Handout N11
October 9, 2001

Notes for Lecture 11

1 Disjoint Set Union-Find

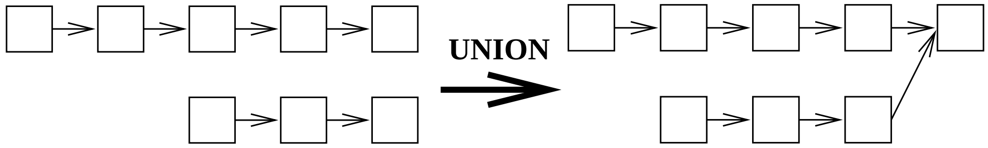
Kruskal's algorithm for finding a minimum spanning tree used a structure for maintaining a collection of disjoint sets. Here, we examine efficient implementations of this structure. It supports the following three operations:

- MAKESET(x) - create a new set containing the single element x.
- UNION(x,y) - replace the two sets containing x and y by their union.
- FIND(x) - return the name of the set containing the element x. For our purposes this will be a *canonical element* in the set containing x.

We will consider how to implement this efficiently, where we measure the cost of doing an *arbitrary* sequence of m UNION and FIND operations on n initial sets created by MAKESET. The minimum possible cost would be $O(m + n)$, i.e. cost $O(1)$ for each call to MAKESET, UNION, or FIND. Our ultimate implementation will be nearly this cheap, and indeed be this cheap for all practical values of m and n.

The simplest implementation one could imagine is to represent each set as a linked list, where we keep track of both the head and the tail. The canonical element is the tail of the list (the final element reached by following the pointers in the other list elements), and UNION simply concatenates lists. In this case FIND has maximal cost proportional to the length of the list, since following each pointer costs $O(1)$, and UNION has cost $O(1)$, to point the tail of one set to the head of the other. The worst case cost is attained by doing n UNIONs, to get a single set, and then m FINDs on the head of the list, for a total cost of $O(mn)$, much larger than our target $O(m + n)$.

To do a better job, we need a more clever data structure. Let us think about how to improve the above simple one. First, instead of taking the union by concatenating lists, we simply make the tail of one list point to the tail of the other, as illustrated below. That way the maximum cost of FIND on any element of the union will have cost proportional to the maximum of the two list lengths (plus one, if both have the same length), rather than the sum.



More generally, we see that a sequence of UNIONs will result in a tree representing each set, with the root of the tree as the canonical element. To simplify coding, we will mark the root by setting the pointer in the root to point to itself. This leads to the following *initial* implementations of MAKESET and FIND:

```
procedure MAKESET(x) ...  initial implementation
  p(x) := x

function FIND(x) ...  initial implementation
  if x ≠ p(x) then return FIND(p(x))
  else return x
```

It is convenient to add a fourth operation LINK(x,y) where x and y are required to be two roots. LINK changes the parent pointer of one of roots, say x, and makes it point to y. It returns the root of the composite tree y. Then UNION(x,y) = LINK(FIND(x), FIND(y)).

But this by itself is not enough to reduce the cost; if we are so unlucky as to make the root of the bigger tree point to the root of the smaller tree, n UNION operations can still lead to a single chain of length n, and the same cost as above.

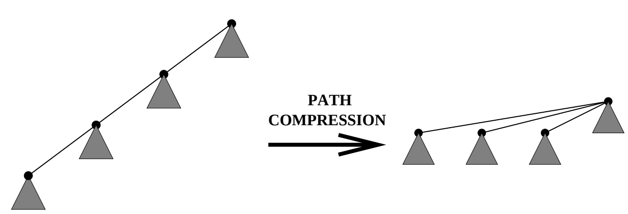
This motivates the first of our two heuristics: UNION BY RANK. This simply means that we keep track of the depth (or RANK) of each tree, and make the shorter tree point to the root of the taller tree; code is shown below. Note that if we take the UNION of two trees of the same RANK, the RANK of the UNION is one larger than the common RANK, and otherwise equal to the max of the two RANKs. This will keep the RANK of tree of n nodes from growing past $O(\log n)$, but m UNIONs and FINDs can then still cost $O(mlogn)$.

```
procedure MAKESET(x) ...  final implementation
  p(x) := x
  RANK(x) := 0

function LINK(x,y)
  if RANK(x) > RANK(y) then swap x and y
  if RANK(x) = RANK(y) then RANK(y) = RANK(y) + 1
  p(x) := y
  return(y)
```

