Data Structures for Disjoint Sets

Some applications involve grouping n distinct objects into a collection of disjoint sets. Two important operations are then finding which set a given object belongs to and uniting the two sets.

A **disjoint set data structure** maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint *dynamic* sets. Each set is identified by a representative, which usually is a member in the set.

1 Operations on Sets

Let x be an object. We wish to support the following operations.

- MAKE-SET(x) creates a new set whose only member is pointed by x; Note that x is not in the other sets.
- UNION(x, y) unites two dynamic sets containing objects x and y, say S_x and S_y, into a new set that $S_x \cup S_y$, assuming that $S_x \cap S_y = \emptyset$;
- FIND-SET(x) returns a pointer to the representative of the set containing x.
- INSERT(a, S) inserts an object a to S, and returns $S \cup \{a\}$.
- DELETE(a, S) deletes an object a from S, and returns $S - \{a\}$.
- SPLIT(a, S) partitions the objects of S into two sets S_1 and S_2 such that $S_1 = \{b \mid b \leq a \ \& \ b \in S\}$, and $S_2 = S - S_1$.
- MINIMUM(S) returns the minimum object in S.

2 Applications of Disjoint-set Data Structures

Here we show two application examples.

- Connected components (CCs)
- Minimum Spanning Trees (MSTs)

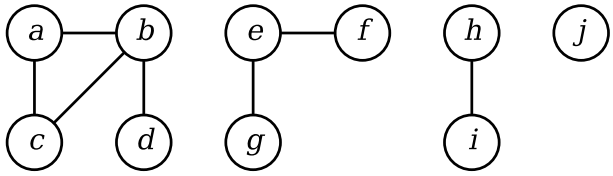

Figure 1: A graph with four connected components: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, and $\{j\}$.

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(e, f)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

Table 1: This table shows the state of the collection of disjoint sets as each edge is processed. The processed edge is listed on the left and the rest of the columns show the state of the collection.

2.1 Algorithm for Connected Components

```
CONNECTED-COMPONENTS(G)
1  for each v ∈ V
2      do MAKE-SET(v)
3  for every edge (u, v) ∈ E
4      do if FIND-SET(u) ≠ FIND-SET(v)
5           then UNION(u, v)
```

```
SAME-COMPONENTS(u, v)
1  if FIND-SET(u) = FIND-SET(v)
2     then return TRUE
3  return FALSE
```

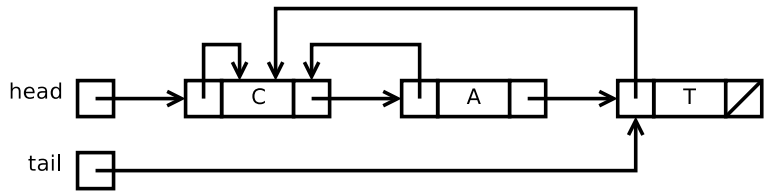

Figure 2: Linked-list representation. Each disjoint set can be represented by a linked-list. Each node has a pointer to the head node and a pointer to the next node.

3 The Disjoint Set Representation

3.1 The Linked-list Representation

A set can be represented by a linked list. In this representation, each node has a pointer to the next node and a pointer to the first node.

Consider the following operation sequence:

MAKE-SET(x_1),
MAKE-SET(x_2),
$\vdots$,
MAKE-SET(x_{n-1}),
MAKE-SET(x_n),
UNION(x_1, x_2),
UNION(x_2, x_3),
$\vdots$,
UNION(x_{n-1}, x_n).

What's the total time complexity of the above operations?

A weighted-union heuristic: Assume that the representative of each set maintains the number of objects in the set, and always merge the smaller list to the larger list, then

Theorem: Using the linked list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, Union, and Find_Set operations, n of which are MAKE-SET, takes $O(m + n \log n)$ time.

Hint: observe that for any $k \leq n$, after x's representative pointer has been updated $\lceil \log k \rceil$ times, the resulting set containing x must have at least k members.

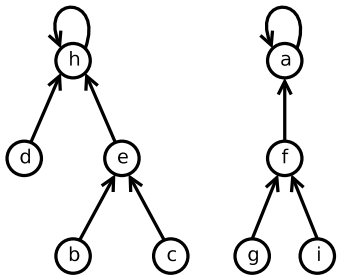

Figure 3: Rooted tree representation of disjoint sets. Each tree has a "representative" h for the left tree and a for the right tree.

4 Disjoint Set Forests

A faster implementation of disjoint sets is through the rooted trees, with each node containing one member and each tree representing one set. Each member in the tree has only one parent.

4.1 The Heuristics for Disjoint Set Operations

1. union by rank.
2. path compression

The idea of **union by rank** is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, for each node we maintain a *rank* that approximates the logarithm of the subtree size and is also an upper bound on the height of the node.

Time complexity: $O(m \log n)$, assuming that there are m union operations.

Path compression is quite simple and very effective. We use this approach during FIND-SET operations to make each node on the path point directly to the root. Path compression does not change any ranks.

Time complexity: $\Theta(f \log_{1+f/n} n)$ if $f \geq n$ and $\Theta(n + f \log n)$ otherwise, assume that there are n MAKE-SET operations and f FIND-SET operations.

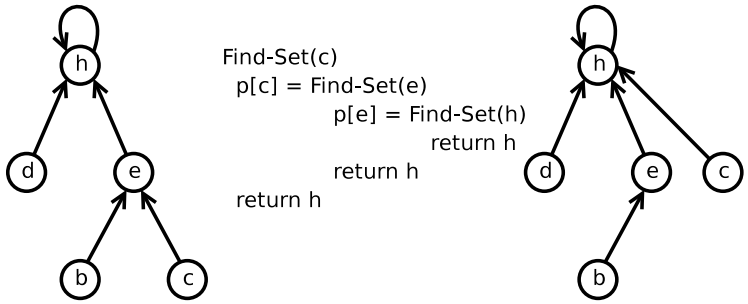

Figure 4: Path compression takes place during the FIND-SET operation. This works by recursing from the given input node up to the root of the tree, forming a *path*. Then the root is returned and assigned as the parent for each node in *path*. The parent of c after FIND-SET(c) is h.

MAKE-SET(x)
1 $p[x] \leftarrow x$
2 $rank[x] \leftarrow 0$

FIND-SET(x)
1 **if** $x \neq p[x]$
2 **then** $p[x] \leftarrow$ FIND-SET($p[x]$)
3 **return** $p[x]$

UNION(x, y)
1 LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1 **if** $rank[x] > rank[y]$
2 **then** $p[y] \leftarrow x$
3 **else** $p[x] \leftarrow y$
4 **if** $rank[x] = rank[y]$
5 **then** $rank[y] \leftarrow rank[y] + 1$

where $rank[x]$ is the height of x in the tree. If both of the above methods are used together, the time complexity is $O(m\alpha(m, n))$.

The Rank properties

- $rank[x] \leq rank[p[x]]$
- for any tree root x, $size(x) \geq 2^{rank[x]}$ (Link operation)
- for any integer r, there are at most $n/2^r$ nodes of rank r
- each node has rank at most $\lfloor \log n \rfloor$, assuming there are at n objects involved.