Maintaining Disjoint Sets

Ref: Chapter 21 of text. (Proofs in Section 21.4 omitted.)

Given:

- A collection $S_1, S_2, \ldots, S_k$ of pairwise disjoint sets.
- Each set has a name, its representative, which is an element of the set.

Note: Any member of the set can serve as the name of a set. The name should not change as long as the set does not change.

Operations:

Note: In the following, $x$ and $y$ are pointers to set elements.

- **MAKE-SET($x$)**: Creates a new set containing the element $x$. (This element also becomes the name of the created set.) The element should not be a member of any other set in the collection.
- **FIND($x$)**: Returns a pointer to the representative of the unique set containing $x$.
- **UNION($x, y$)**: Unions the sets (say, $S_x$ and $S_y$) that contain $x$ and $y$.
  (a) One of the members of $S_x \cup S_y$ is chosen as the name of the new set. (It is common to choose the name of $S_x$ or that of $S_y$ as the name of the new set.)
  (b) After the union, $S_x$ and $S_y$ no longer exist in the collection (because of disjointness).
  (c) A UNION operation uses two FIND operations.

Example: To be presented in class.

Two simple applications:

(a) Assemblers: Several assembly languages allow statements of the following form:

<table>
<thead>
<tr>
<th></th>
<th>EQU</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>EQU</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>EQU</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>EQU</td>
<td>50</td>
</tr>
</tbody>
</table>
The effect is to set all of A, B, C and D to 50. Such statements can be processed using \texttt{UNION} and \texttt{FIND} operations.

A variation: \texttt{FORTRAN} allows the \texttt{EQUIVALENCE} statement:

\begin{verbatim}
EQUIVALENCE X, Y
\end{verbatim}

The effect is that \(X\) and \(Y\) represent the same memory location (aliasing).

(b) Finding connected components: The following algorithm can be used. (Undirected graph is \(G(V, E)\).)

\begin{verbatim}
CONNECTED-COMPONENTS(V, E)
1. for each \(v \in V\) do
    MAKE-SET\((v)\)
2. for each \((u, v) \in E\) do
    if \(\text{FIND}(u) \neq \text{FIND}(v)\)
        then \text{UNION}(u, v)
\end{verbatim}

Now, a query of the form “Are \(u\) and \(v\) in the same component?” can be answered as follows.

\begin{verbatim}
SAME-COMPONENT\((u, v)\)
1. if \(\text{FIND}(u) = \text{FIND}(v)\)
    then return true
  else return false
\end{verbatim}

Example: To be discussed in class.

More complex applications:

- Equivalence of two deterministic finite automata (Hopcroft & Karp, 1971).
- Fast scheduling algorithms (Gabow, 1982).

\textbf{Goal}: To efficiently execute a sequence of \texttt{MAKE-SET}, \texttt{UNION} and \texttt{FIND} operations (i.e., the total time for executing the entire sequence should be asymptotically as small as possible).
Notation:

- \( m \): Total no. of operations in the sequence.
- \( n \): Total no. of \texttt{MAKE-SET} operations in the sequence (i.e., the total number of elements in all the sets).
- \( n_u \): Total no. of \texttt{UNION} operations in the sequence.
- \( n_f \): Total no. of \texttt{FIND} operations in the sequence.

Observations: (a) \( m \geq n \). (b) \( n_u \leq n - 1 \).

Some history:

- Galler & Fischer 1970: Proposed an \( O(m + n \log m) \) implementation. (Later, Fischer developed an \( O(m \log \log n) \) implementation.)
- Rosenkrantz & Stearns, 1971: Developed an \( O(m \log \log n) \) implementation.
- Hopcroft & Ullman, 1973: Developed an \( O(m \log^* n) \) implementation. \((\log^* n \leq 5 \text{ for } n \leq 2^{65,356})\)

- Tarjan, 1975: Improvement to \( O(m \hat{\alpha}(m, n)) \). \((\hat{\alpha}(m, n) \leq 4 \text{ for all practical purposes.})\)

Text: \( O(m \alpha(n)) \) implementation. \((\alpha(n) \leq 4 \text{ for all } n < 10^{80})\)

Some simple implementations:

Idea 1: (Array)

- Assume that \( n \) is known in advance (i.e., the universal set is \( \{1, 2, \ldots, n\} \)).
- Use an array \texttt{SET-OF}[1 .. n] so that \texttt{SET-OF}[i] is the name of the set containing element \( i \).

