COMPSCI 342: Data Structures and Algorithms

Data Structures for the Disjoint Sets Problem

We want to maintain n distinct elements as a collection of disjoint sets. Each set is identified by a *representative*, which is some member of the set. The three operations are:

MAKESET(x) Create a new set whose only member is pointed to by x. Since the sets are disjoint we require that x not already be in a pre-existing set.

FIND(x) Return a pointer to the representative of the unique set containing x.

UNION(x, y) Unite the dynamic sets represented by x and y into a new set that is the union of the two original sets. The original sets are destroyed.

We always have n MAKESET() operations, at most $n-1$ UNION() operations and have m FIND() operations, where $m \geq n$. The goal is to design a data structure that will support this sequence of the three operations as efficiently as possible. The runtime will be analyzed for the whole sequence of operations rather than one operation at a time. (This is an example of *amortized analysis.*)

Attempt I

The elements are from the set $1, 2, \ldots, n$.
set[1..n]: *set*[i] is the label of the set to which element i belongs.

The array *set*[1..n] is the only data structure that this initial attempt uses.

```
MAKESET($x$)
1. set[$x$] $\leftarrow x$
```

```
FIND1($x$)
1. return set[$x$]
```

```
UNION1($x$,$y$)
1. for $k \leftarrow 1$ to $n$
2. do if set[$k$] $= y$
3.       then set[$k$] $\leftarrow x$
```

The worst-case runtime is $\Theta(m + n^2)$.

Attempt II

Represent each set by a linked list. Each cell has a next pointer and a pointer to the beginning of the list. The label of a set is the label of the first element in the linked list.

Pseudo-code formulation left as an exercise.

Attempt III

Use trees to represent sets. We will use the parent-pointer representation of trees. This makes union really simple but find becomes harder.

Analogy: *Someone changing addresses— instead of notifying everyone, it is simpler to leave a forwarding address. But this means that finding the right address takes more steps.*

parent[1..n]: parent[i] is the parent of element i in the tree representing the set to which element i belongs.

```
MAKESET($x$)
1. parent[$x$] $\leftarrow x$
```

```
FIND3($x$)
1. if parent[$x$] $\neq x$
2. then return FIND3(parent[$x$])
3. else return $x$
```

```
UNION3($x$,$y$)
1. parent[$y$] $\leftarrow x$
```

The worst-case runtime is $\Theta(mn)$.

Attempt IV

Now we attempt to balance the trees with the heuristic that the smaller tree is merged with the larger tree. We keep track of the number of elements in the tree in the array *size*[1..n].

```
MAKESET($x$)
1. parent[$x$] $\leftarrow x$ size[$x$] $\leftarrow 1$ 2.
```

```
UNION4($x$,$y$)
1. if size[$x$] $\leq$ size[$y$]
2. then parent[$x$] $\leftarrow y$
3.          size[$y$] $\leftarrow$ size[$x$] + size[$y$]
4. else
5.          parent[$y$] $\leftarrow x$
6.          size[$x$] $\leftarrow$ size[$x$] + size[$y$]
```

Another balancing strategy is to use the heights of the trees to perform the union. Suppose we keep track of the heights of the trees in the array *height*[1..n]. Then we can make the shorter tree be a subtree of the taller tree. The height only increases if the two trees being union-ed have the same height. The pseudo-code is left as an exercise.

The worst-case runtime for either balancing technique is $\Theta(m \lg n)$.

Attempt V

Analogy: *If several changes of addresses occur, then the mail has to go hopping around each time. At some point, it would be a good idea to notify all the forwarding stations about the final destination, so they can forward the mail directly.*

With path-compression, we change the parent pointer of all elements encountered during a find to point directly to the root of the tree. The next find procedure is as follows.

```
FIND4($x$)
1. $root \leftarrow x$
2. while $root \neq$ parent[$root$]
3. do $root \leftarrow$ parent[$root$]
4. i $\leftarrow x$
5. while ($i \neq root$)
6. do $j \leftarrow$ parent[$i$]
7.     parent[$i$] $\leftarrow root$
8.     $i \leftarrow j$
9. return $root$
```

Here is an elegant recursive version:

```
FIND4($x$)
1. if parent[$x$] $\neq x$
2. then parent[$x$] $\leftarrow$ FIND4(parent[$x$])
3. return parent[$x$]
```

We can avoid the two passes by halving the path as we move up to the root. The idea is to make an element point to its grandparent instead of parent as we move up to the root.

The pseudo-code for path-halving technique is left as an exercise.

The worst-case runtime for the sequence of operations is $O(m \lg^* n)$, which is almost linear for any practical scenario. See your textbook for the definition of the iterated logarithm ($\lg^* n$) function (in Chapter 3).

Applications

- Handling COMMON, EQUIVALENCE statements in FORTRAN.
- Computational geometry problems.
- Equivalence of finite state machines.
- Performing unification in logic programming.
- Several graph algorithms.

Implementation

Check out the directory ~`amit/cs342/lab/DisjointSets` on the machine `onyx` for a sample implementation. Read the `README` file for more details.