compression heuristics.

Creativity

C-7.1 Describe how to implement a union-find structure using extendable arrays, which each contains the elements in a single set, instead of linked lists. Show how this solution can be used to process a sequence of m union-find operations on an initial collection of n singleton sets in $O(n \log n + m)$ time.

C-7.2 Consider a method, remove(e), which removes e from whichever list it belongs to, in a list-based implementation of a union-find structure. Describe how to modify the list-based implementation so that this method runs in time $O(1)$.

C-7.3 Suppose that we implement a union-find structure by representing each set using a balanced search tree. Describe and analyze algorithms for each of the methods for a union-find structure so that every operation runs in at most $O(\log n)$ time in the ***worst case***.

C-7.4 Let A be a collection of objects. Describe an efficient method for converting A into a set. That is, remove all duplicates from A. What is the running time of this method?

C-7.5 Suppose we implement the tree-based union-find data structure using the union-by-size heuristic and a ***partial*** path-compression heuristic. The partial path compression in this case means that, after performing a sequence of pointer hops for a find operation, we update the parent pointer for each node u along this path to point to its grandparent. Show that the total running time for performing m union and find operations, starting with n singleton sets, is still $O((n+m)\alpha(n))$ in this case.

C-7.6 Suppose we implement the tree-based union-find data structure using the union-by-size and path-compression heuristics. Show that the total running time for performing a sequence of m union and find operations, starting with n singleton sets, is $O(n+m)$ if all the unions come before all the finds.

C-7.7 Suppose we implement the tree-based union-find data structure using the union-by-size heuristic and path-compression heuristics. Show that the total running time for performing a sequence of m union and find operations, starting with n singleton sets, is $O(m)$, if $m \geq n \log n$.

C-7.8 Suppose we implement the tree-based union-find data structure, but we don't use the union-by-size heuristic nor the path-compression heuristic. Show that the total running time for performing a sequence of n union and find operations, starting with n singleton sets, is $\Theta(n^2)$ in this case. That is, provide a proof that it is $O(n^2)$ and an example that requires $\Omega(n^2)$ time.

Applications

A-7.1 Another problem of interest in percolation theory is to determine the threshold probability where a liquid will permeate a porous material. One way to model this is to consider the barriers between pairs of adjacent cells in some random order and remove them in this order. At the point when the top and bottom are connected, we would then take the ratio of r/s as an approximation to this threshold probability, where r is the number of barriers considered up to this point and s is the total number of barriers. Describe an algorithm for efficiently computing this threshold value right at the moment it occurs, given a random listing of the pairwise barriers in the porous material as input.

A-7.2 One of the tasks for an operating system is the job of scheduling computations to be performed by the processor(s) that are part of that system. A subtask that comes up in some processor scheduling problems is to solve a sequence σ of $O(n)$ priority queue operations, where each operation in σ is either an insert(i) or removeMin(), such that i is a distinct integer in the range from 1 to n. This problem is known as the ***offline-min problem***, since the entire sequence, σ, is

given in advance. Interestingly, the offline nature of this problem gives us a faster way of answering the operations in σ than using a heap. Namely, if k is the smallest integer inserted somewhere in σ, then we know the very next removeMin() in σ will return k. Likewise, after we have matched up k and this removeMin(), and deleted both from σ, then we can repeat this argument on the operations that remain. Use this observation to design an algorithm for answering all the operations in σ in $O(n\alpha(n))$ time, and thereby solve the offline-min problem.

A-7.3 In image-processing applications, such as for optical character recognition, it is often useful to group together contiguous sets of similarly colored pixels in an image. (See Figure 7.11.) For instance, in a black-and-white image, we might say that a black pixel, p, is adjacent with another black pixel, q, if q shares a boundary with p's North, East, South, or West boundary. Typically, the way an image is represented imposes constraints on an algorithm for finding the contiguous parts of similarly colored pixels, and, as an image-processing algorithm designer, we often don't get to dictate the order in which pixels are presented. Design an efficient algorithm that can take a sequence of black or white pixels, given in an arbitrary order, taken from some image, and output all the contiguous shapes in that image. You may assume that each time a pixel is given, you are told its (x, y)-coordinates and the colors of its North, East, South, and West neighbors.

