

16 Union-Find

In this lecture, we present 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 m 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 $[n] = \{1, 2, \dots, n\}$. 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

```
set FIND( $i$ ): return  $P \in C$  with  $i \in P$ ;  
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:

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

This is similar to many everyday situations where it is usually 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 68 shows a sample set system and its representation. It is convenient to also store the size of the set with the first element.

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.

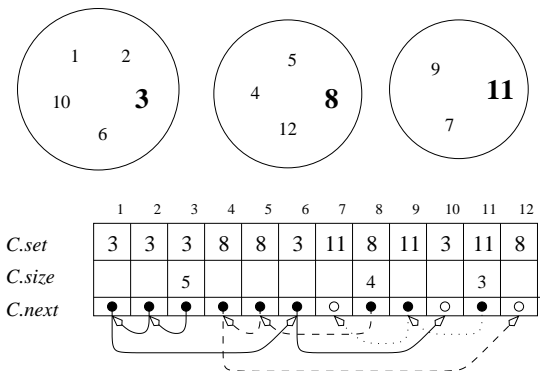


Figure 68: The system consists of three sets, each named by the bold element. Each element stores the name of its set, possibly the size of its set, and possibly a pointer to the next element in the same set.

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

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. \square

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;
- 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.

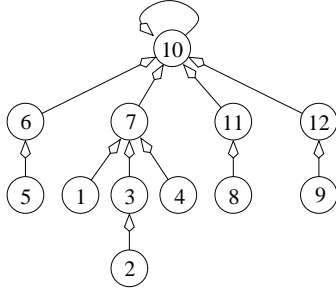


Figure 69: The UNION operations create a tree by linking the root of the first set to the root of the second set.

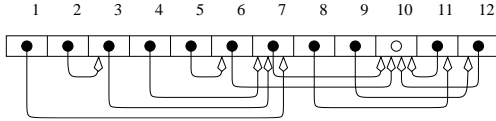


Figure 70: The table stores indices which function as pointers as well as names of elements and of sets. The white dot represents a pointer to itself.

Figure 69 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 70 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. This is the idea of *weighted union* again. 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:

```
int FIND(int i)
  if  $C[i].p \neq i$  then return FIND( $C[i].p$ ) endif;
  return  $i$ .
```

```
void UNION(int i, j)
  if  $C[i].size < C[j].size$  then  $i \leftrightarrow j$  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:

```
int FIND(int i)
  if  $C[i].p \neq i$  then  $C[i].p = \text{FIND}(C[i].p)$  endif;
  return  $C[i].p$ .
```

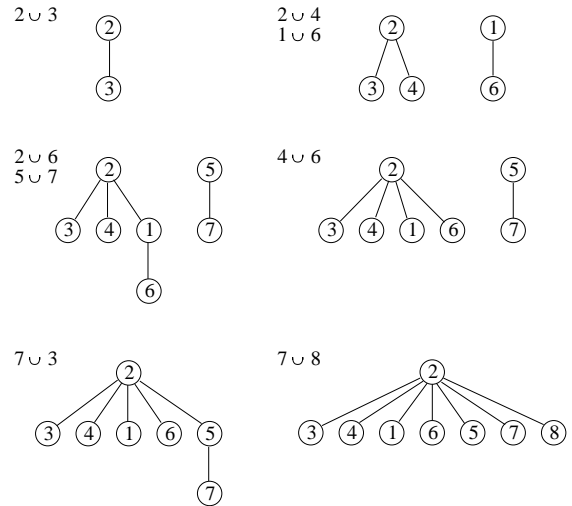


Figure 71: The operations and up-trees develop from top to bottom and within each row from left to right.

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 71 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 $F(3) = 2^{2^2}$. In general, $F(i)$ is the tower of i 2s. Table 5 shows the values of F for the first six arguments. For $i \leq 3$, F is very small, but

i	0	1	2	3	4	5
F	1	2	4	16	65,536	$2^{65,536}$

Table 5: Values of F .

