Union-Find

(see Section 21 on Data Structures for Disjoint Sets in Cormen, Leiserson, Rivest, Stein)

This section presents two data structures for the disjoint set system problem we encountered in the implementation of Kruskal’s algorithm for minimum spanning trees. An interesting feature of the problem is that operations can be executed in a time that is only ever so slightly more than linear in m.

Abstract Data Type. A disjoint set system is an abstract data type that represents a partition C of a set \([n] = \{1, 2, \ldots, n\}\). In other words, \(C\) is a set of pairwise disjoint subsets of \([n]\) such that the union of all sets in \(C\) is \([n]\). The data type supports

- \(\text{int FIND}(i) : \text{return } P \in C \text{ with } i \in P;\)
- \(\text{void UNION}(P, Q) : C = C - \{P, Q\} \cup \{P \cup Q\}.\)

In most applications, the sets themselves are irrelevant, and it is only important to know when two elements belong to the same set and when they belong to different sets in the system. For example, Kruskal’s algorithm executes the operations only in the following sequence:

\[
P = \text{FIND}(i); \quad Q = \text{FIND}(j); \quad \text{if } P \neq Q \text{ then UNION}(P, Q) \text{ endif.}
\]

This is similar to many everyday situations where other than for communication it is not important to know what it is as long as we recognize when two are the same and when they are different.

Linked Lists. We construct a fairly simple and reasonably efficient first solution using linked lists for the sets. We use a table of length \(n\), and for each \(i \in [n]\), we store the name of the set that contains \(i\). Furthermore, we link the elements of the same set and use the name of the first element as the name of the set. Figure 99 shows a sample set system and its representation. It is convenient to also store the size of the set with the first element.

![Figure 99: The system consists of three sets, each named by the bold element. Each element stores the name of its set, possibly the size of its set, and possibly a pointer to the next element in the same set.](image)

To perform a UNION operation, we need to change the name for all elements in one of the two sets. To save time, we do this only for the smaller set. To merge the two lists without traversing the longer one, we insert the shorter list between the first two elements of the longer list.

\[
\text{void UNION(int P, Q)} \\
\text{if } C[P].\text{size} < C[Q].\text{size} \text{ then } P \leftrightarrow Q \text{ endif;} \\
C[P].\text{size} = C[P].\text{size} + C[Q].\text{size}; \\
second = C[P].\text{next}; \quad C[P].\text{next} = Q; \quad t = Q; \\
\text{while } t \neq 0 \text{ do} \\
C[t].\text{set} = P; \quad u = t; \quad t = C[t].\text{next} \\
\text{ endwhile; } C[u].\text{next} = second.
\]

In the worst case, a single UNION operation takes time \(\Theta(n)\). The amortized performance is much better because we spend time only on the elements of the smaller set.

Weighted UNION Lemma. \(n - 1\) UNION operations applied to a system of \(n\) singleton sets take time \(O(n \log n)\).
PROOF. For an element, $i$, we consider the cardinality of the set that contains it, $\sigma(i) = C[\text{FIND}(i)].size$. Each time the name of the set that contains $i$ changes, $\sigma(i)$ at least doubles. After changing the name $k$ times, we have $\sigma(i) \geq 2^k$ and therefore $k \leq \log_2 n$. In other words, $i$ can be in the smaller set of a UNION operation at most $\log_2 n$ times. The claim follows because a UNION operation takes time proportional to the cardinality of the smaller set.

Up-trees. Thinking of names as pointers, the above data structure stores each set in a tree of height one. We can use more general trees and get more efficient UNION operations at the expense of slower FIND operations. We consider a class of algorithms with the following commonalities:

- each set is a tree and the name of the set is the index of the root element;
- FIND traverses a path from a node to the root;
- UNION links two trees.

It suffices to store only one pointer per node, namely the pointer to the parent. This is why these trees are called up-trees. It is convenient to let the root point to itself.

