Data Structures

Disjoint Sets (Union Find)
Disjoint-Set

• Also known as “union find”.
• Maintain collection $S = \{S_1, \ldots, S_k\}$ of disjoint **dynamic** (changing over time) sets.
• Each set is identified by a **representative**, which is some member of the set.
• Doesn’t matter which member is the representative,
  – As long as if we ask for the representative twice without modifying the set, we get the same answer both times.
Operations

• **makeSet (x):**
  – make a new set $S_i = \{x\}$, and add $S_i$ to $S$.

• **union (x, y):**
  – if $x \in S_x$, $y \in S_y$, then $S \leftarrow S - S_x - S_y \cup \{S_x \cup S_y\}$.
  – Destroys $S_x$ and $S_y$, since sets must be disjoint.

• **findSet (x):**
  – return representative of set containing $x$. 
Analysis

• Analysis in terms of:
  – $n = \text{number of makeSet operations}$,
  – $m = \text{total number of operations}$.

• Notes:
  – Since makeSet counts toward total number of operations, $m \geq n$.
  – Can have at most $n - 1$ union operations, since after $n-1$ unions, only 1 set remains.
An Example - Connected Components

• For a graph G = (V, E), vertices u, v are in same connected component iff there’s a path between them.
  – Connected components partition vertices into equivalence classes.
An Example - Connected Components

connectedComponents \((V, E)\)
for each vertex \(v \in V\) do
  makeSet(v)
for each edge \((u, v) \in E\) do
  if findSet(u) \neq findSet(v) then
    union(u, v)
sameComponent \((u, v)\)
  if findSet(u) = findSet(v) then
    return true
  else
    return false
Linked List Representation

- Each set is a singly linked list.
- Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object.
- Each set object has pointers head and tail to the first and last objects.
- makeSet:
  - create a singleton list. $O(1)$
- findSet:
  - return pointer to representative. $O(1)$
- union:
  - Append x’s list onto end of y’s list.
Linked List Naive Union

UNION(x, y): append x’s list onto end of y’s list. Use y’s tail pointer to find the end.

- Need to update the representative pointer for every node on x’s list.
- If appending a large list onto a small list, it can take a while.

<table>
<thead>
<tr>
<th>Operation</th>
<th># objects updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION(x₁, x₂)</td>
<td>1</td>
</tr>
<tr>
<td>UNION(x₂, x₃)</td>
<td>2</td>
</tr>
<tr>
<td>UNION(x₃, x₄)</td>
<td>3</td>
</tr>
<tr>
<td>UNION(x₄, x₅)</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>UNION(xₙ₋₁, xₙ)</td>
<td>n − 1</td>
</tr>
</tbody>
</table>

Θ(n²) total

Amortized time per operation = Θ(n).
Weighted-Union Heuristic

• Always append the smaller list to the larger list.
  – A single union can still take $\Omega(n)$ time, e.g., if both sets have $n/2$ members.

• Theorem:
  – With weighted union, a sequence of $m$ operations on $n$ elements takes $O(m + n \log n)$ time.
Weighted-Union Heuristic

• Proof:
  – Each makeSet and findSet still takes $O(1)$.
  – How many times can each object’s representative pointer be updated?
    • It must be in the smaller set each time.

<table>
<thead>
<tr>
<th>times updated</th>
<th>size of resulting set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\geq 2$</td>
</tr>
<tr>
<td>2</td>
<td>$\geq 4$</td>
</tr>
<tr>
<td>3</td>
<td>$\geq 8$</td>
</tr>
<tr>
<td>$k$</td>
<td>$\geq 2^k$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$\lg n$</td>
<td>$\geq n$</td>
</tr>
</tbody>
</table>

  – Therefore, each representative is updated $\leq \lg n$ times.
Disjoint-Set Forest

- Forest of trees.
  - 1 tree per set.
  - Root is representative.
  - Each node points only to its parent.
Operations

• **makeSet:**
  - Make a single-node tree.

• **union:**
  - Make one root a child of the other.

• **findSet:**
  - Follow pointers to the root.
  - Not so good - could get a linear chain of nodes.
Union by height

• Each root stores the height of the tree.
• To merge two trees, make the root with smaller height point to the root with larger height.
• If the height of a tree is $h$, the tree contains at least $2^h$ nodes (proof by induction).
• The height of each tree is $O(\log n)$ and therefore findSet and union take $\Theta(\log n)$ in the worst case.
Path Compression

- Another heuristics to improve running time.
- Define **find path**: nodes visited during findSet on the trip to the root.
- After each findSet, make all nodes on the find path **direct children** of the root.
Union by Rank

• If path compression is used, we cannot maintain the height of the tree in the root.
• Instead, we store in each root a rank value, which is an upper bound on the height of the tree.
• To merge two trees, make the root with smaller rank point to the root with larger rank.
Using the Heuristics

\textbf{makeSet} (x)
\begin{align*}
  & \text{x.p} = \text{x} \\
  & \text{x.rank} = 0
\end{align*}

\textbf{findSet} (x)
\begin{align*}
  & \text{if } \text{x} \neq \text{x.p} \text{ then} \\
  & \quad \text{x.p} = \text{findSet(x.p)} \\
  & \text{return } \text{x.p}
\end{align*}

\textbf{union} (x, y)
\begin{align*}
  & \text{link(findSet(x), findSet(y))}
\end{align*}

\textbf{link} (x, y)
\begin{align*}
  & \text{if } \text{x.rank} > \text{y.rank} \text{ then} \\
  & \quad \text{y.p} = \text{x} \\
  & \text{else} \\
  & \quad \text{x.p} = \text{y} \\
  & \quad \text{// If equal ranks, choose y as parent and} \\
  & \quad \text{// increment its rank.} \\
  & \quad \text{if } \text{x.rank} = \text{y.rank} \text{ then} \\
  & \quad \quad \text{y.rank} = \text{y.rank + 1}
\end{align*}
Running time

• If use both path compression and union by rank, $O(m \alpha(n))$, where

\[
\begin{array}{c|c}
 n & \alpha(n) \\
0-2 & 0 \\
3 & 1 \\
4-7 & 2 \\
8-2047 & 3 \\
2048-A_4(1) & 4 \\
\end{array}
\]

– $A_4(1)$ is $\gg 10^{80}$ (the number of atoms in observable universe).

• This bound is **tight**: 
  – There is a sequence of operations that takes $(m \alpha(n))$ time.