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 one or less. In other words, the iterated logarithm is the inverse of F ,

$$\begin{aligned} \log^* n &= \min\{i \mid F(i) \geq n\} \\ &= \min\{i \mid \log_2 \log_2 \dots \log_2 n \leq 1\}, \end{aligned}$$

where the binary logarithm is taken i times. As n goes to infinity, $\log^* n$ goes to infinity, but very slowly.

Levels and groups. The analysis of the path compression algorithm uses two Census Lemmas discussed shortly. Let A_1, A_2, \dots, A_m 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 μ is its height of its subtree in T plus one.

LEVEL CENSUS LEMMA. There are at most $n/2^{\ell-1}$ nodes at level ℓ .

PROOF. We use induction to show that a node at level ℓ has a subtree of at least $2^{\ell-1}$ nodes. The claim follows because subtrees of nodes on the same level are disjoint. \square

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

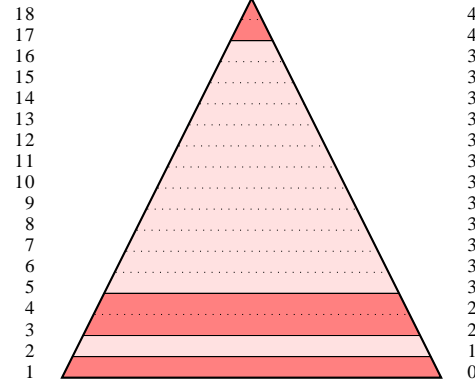


Figure 72: A schematic drawing of the tree T between the column of level numbers on the left and the column of group numbers on the right. The tree is decomposed into five groups, each a sequences of contiguous levels.

Define the *group number* of a node μ as the iterated logarithm of the level, $g(\mu) = \log^* \lambda(\mu)$. Because the level does not exceed n , we have $g(\mu) \leq \log^* n$, for every node μ in T . The definition of g decomposes an up-tree into at most $1 + \log^* n$ groups, as illustrated in Figure 72. The number of levels in group g is $F(g) - F(g-1)$, which gets large very fast. On the other hand, because levels get smaller at an exponential rate, the number of nodes in a group is not much larger than the number of nodes in the lowest level of that group.

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

PROOF. Each node with group number g has level between $F(g-1) + 1$ and $F(g)$. We use the Level Census Lemma to bound their number:

$$\begin{aligned} \sum_{\ell=F(g-1)+1}^{F(g)} \frac{n}{2^{\ell-1}} &\leq \frac{n \cdot (1 + \frac{1}{2} + \frac{1}{4} + \dots)}{2^{F(g-1)}} \\ &= \frac{2n}{F(g)}, \end{aligned}$$

as claimed. \square

Analysis. The analysis is based on the interplay between the up-trees obtained with and without path compression.

The latter are constructed by the weighted union operations and eventually form a single tree, which we denote as T . The former can be obtained from the latter by the application of path compression. Note that in T , the level strictly increases from a node to its parent. Path compression preserves this property, so levels also increase when we climb a path in the actual up-trees.

We now 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 each UNION operation takes 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:

$$\begin{aligned} Y_i &= \{\mu \in X_i \mid \mu \text{ is root or child of root}\}, \\ Z_i &= \{\mu \in X_i - Y_i \mid g(\mu) < g(p_i(\mu))\}, \\ W_i &= \{\mu \in X_i - Y_i \mid g(\mu) = g(p_i(\mu))\}. \end{aligned}$$

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 μ , the FIND operations A_j for which $\mu \in W_j$. In other words, we count how often μ can change parent until its parent has a higher group number than μ . Each time μ changes parent, the new parent has higher level than the old parent. It 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 $2n/F(g)$ by the Group Census Lemma. Hence

$$\begin{aligned} w &\leq \sum_{g=0}^{\log^* n} \frac{2n}{F(g)} \cdot (F(g) - F(g-1)) \\ &\leq 2n \cdot (1 + \log^* n). \end{aligned}$$

This implies that

$$\begin{aligned} x &\leq 2m + m \log^* n + 2n(1 + \log^* n) \\ &= O(m \log^* n), \end{aligned}$$

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 Ackermann 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 $\alpha(n)$.