![Figure 100: The UNION operations create a tree by linking the root of the first set to the root of the second set.](image)

Figure 100 shows the up-tree generated by executing the following eleven UNION operations on a system of twelve singleton sets: $2 \cup 3$, $4 \cup 7$, $2 \cup 4$, $1 \cup 2$, $4 \cup 10$, $9 \cup 12$, $12 \cup 2$, $8 \cup 11$, $8 \cup 2$, $5 \cup 6$, $6 \cup 1$. Figure 101 shows the embedding of the tree in a table. UNION takes constant time and FIND takes time proportional to the length of the path, which can be as large as $n - 1$.

Weighted Union. The running time of FIND can be improved by linking smaller to larger trees. Assume a field $C[i].p$ for the index of the parent ($C[i].p = i$ if $i$ is a root), and a field $C[i].size$ for the number of elements in the tree rooted at $i$. We need the size field only for the roots and we need the index to the parent field everywhere except for the roots. The FIND and UNION operations can now be implemented as follows:

```c
int FIND(int i)
    if C[i].p != i then return FIND(C[i].p) endif;
    return i.

void UNION(int i, j)
    if C[i].size < C[j].size then i ↔ j endif;
    C[i].size = C[i].size + C[j].size;
    C[j].p = i.
```

The size of a subtree increases by at least a factor of 2 from a node to its parent. The depth of a node can therefore not exceed $\log_2 n$. It follows that FIND takes at most time $O(\log n)$. We formulate the result on the height for later reference.

HEIGHT LEMMA. An up-tree created from $n$ singleton nodes by $n - 1$ weighted union operations has height at most $\log_2 n$.

Path Compression. We can further improve the time for FIND operations by linking traversed nodes directly to the root. This is the idea of path compression. The UNION operation is implemented as before and there is only one modification in the implementation of the FIND operation:

```c
int FIND(int i)
    if C[i].p != i then C[i].p = FIND(C[i].p) endif;
    return C[i].p.
```

If $i$ is not root then the recursion makes it the child of a root, which is then returned. If $i$ is a root, it returns itself because in this case $C[i].p = i$, by convention. Figure 102 illustrates the algorithm by executing a sequence of eight operations $i \cup j$, which is short for finding the sets
that contain \( i \) and \( j \), and performing a UNION operation if the sets are different. At the beginning, every element forms its own one-node tree. With path compression, it is difficult to imagine that long paths can develop at all.

**Iterated Logarithm.** We will prove shortly that the iterated logarithm is an upper bound on the amortized time for a FIND operation. We begin by defining the function from its inverse. Let \( F(0) = 1 \) and \( F(i + 1) = 2^{F(i)} \). We have \( F(1) = 2 \), \( F(2) = 2^2 \), and \( F(3) = 2^2 \). In general, \( F(i) \) is the tower of \( i \) 2s. Table 14 shows the values of \( F \) for the first six arguments. For \( i \leq 3 \), \( F \) is very small, but for \( i = 5 \) it already exceeds the number of atoms in our universe. Note that the binary logarithm of a tower of \( i \) 2s is a tower of \( i - 1 \) 2s. The *iterated logarithm* is the number of times we can take the binary logarithm before we drop down to a non-positive value. In other words, the iterated logarithm is the inverse of \( F \),

\[
\log^* n = \min \{ i \mid F(i) \geq n \} = \min \{ i \mid \log_2 \log_2 \ldots \log_2 n \leq 1 \},
\]

where the binary logarithm is taken \( i \) times. As \( n \) goes to infinity, \( \log^* n \) goes to infinity, but it does this in a painstakingly slow manner.

**Levels and Groups.** The analysis of the path compression algorithm uses two Census Lemmas discussed shortly. Let \( A_1, A_2, \ldots, A_n \) be a sequence of UNION and FIND operations, and let \( T \) be the collection of up-trees we get by executing the sequence, but without path compression. In other words, the FIND operations have no influence on the trees. The *level* \( \lambda(\mu) \) of a node \( \mu \) is its height in \( T \) plus one.

