Lecture Notes for Chapter 21: Data Structures for Disjoint Sets

Chapter 21 overview

Disjoint-set data structures

- Also known as “union find.”
- Maintain collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ 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.

[We do not include notes for the proof of running time of the disjoint-set forest implementation, which is covered in Section 21.4.]

Operations

- **MAKE-SET($x$)**: make a new set $S_i = \{x\}$, and add $S_i$ to $\mathcal{S}$.
- **UNION($x$, $y$)**: if $x \in S_x$, $y \in S_y$, then $\mathcal{S} \leftarrow \mathcal{S} - S_x - S_y \cup \{S_x \cup S_y\}$.
  - Representative of new set is any member of $S_x \cup S_y$, often the representative of one of $S_x$ and $S_y$.
  - Destroys $S_x$ and $S_y$ (since sets must be disjoint).
- **FIND-SET($x$)**: return representative of set containing $x$.

Analysis in terms of:

- $n = \# \text{ of elements} = \# \text{ of MAKE-SET operations},$
- $m = \text{ total } \# \text{ of operations}.$
Analysis:

- Since MAKE-SET counts toward total # of operations, \( m \geq n \).
- Can have at most \( n - 1 \) UNION operations, since after \( n - 1 \) UNIONs, only 1 set remains.
- Assume that the first \( n \) operations are MAKE-SET (helpful for analysis, usually not really necessary).

Application: dynamic connected components.

For a graph \( G = (V, E) \), vertices \( u, v \) are in same connected component if and only if there’s a path between them.

- Connected components partition vertices into equivalence classes.

\[
\text{CONNECTED-COMPONENTS}(V, E) \\
\text{for each vertex } v \in V \\
\quad \text{do MAKE-SET}(v) \\
\text{for each edge } (u, v) \in E \\
\quad \text{do if } \text{FIND-SET}(u) \neq \text{FIND-SET}(v) \\
\quad \quad \text{then UNION}(u, v) \\
\text{SAME-COMPONENT}(u, v) \\
\text{if } \text{FIND-SET}(u) = \text{FIND-SET}(v) \\
\quad \text{then return } \text{TRUE} \\
\quad \text{else return } \text{FALSE}
\]

Note: If actually implementing connected components,

- each vertex needs a handle to its object in the disjoint-set data structure,
- each object in the disjoint-set data structure needs a handle to its vertex.

Linked list representation

- Each set is a singly linked list.
- Each list node has fields for
  - the set member
  - pointer to the representative
  - next
- List has head (pointer to representative) and tail.

MAKE-SET: create a singleton list.

FIND-SET: return pointer to representative.

UNION: a couple of ways to do it.

1. 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>

Amortized time per operation = \( \Theta(n) \).

2. **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 \( O(m + n \lg n) \) time.

**Sketch of proof** Each MAKE-SET and FIND-SET 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>≥ 2</td>
</tr>
<tr>
<td>2</td>
<td>≥ 4</td>
</tr>
<tr>
<td>3</td>
<td>≥ 8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( k )</td>
<td>≥ ( 2^k )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \lg n )</td>
<td>≥ ( n )</td>
</tr>
</tbody>
</table>

Therefore, each representative is updated \( \leq \lg n \) times. \( \blacksquare \) (theorem)

Seems pretty good, but we can do much better.

**Disjoint-set forest**

Forest of trees.

• 1 tree per set. Root is representative.
• Each node points only to its parent.
• **MAKE-SET**: make a single-node tree.
• **UNION**: make one root a child of the other.
• **FIND-SET**: follow pointers to the root.

Not so good—could get a linear chain of nodes.

**Great heuristics**

• *Union by rank*: make the root of the smaller tree (fewer nodes) a child of the root of the larger tree.
  • Don’t actually use *size*.
  • Use *rank*, which is an upper bound on height of node.
  • Make the root with the smaller rank into a child of the root with the larger rank.

• **Path compression**: *Find path* = nodes visited during FIND-SET on the trip to the root. Make all nodes on the find path direct children of root.

```plaintext
MAKE-SET(x)
p[x] ← x
rank[x] ← 0

UNION(x, y)
LINK(FIND-SET(x), FIND-SET(y))
```
LINK\((x, y)\)

if \(\text{rank}[x] > \text{rank}[y]\)
   \(\text{then } p[y] \leftarrow x\)
else \(p[x] \leftarrow y\)
   \(\triangleright \text{If equal ranks, choose } y \text{ as parent and increment its rank.}\)
     if \(\text{rank}[x] = \text{rank}[y]\)
     \(\text{then } \text{rank}[y] \leftarrow \text{rank}[y] + 1\)

FIND-SET\((x)\)
if \(x \neq p[x]\)
   \(\text{then } p[x] \leftarrow \text{FIND-SET}(p[x])\)
return \(p[x]\)

FIND-SET makes a pass up to find the root, and a pass down as recursion unwinds to update each node on find path to point directly to root.

**Running time**

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

<table>
<thead>
<tr>
<th>(n)</th>
<th>(\alpha(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4–7</td>
<td>2</td>
</tr>
<tr>
<td>8–2047</td>
<td>3</td>
</tr>
<tr>
<td>2048–(A_4(1))</td>
<td>4</td>
</tr>
</tbody>
</table>

What’s \(A_4(1)\)? See Section 21.4, if you dare. It’s \(\gg 10^{80} \approx \text{# of atoms in observable universe.}\)

This bound is tight—there is a sequence of operations that takes \(\Omega (m \alpha(n))\) time.