23 The Union-Find problem

(23.1) The UNION-FIND problem is to maintain an evolving partition of a fixed set of n items — $\{0 \ldots n-1\}$ for simplicity — under two operations:

FIND(x): to identify the set in the partition currently containing x, and

UNION($s1, s2$): given two different sets in the partition to combine them into one.

The partition is initially into n singleton sets, and it coarsens over a mixed sequence of UNIONs and FINDs — there can be at most $n-1$ UNIONs, which would bring all the items into the same set.

It is natural to represent the partition as a forest of trees, defined by an array `parent`, where `parent[x]` is the parent of x in the forest — -1 if x is currently the root of a tree of the forest. The array is initially zero.

With this arrangement, to implement `find(x)` it is enough to return the root of the tree currently containing x, and to implement `union(x1, x2)`, where x_1 and x_2 are currently roots, just make x_1 the parent of x_2 or vice-versa.

(23.2) Potential inefficiency, and size array. With no further control over the structures it is possible that the trees in the forest become deep, so a `find` operation could visit $\Omega(n)$ nodes, which is hardly efficient. The first improvement is to control the `union` operation to keep the trees shallow. This is achieved using another array `size`, where

As long as `x` is a root, `size[x]` is the number of descendants `x` has in the forest.

(23.3) Size balancing. The array `size` is initialised to 1 for each entry, and the `parent[]` array to -1. The UNION operation ensures that the node with fewer descendants is not made parent. This heuristic is called *size balancing.*

```
void union ( int x1, x2 )
{
  if (size[x1] < size[x2])
  {
    parent[x1] = x2;
    size[x2] += size[x1];
  }
  else
  {
    parent[x2] = x1;
    size[x1] += size[x2];
  }
}
```

Clearly `size[x]` is maintained as the number of descendants of x, whether or not x is a root: however this will not hold after we revise the `find` operation.

(23.4) Lemma *A node of height k has size at least 2^k.*

PROOF. The size-balancing heuristic ensures that the size of the parent of a node is at least twice the size of the node. There exists a path of length k beginning at a leaf and ending at the node; hence the size is at least 2^k. ∎

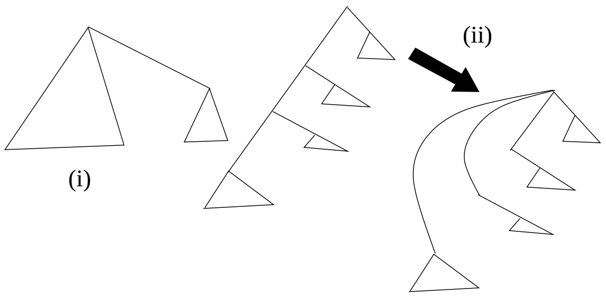

Figure 1: (i) size balancing (ii) path compression

(23.5) Path compression. The revised `find` operation includes a 'path-compression' heuristic as follows:

```
int find ( int x )
{ int w,y,z;
  y = x;
  while ( parent[y] >= 0 )
  {
    y = parent[y];
  }

  z := x;
  while ( z != y )
  {
    w = parent[z];
    parent[z] = y;
    z := w;
  }
  return y;
}
```

The idea is that nodes are brought closer to the root of the tree containing them: the tree retains the same root y, so the heuristic doesn't invalidate the algorithm. However, it doesn't update the `size` array, so `size[x]` is only guaranteed to count x's descendants if x is a root. As a result of the size-balancing strategy, no tree's height exceeds $\log_2 n$.

We now estimate the amortised cost of a series of m unions and n finds.

(23.6) Amortised analysis. This analysis does accounting rather than using a potential function (the latter is possible but not so natural). The cost of a union is $O(1)$, and no more need be said there. In estimating the cost of a find, some of the cost is 'charged' to the find operation and some is 'charged' to the nodes involved. It is just counting the cost in two different ways.

(23.7) The 'reference forest.' We are going to estimate the overall runtime of a mixed sequence, call it S, of UNIONs and FINDs. For the rest of the discussion we shall suppose S to be fixed. The sequence S constructs a forest of trees. It helps to imagine another forest defined in terms of S, formed by executing the UNIONs but not collapsing the trees with path compression. We call this imaginary forest the *reference forest.*

(23.8) Rank. We begin by defining the *transient rank* of a node x to be

$$\lfloor \log_2(\texttt{size[x]}) \rfloor.$$

The transient rank of x is subject to change as long as x remains a root; however, once x acquires a parent, `size[x]` is frozen; we take the *final* value of the transient rank, and define this to be the *rank* of x. Equivalently,

(23.9) Definition *The rank of x is $\lfloor \log_2(s) \rfloor$, where s is the number of descendants x has in the* reference *forest.*

(23.10) Lemma (i) *The parent of x in the real forest is always an ancestor of x in the reference forest;* (ii) `rank(parent[x]) > rank[x]` *always.*

Proof. (i) is true when x first acquires a parent y, since y is its parent in the reference forest. Thereafter whenever the parent u of x is replaced by another node v, by induction we can assume that v is an ancestor of u in the reference forest, and u and ancestor of x, so v is an ancestor of x in the reference forest.