Running times of operations:

- \texttt{MAKE-SET}(x) : \( O(1) \).
- \texttt{FIND}(x) : \( O(1) \).
- \texttt{UNION}(x, y) : \( O(n) \) (the array needs to be scanned).
Thus, total time to execute a sequence of \( m \) operations is 
\[ O(n + n \cdot n_u + n_f) = O(nm) \] (since \( n_u \leq m \) and \( n_f \leq m \)).

Amortized (or average) time per operation = \( O(n) \).

**Idea 2: (Linked list)**
- Use a linked list for each set (i.e., each element of the set is a node in the list).
- Keep pointers to the first and last nodes of each list (head and tail pointers).
- The first node of each list has the name of the set.
- Keep a “back pointer” from each node to the first node of its list.

Running times of operations:
- **MAKE-SET(\( x \))**: \( O(1) \) time.
- **FIND(\( x \))**:  
  (a) If \( x \) is a pointer to a node, then running time = \( O(1) \).
  (b) If \( x \) is an element, then running time = \( O(n) \).
- **UNION(\( x, y \))**: \( O(n) \) time.
  (a) The sets containing \( x \) and \( y \) can be found in \( O(1) \) or \( O(n) \) time depending on whether \( x \) and \( y \) are pointers or elements.
  (b) Using head and tail pointers, the two linked lists can be joined into a single list in \( O(1) \) time.
  (c) The back pointers of nodes in one of the lists must be changed: \( O(n) \) time.

Thus, total time to execute a sequence of \( m \) operations is
\[ O(n + n \cdot n_u + n_f) \text{ or } O(n + n \cdot n_u + n \cdot n_f) \]
depending on the time for each **FIND** operation.
In either case, total time = \( O(nm) \). Amortized time per operation = \( O(n) \).
Note: When \( n = m/2 \), the total time is \( O(m^2) \). There is a sequence of \( m \) instructions for which the linked list implementation requires \( \Omega(m^2) \) time. (Details in class.)

**An improvement:** Assume that the arguments to the operations are pointers.

**Weighted Union Rule:**
- For each list maintain the number of nodes in the list (i.e., the cardinality of the corresponding set).
- When executing **Union**, update the back pointers for the *smaller* set. (Break ties arbitrarily.)

**Theorem 1:** Using the linked list representation and the weighted union rule, a sequence of \( m \) operations can be executed in \( O(m + n \log n) \) time.

**Proof:** To be discussed in class.

**Exercise:** Specify a sequence of \( m \) operations to show that the time bound of Theorem 1 is tight.

---

**Further improvement:**
- Represent each set as a *rooted tree*.
- Each node of the tree represents an element.
- Each node points to its parent. (The root points to itself.)
- The root gives the name of the set.

**Example:** To be presented in class.

**Running times of operations:**
- **Make-Set**(\( x \)) : Create a tree with just one node (\( x \)). Time: \( O(1) \).
- **Find**(\( x \)) : Starting with \( x \), walk up the tree until the root. Time: \( O(n) \).
- **Union**(\( x, y \)) : Find the roots of the two trees and make the root of one the parent of the other. Time: \( O(n) \).

Total time to execute \( m \) operations = \( O(nm) \). So, the tree by itself doesn’t improve the time.
(a) Union by Rank:
- Basic idea: Make the root of the tree with the smaller number of nodes point to the root of the other tree. (This idea is similar to the Weighted Union Rule for the linked list implementation.)
- To facilitate analysis, use ranks of the two roots instead of the number of nodes in the trees.
- Rank of a node is an upper bound on the height of the node in its tree.

(b) Path Compression:
- Used during FIND.
- Basic idea: Suppose the call FIND(x) uses the path
  \[ x \rightarrow y_1 \rightarrow y_2 \ldots \rightarrow y_r \rightarrow w \]
  where w is the root, then make w the parent of x, y_1, y_2, \ldots, y_r.
- Path compression reduces the time for subsequent FIND operations.
- Path compression does not change the rank of any node.

Pseudocode using the two rules: Handout 12.1.

Exercise: Write a non-recursive version of FIND (with path compression).

Worst-case running times:
- When neither union by rank nor path compression are used: \( \Theta(m^2) \).
- When only union by rank is used: \( \Theta(m \log n) \).
- When only path compression is used: \( \Theta(m \log n) \).
- When both union by rank and path compression are used:
  \[ O(m \log^* n) : \text{Hopcroft \& Ullman, 1973.} \]
  \[ \Theta(m \alpha(m, n)) : \text{Tarjan, 1975.} \]
  \[ O(m \alpha(n)) : \text{Text.} \]

Discussion of \( \alpha(n) \): To be presented in class.