UC Berkeley—CS 170: Efficient Algorithms and Intractable Problems | Handout 9
Lecturer: Michael Jordan | October 3, 2005

Notes 9 for CS 170

1 Disjoint Set Union-Find

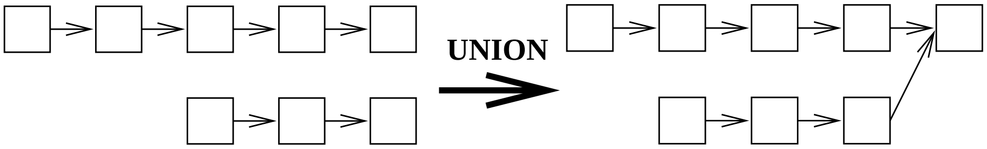
Kruskal's algorithm for finding a minimum spanning tree used a structure for maintaining a collection of disjoint sets. Here, we examine efficient implementations of this structure. It supports the following three operations:

- 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 $x \neq p(x)$ 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.

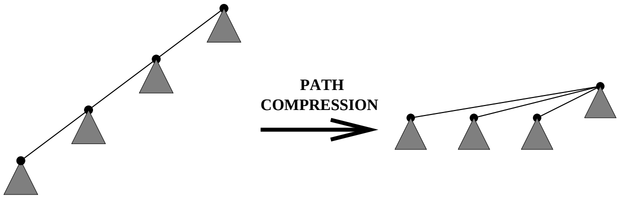
This motivates the first of our two heuristics: UNION BY RANK. This simply means that we keep track of the depth (or RANK) of each tree, and make the shorter tree point to the root of the taller tree; code is shown below. Note that if we take the UNION of two trees of the same RANK, the RANK of the UNION 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 $x \neq p(x)$ 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 $\log^* n \leq 5$ for all $n \leq 2^{2^{16}} = 2^{65536} \approx 10^{19728}$. 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 $\log^* n \leq 5$ in practice.

2 Analysis of Union-Find

Suppose we initialize the data structure with n `makeset` operations, so that we have n elements each forming a different set of size 1, and let us suppose we do a sequence of k operations of the type `union` or `find`. We want to get a bound on the total running time to perform the k operations. Each `union` performs two `find` and then does a constant amount of extra work. So it will be enough to get a bound on the running time needed to perform $m \leq 2k$ `find` operations.

Let us consider at how the data structure looks at the end of all the operations, and let us see what is the rank of each of the n elements. First, we have the following result.

Lemma 1
If an element has rank k, then it is the root of a subtree of size at least 2^k.
Proof: An element of rank 0 is the root of a subtree that contains at least itself (and so is of size at least 1). An element u can have rank $k+1$ only if, at some point, it had rank k and it was the root of a tree that was joined with another tree whose root had rank k. Then u became the root of the union of the two trees. Each tree, by inductive hypothesis was of size at least 2^k, and so now u is the root of a tree of size at least 2^{k+1}. □

Let us now group our n elements according to their final rank. We will have a group 0 that contains elements of rank 0 and 1, group 1 contains elements of rank 2, group 2 contains elements of rank in the range $\{3, 4\}$, group 3 contains elements of rank between 5 and 16, group 4 contains elements of rank between 17 and 2^{16} and so on. (In practice, of course, no element will belong to group 5 or higher.) Formally, each group contains elements of rank in the range $(k, 2^k]$, where k itself is a power of a power ... of a power of 2. We can see that these groups become sparser and sparser.

Lemma 2
No more than $n/2^k$ elements have rank in the range $(k, 2^k]$.
Proof: We have seen that if an element has rank r, then it is the root of a subtree of size at least 2^r. It follows that there cannot be more than $n/2^r$ elements of rank r. The total number of elements of rank between $k+1$ and 2^k is then at most

$$n \sum_{r=k+1}^{2^k} \frac{1}{2^r} < n \sum_{r=k+1}^{\infty} \frac{1}{2^r} = \frac{n}{2^k} \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{n}{2^k}$$

□

By definition, there are no more than $\log^* n$ groups.

To compute the running time of our m operations, we will use the following trick. We will assign to each element u a certain number of "tokens," where each token is worth $O(1)$ running time. We will give out a total of $n \log^* n$ tokens.

We will show that each `find` operation takes $O(\log^* n)$ time, plus some additional time that is paid for using the tokens of the vertices that are visited during the `find` operation. In the end, we will have used at most $O((m+n)\log^* n)$ time.

Let us define the token distribution. If an element u has (at the end of the m operations) rank in the range $(k, 2^k]$ then we will give (at the beginning) 2^k tokens to it.

LEMMA 3
We are distributing a total of at most $n \log^* n$ tokens.
PROOF: Consider the group of elements of rank in the range $(k, 2^k]$: we are giving 2^k tokens to them, and there are at most $n/2^k$ elements in the group, so we are giving a total of n tokens to that group. In total we have at most $\log^* n$ groups, and the lemma follows. □

We need one more observation to keep in mind.

LEMMA 4
At any time, for every u that is not a root, $rank[u] < rank[p[u]]$.
PROOF: After the initial series of `makeset`, this is an invariant that is maintained by each `find` and each `union` operation. □

We can now prove our main result

THEOREM 5
Any sequence of operations involving m `find` operations can be completed in $O((m+n)\log^* n)$ time.

PROOF: Apart from the work needed to perform `find`, each operation only requires constant time (for a total of $O(m)$ time). We now claim that each `find` takes $O(\log^* n)$ time, plus time that is paid for using tokens (and we also want to prove that we do not run out of tokens).

The accounting is done as follows: the running time of a `find` operation is a constant times the number of pointers that are followed until we get to the root. When we follow a pointer from u to v (where $v = p[u]$) we charge the cost to `find` if u and v belong to different groups, or if u is a root, or if u is a child of a root; and we charge the cost to u if u and v are in the same group (charging the cost to u means removing a token from u's allowance) . Since there are at most $\log^* n$ groups, we are charging only $O(\log^* n)$ work to `find`. How can we make sure we do not run out of coins? When `find` arrives at a node u and charges u, it will also happen that u will move up in the tree, and become a child of the root (while previously it was a grand-child or a farther descendent); in particular, u now points to a vertex whose rank is larger than the rank of the vertex it was pointing to before. Let k be such that u belongs to the range group $(k, 2^k]$, then u has 2^k coins at the beginning. At any time, u either points to itself (while it is a root) or to a vertex of higher rank. Each time u is charged by a `find` operation, u gets to point to a parent node of higher and higher rank. Then u cannot be charged more than 2^k time, because after that the parent of u will move to another group. □