(ii) by the same arguments as in 23.4, all proper ancestors of x in the reference forest have rank greater than `rank(x)`, so (ii) follows from (i). ∎

(23.11) Our analysis will involve a rapidly increasing sequence $F_0, F_1, F_2 \ldots$

$F_0 = 0$, but we shall not specify the sequence yet. Given a node x, let its *rank group* be the least r such that $\texttt{rank}(x) \leq F_r$.

(23.12) Lemma *There are at most $n/2^r$ nodes of rank r.*

Proof. From Lemma 23.10, all nodes of the same rank are independent nodes in the reference forest (that is, neither is ancestor of another, and they have disjoint sets of descendants). By Lemma 23.4, each such node has at least 2^r descendants. Thus, if $x_1, \ldots, x_k$ are all the nodes of rank r and $X_1, \ldots, X_k$ their descendants, then

$$\sum_{j=1}^{k} |X_j| = |\bigcup X_j| \leq n,$$

and $|X_j| \geq 2^r$ for each j, so $\sum 2^r \leq n$, i.e., $k2^r \leq n$ or $k \leq n/2^r$. ∎.

(23.13) Corollary *If $r > 0$, there are fewer than*

$$\frac{n}{2^{F_{r-1}}}$$

nodes in rank group r.

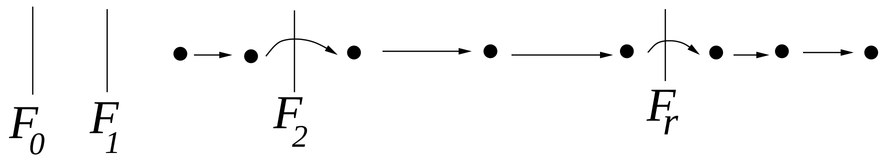

Figure 2: where a find path crosses rank groups

Proof. Nodes with rank group $r > 0$ have ranks between $F_{r-1}+1$ and F_r. Hence, applying the above lemma, there are fewer than

$$\sum_{s=F_{r-1}+1}^{\infty} \frac{n}{2^s} = \frac{n}{2^{F_{r-1}}}. \quad ▮$$

(23.14) The sequence F_j defined. We now specialise the sequence F_j by requiring $F_{j+1} = 2^{F_j}$; a very rapidly increasing sequence, and the above corollary implies

(23.15) Lemma *If $j > 0$ then there are fewer than n/F_j nodes in the j-th rank group.* ▮

(23.16) Definition $\log^* n$ *is the smallest s such that $F_s \geq n$.*

Let L be the index of the highest rank group, i.e., the F_L bounds all node ranks and is the least possible such bound. Since the ranks are bounded by $\lfloor \log_2 n \rfloor$,

$$L \leq \log^* n.$$

(23.17) Amortised cost of a find operation. Let us consider a FIND operation; indeed, let $x_0, x_1 \ldots x_k$ be the sequence of nodes visited tracing from $x_0 = x$ to the current root $y = x_k$. Estimate the real cost of the FIND by the length of the path, i.e., k. We imagine that the cost is measured by the number of nodes from x_0 to x_{k-1}.

Along this path, we ignore the contribution of almost all of the nodes: if a node x_j and its parent x_{j+1} ($j \leq k-2$) are in the same rank *group*, then we ignore the contribution of x_j to the cost. This means we only count the contribution of x_{k-1} and of those other nodes x_j such that x_{j+1} and x_j are in different rank groups: thus the rank of x_j is $\leq F_s$ and the rank of x_{j+1} is $> F_s$ for some s.

There are at most $L+1$ rank groups, so the total number of nodes in the FIND path, counted in this way, is at most $L+1$ (x_{k-1} and L other possibilities). The total contribution of all m FINDs, counted in this way, is $O(m \log^* n)$.

The ignored costs are where a node x on a FIND path has a parent y in the same rank group as x, and y is not the last node in the FIND path (in other words, x is not x_{k-1}). The FIND operation gives x a new parent, an ancestor of higher rank than y. If x is in the r-th rank group, then its parent can change at most F_r times before its parent is in a higher rank group, and the contribution of x to subsequent FIND operations is not ignored. Thus the total number of FIND operations in which the contribution of x was not counted is at most F_r.

Summing this up for all nodes in the r-th rank group we get at most $F_r n / F_r = n$ (Lemma 23.15). The zero-th rank group makes no contribution here, so the overall contribution is at most nL. In any case, it is $O(n \log^* n)$. The UNIONs are $O(n)$. Hence

(23.18) Theorem *The overall cost of a mixed sequence S of UNIONs and FINDs, involving m FINDs, is $O((m+n)\log^*(n))$.* ∎