The second heuristic, PATH COMPRESSION, is motivated by observing that since each FIND operation traverses a linked list of vertices on the way to the root, one could make *later* FIND operations cheaper by making each of these vertices point directly to the root:

```
function FIND(x) ...  final implementation
  if x ≠ p(x) then
    p(x) := FIND(p(x))
    return(p(x))
  else return(x)
```



We will prove below that any sequence of m UNION and FIND operations on n elements take at most $O((m+n)\log^* n)$ steps, where $\log^* n$ is the number of times you must iterate the log function on n before you get a number less than or equal to 1. Recall that $\log^* n <= 5$ for all $n \le 2^{2^{16}} = 2^{65536} \approx 10^{19728}$. Since the number of atoms in the universe is estimated to be at most 10^{80}, which is a conservative upper bound on the size of any computer memory (as long each bit is at least the size of an atom), it is unlikely that you will ever have a graph with this many vertices, so $\log^* n \le 5$ in practice.

2 Complexity Analysis of Disjoint Set Union-Find

First a few observations about $RANK$:

1. if $v \neq p(v)$ then $RANK(p(v)) > RANK(v)$.
2. whenever $p(v)$ is updated, $RANK(p(v))$ increases.
3. The number of elements of $RANK$ k is at most $\frac{n}{2^k}$.
4. The number of elements of $RANK \ge k$ is at most $\frac{n}{2^{k-1}}$.

Observation 3 follows from the fact that the RANK of an element v changes only if LINK(v,w) is executed and $RANK(w) = RANK(v)$, and if the operation causes w to point to v. In this case $RANK(v)$ is incremented by 1. Now it is easy to see by induction that if $RANK(v)$ is incremented to k, then the tree rooted at v contains at least 2^k elements, thus proving the observation.

Observation 4 follows from Observation 3: the number of elements of $RANK \ge k$ is at most $\sum_{j=k}^{\infty} \frac{n}{2^j} = \frac{n}{2^{k-1}}$.

Observation 3 also implies that the maximum RANK that an element can have is $\log n$.

Note that as soon as an element becomes a non-root vertex, its RANK is forever fixed. Now let us divide the (non-root) elements into groups according to their RANKs: we will assign to group i all vertices whose RANK r satisfies $\log^* r = i$. Thus each group will consist of all vertices with RANKs in the interval $(k, 2^k]$ where k is itself an iterated power of 2.

It is easy to establish the following assertions about these groups:

1. The number of distinct groups is at most $\log^* n$.
2. The number of elements in the group $(k, 2^k]$ is at most $\frac{n}{2^{k-1}}$.

Let us assign 2^k tokens to each element in group $(k, 2^k]$. Then the total number of tokens assigned to all the elements in that group is at most $2^k \frac{n}{2^{k-1}} = 2n$. Moreover, since the total number of groups is at most $\log^* n$, the number of tokens assigned to all elements in all groups is at most $2n \log^* n$.

Recall that the number of steps for a FIND operation is proportional to the number of pointers that the FIND operation must follow up the tree. We pay for each pointer (u, v) that FIND chases as follows:

- if u and v belong to different groups then FIND pays for chasing this pointer.

- if u and v belong to the same group then u pays using one of its tokens.

The main point here is that each time a FIND operation goes through an element u, its parent pointer is changed to the root of the current tree (due to path compression), and therefore $RANK(p(u))$ increases by at least 1. If u is in the group $(k, 2^k]$, then the RANK of the parent can increase fewer than 2^k times before it moves up to a higher group. Therefore the 2^k tokens assigned to u are sufficient to pay for all the FIND operations that go through u and stay in the same group.

Finally, the total number of steps for m UNION and FIND operations on n elements can be estimated as follows: since LINK requires $O(1)$ steps, and a UNION is implemented using two FIND operations and a LINK operation, we just need to upper-bound the time for $2m$ FIND operations. The total number of steps charged each FIND operation is at most $\log^* n$ for a total of $O(m \log^* n)$. The total number of tokens is at most $n \log^* n$, and each token pays for a constant number of steps. Therefore the total number of steps is $O((m + n) \log^* n)$.