COP 4531
Complexity & Analysis of Data Structures & Algorithms

Lecture 8
Data Structures for Disjoint Sets

Thanks to the text authors who contributed to these slides
Data Structures for Disjoint Sets

• 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.
• It 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

• **MAKE-SET(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
  \[
  S = S - S_x - S_y \cup \{S_x \cup S_y\}.
  \]
  - Representative of new set is any member of \( S_x \cup S_y \), often \( x \) or \( y \).
  - Destroys \( S_x \) and \( S_y \) (since sets must be disjoint).

• **FIND-SET(x):** returns representative of set containing \( x \).

• **Analysis in terms of:**
  • \( n = \# \) of elements = \# of MAKE-SET operations,
  • \( m = \) total \# 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).
Example Application: Dynamic connected components

Connected-Components(G)
for each vertex $v \in G.V$
  Make-Set(v)
for each edge $(u, v) \in G.E$
  if Find-Set(u) $\neq$ Find-Set(v)
    Union(u, v)

Same-Component(u, v)
if Find-Set(u) == Find-Set(v)
  return True
else return False

• When 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
Example

(a)

<table>
<thead>
<tr>
<th>Edge processed</th>
<th>Collection of disjoint sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial sets</td>
<td>{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}</td>
</tr>
<tr>
<td>(b,d)</td>
<td>{a} {b,d} {c} {e} {f} {g} {h} {i} {j}</td>
</tr>
<tr>
<td>(e,g)</td>
<td>{a} {b,d} {c} {e,g} {f} {h} {i} {j}</td>
</tr>
<tr>
<td>(a,c)</td>
<td>{a,c} {b,d} {e,g} {f} {h} {i} {j}</td>
</tr>
<tr>
<td>(h,i)</td>
<td>{a,c} {b,d} {e,g} {f} {h,i} {j}</td>
</tr>
<tr>
<td>(a,b)</td>
<td>{a,b,c,d} {e,g} {f} {h,i} {j}</td>
</tr>
<tr>
<td>(e,f)</td>
<td>{a,b,c,d} {e,f,g} {h,i} {j}</td>
</tr>
<tr>
<td>(b,c)</td>
<td>{a,b,c,d} {e,f,g} {h,i} {j}</td>
</tr>
</tbody>
</table>
Implementing Union-Find: Linked list representation

• Each set is a singly linked list, represented by an object with attributes
  - head: the first element in the list, assumed to be the set’s representative, and
  - tail: the last element in the list.

• Objects may appear within the list in any order.

• Each object in the list has attributes for
  - the set member,
  - pointer to the set object, and
  - next.
Sets using linked lists and basic implementation of the union operation

(a) $S_1$

(b) $S_1$
Analysis of complexity of operations

- **Make-Set(x):** create a singleton list
  - complexity: $\Theta(1)$
- **Find-Set(x):** follow vertex $x$ pointer back to the list object, and then follow the head pointer to the representative
  - complexity: $\Theta(1)$
- **Union(x, y):** append $y$'s list onto end of $x$'s list. Use $x$'s tail pointer to find the end
  - Need to update the pointer back to the set object for every node on $y$'s list.
  - If appending a large list onto a small list, it can take a while.
  - What is worst case complexity for $n$ union operations?
  - Turns out to be $\Theta(n^2)$ for $n$ operations or an amortized complexity of $\Theta(n)$ per union operation
Worst case for n union operations can be $\Theta(n^2)$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of objects updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-SET($x_1$)</td>
<td>1</td>
</tr>
<tr>
<td>MAKE-SET($x_2$)</td>
<td>1</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>MAKE-SET($x_n$)</td>
<td>1</td>
</tr>
<tr>
<td>UNION($x_2$, $x_1$)</td>
<td>1</td>
</tr>
<tr>
<td>UNION($x_3$, $x_2$)</td>
<td>2</td>
</tr>
<tr>
<td>UNION($x_4$, $x_3$)</td>
<td>3</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>UNION($x_n$, $x_{n-1}$)</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>
Weighted-union heuristic

- Always append the smaller list to the larger list (break ties arbitrarily)
- A single union can still take $\Omega(n)$
  - consider case if both sets have $n/2$ members
- However, a sequence of $m$ operations on $n$ elements takes $O(m + n \lg n)$ time
  - Why? Each Make-set and Find-set still takes $O(1)$.
  - But an object’s representative updated by union must always be in the smaller set each time of the update.
  - After the update it must be in a set of twice the size
  - Thus each representative can only be updated $\leq \lg n$ times.
Can we still do better?
Yes, with a Disjoint-set forest representation

• First Try
  - 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 child of the other
  - Find-Set: follow pointers to the root
Problem with naïve implementation of Disjoint-set forests

- Note that Union is $\Theta(1)$ but Find-Set is not. What is Find-Set’s complexity?
- Well, the worst case is if we create a linear chain of nodes.
We do better using two heuristics

- **Union by rank**: make the root of the smaller tree (fewer nodes) a child of the root of the larger tree
  - we actually use rank, which is an upper bound on the height of a node
  - make root with smaller rank child of root with larger rank
- **Path compression**: Let Find path be the nodes visited during Find-Set.
  - make all nodes on find path direct children of root
Complexity of \( m \) operations on \( n \) elements using disjoin-set forests

- Path compression (shown above) and union by rank together result in:
- Worst case running time is now \( O(m \alpha(n)) \) where \( \alpha(n) \) is a very very slowing growing function of \( n \).
- Effective, it can be viewed as a constant which gives us a running time linear in \( m \)
Summary

• We have two good ways to represent algorithms that use Union-Find
  - linked lists which give us $O(m + n \lg n)$ running time
  - disjoint forests which effectively give us $O(m)$ running time

• Union-Find is used in the spanning tree algorithms that we next explore (Kruskal’s algorithm and Prim’s algorithm)
  - we assume the best representation when needed in analyzing the running time of these algorithms