**LEVEL CENSUS LEMMA.** There are at most \( 2n/2^\ell \) nodes at level \( \ell \).

**PROOF.** A node at level \( \ell \) has a subtree of at least \( 2^{\ell-1} \) nodes. Subtrees of nodes on the same level are disjoint. 

![level census lemma](image)

Note that if \( \mu \) is a proper descendent of another node \( \nu \) at some moment during the execution of the operation sequence then \( \mu \) is a proper descendent of \( \nu \) in \( T \). In this case \( \lambda(\mu) < \lambda(\nu) \).

**GROUP CENSUS LEMMA.** There are at most \( 2n/F(g) \) nodes with group number \( g \).

![group census lemma](image)
PROOF. The nodes with group number \( g \) have level between \( F(g - 1) + 1 \) and \( F(g) \). We use the Level Census Lemma to bound their number:

\[
\sum_{\ell = F(g-1)+1}^{F(g)} \frac{2n}{2^\ell} \leq \frac{2n \cdot (1 + \frac{1}{2} + \frac{1}{4} + \ldots)}{2^{F(g-1)+1}} = \frac{2n}{F(g)},
\]

as claimed. \( \Box \)

Analysis. We show that any sequence of \( m \geq n \) UNION and FIND operations on a ground set \( [n] \) takes time at most \( O(m \log^* n) \) if weighted union and path compression is used. We can focus on FIND because UNION operations take only constant time. For a FIND operation \( A_i \), let \( X_i \) be the set of nodes along the traversed path. The total time for executing all FIND operations is proportional to

\[
x = \sum_i \text{card} X_i.
\]

For \( \mu \in X_i \), let \( p_i(\mu) \) be the parent during the execution of \( A_i \). We partition \( X_i \) into the topmost two nodes, the nodes just below boundaries between groups, and the rest:

\[
Y_i = \{ \mu \in X_i \mid \mu \text{ is root or child of root}\},
\]

\[
Z_i = \{ \mu \in X_i - Y_i \mid g(\mu) < g(p_i(\mu)) \},
\]

\[
W_i = \{ \mu \in X_i - Y_i \mid g(\mu) = g(p_i(\mu)) \}.
\]

Clearly, \( \text{card} Y_i \leq 2 \) and \( \text{card} Z_i \leq \log^* n \). It remains to bound the total size of the \( W_i \), \( w = \sum_i \text{card} W_i \). Instead of counting, for each \( A_i \), the nodes in \( W_i \), we count, for each node \( \mu \), the FIND operations \( A_j \) for which \( \mu \in W_j \). In other words, we count how often \( \mu \) changes parent until its parent has a higher group number than \( \mu \). Each time \( \mu \) changes parent, the new parent has higher level than the old parent. If follows that the number of changes is at most \( F(g(\mu)) - F(g(\mu) - 1) \). The number of nodes with group number \( g \) is at most \( \frac{2n}{F(g)} \) by the Group Census Lemma. Hence

\[
w \leq \sum_{g=0}^{\log^* n} \frac{2n}{F(g)} \cdot (F(g) - F(g - 1))
\]

\[
\leq 2n \cdot (1 + \log^* n).
\]

This implies that

\[
x \leq 2m + m \log^* n + 2n(1 + \log^* n)
\]

\[
= O(m \log^* n),
\]

assuming \( m \geq n \). This is an upper bound on the total time it takes to execute \( m \) FIND operations. The amortized cost per FIND operation is therefore at most \( O(\log^* n) \), which for all practical purposes is a constant.

Summary. We proved an upper bound on the time needed for \( m \geq n \) UNION and FIND operations. The bound is more than constant per operation, although for all practical purposes it is constant. The \( \log^* n \) bound can be improved to an even smaller function, usually referred to as \( \alpha(n) \) or the inverse of the Ackerman function, that goes to infinity even slower than the iterated logarithm. It can also be proved that (under some mild assumptions) there is no algorithm that can execute general sequences of UNION and FIND operations in amortized time that is asymptotically less than \( \alpha(n) \).