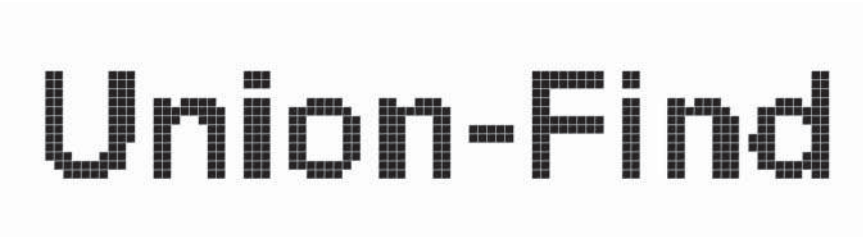


Figure 7.11: Contiguous regions of similarly colored pixels in an image. This image has 12 such regions.

A-7.4 The game of ***Hex*** is said to have, as one of its inventors, the mathematician John Nash, who is the subject of the book and movie *A Beautiful Mind*. In this game, two players, one playing black and the other playing white, take turns placing stones of their respective colors on an $n \times n$ hexagonal grid. Once a stone is placed, it cannot be moved. The black player's goal is to connect the top and bottom sides of the grid, and the white player's goal is to connect the left and right sides of the grid, using stones of their respective colors. Two cells are considered connected if they share an edge and both have the same color stone. (See Figure 7.12.) Describe an efficient scheme where you can determine after each move whether black or white has just won a game of Hex.

A-7.5 Consider the game of Hex, as in the previous exercise, but now with a twist. Suppose some number, k, of the cells in the game board are colored gold and if the set of stones that connect the two sides of a winning player's board are also connected to $k' \leq k$ of the gold cells, then that player gets k' bonus points.

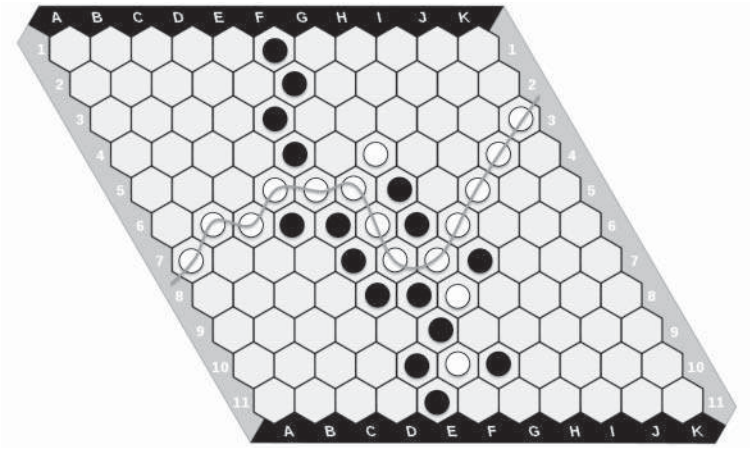


Figure 7.12: An 11×11 instance of the game of Hex. In this case, the white player has just placed a stone to create a winning configuration. (Background Hex board image is in the public domain; credit: Tiltec.)

> Describe an efficient way to detect when a player wins and also, at that same moment, determine how many bonus points they get. What is the running time of your method over the course of a game consisting of n moves?

Chapter Notes

Tarjan [206] was the first to show that a sequence of n union and find operations, implemented as described in this chapter, can be performed in $O(n\,\alpha(n))$ time, and this bound is tight in the worst case (see also [207, 209]). The analysis we give for the tree-based implementation of the union-find operations is based on related discussions by Hopcroft and Ullman [103] and Kozen [135]. Cormen *et al.* [51] provide an alternative analysis based on a potential argument and another analysis can be found in a more recent paper by Seidel and Sharir [192].

Kaplan *et al.* [116] and Alstrup *et al.* [11] study an efficient way to support deletions as well as union-find operations. Agarwal *et al.* [5] study the union-find problem in the external-memory model and explore several applications in terrain analysis. Gabow and Tarjan [77] show that certain special cases of the union-find problem can be solved in linear time, including instances that arise in the offline-min problem, and Frederickson [75] shows a connection between processor scheduling and the offline-min problem. Dillencourt *et al.* [57] study the problem of finding contiguous sets of similarly colored pixels in an image and show that if an image is given in raster order, this problem can be solved in linear time by exploiting special properties in the union-find problems that arise in this application. For more algorithms for social networks, see the book by Easley and Kleinberg [60].