CSCE 411
Design and Analysis of Algorithms

Set 7: Disjoint Sets
Slides by Prof. Jennifer Welch
Spring 2014
Disjoint Sets Abstract Data Type

- **State**: collection of disjoint dynamic sets
  - number of sets and composition can change, but must always be disjoint
  - each set has representative element, serves as name of the set

- **Operations**:  
  - **Make-Set(x)**: creates \{x\} and adds to collection of sets  
  - **Union(x,y)**: replaces x's set \(S_x\) and y's set \(S_y\) with \(S_x \cup S_y\)  
  - **Find-Set(x)**: returns (pointer to) the representative of the set containing \(x\)
Disjoint Sets Example

- Make-Set(a)
- Make-Set(b)
- Make-Set(c)
- Make-Set(d)
- Union(a,b)
- Union(c,d)
- Find-Set(b) returns a
- Find-Set(d) returns c
- Union(b,d)
Example Use of Disjoint Sets

- Finding all connected components of an undirected graph:

  - input: G = (V,E)
  - for each v in V do Make-Set(v)
  - for each e = (u,v) in E do
    - if Find-Set(u) ≠ Find-Set(v) then
      - Union(u,v)
  - each set in the disjoint sets data structure consists of all the vertices in one connected component
Linked List Representation

- Store set elements in a linked list
  - each list node has a pointer to the next list node
- First list node is set representative (rep)
- Each list node also has a pointer to the first list node (rep)
- Keep external pointers to first list node (rep) and last list node (tail)
Linked List Representation
Linked List Representation

- **Make-Set(x):** make a new linked list containing just a node for x
  - O(1) time

- **Find-Set(x):** given (pointer to) linked list node containing x, follow rep pointer to head of list
  - O(1) time

- **Union(x,y):** append x's list to end of y's list and update all rep pointers in x's old list to point to head of y's list
  - O(size of x's old list) time
Time Analysis

- What is worst-case time for any sequence of Disjoint Set operations, using the linked list representation?
- Let $m$ be number of ops in the sequence
- Let $n$ be number of Make-Set ops (i.e., number of elements)
Expensive Case

- $\text{MS}(x_1), \text{MS}(x_2), \ldots, \text{MS}(x_n)$,
  $\text{U}(x_1, x_2), \text{U}(x_2, x_3), \ldots, \text{U}(x_{n-1}, x_n)$

- Total time is $O(n^2)$, which is $O(m^2)$ since $m = 2n - 1$

- So amortized time per operation is $O(m^2)/m = O(m)$
Linked List with Weighted Union

- Always append smaller list to larger list
- Need to keep count of number of elements in list in rep node
  - call this count the “weight” of the list
- Calculate worst-case time for a sequence of \( m \) operations
- Make-Set and Find-Set operations contribute \( O(m) \) total, since each takes constant time
Analyzing Time for All Unions

- How many times can the rep pointer for an arbitrary node x be updated?
  - First time x's rep pointer is updated, the new set has at least 2 elements
  - Second time x's rep pointer is updated, the new set has at least 4 elements
    - x's set has at least 2 and the other set is at least as large as x's set
Analyzing Time for All Unions

- The maximum size of a set is \( n \) (the number of Make-Set ops in the sequence of ops).
- So \( x \)'s rep pointer can be updated at most \( \log_2 n \) times.
- Thus total time for all unions is \( O(n \log n) \).
- Note style of counting - focus on one element and how it fares over all the Unions.
Amortized Time

- Grand total for sequence is $O(m+n \log n)$
- Amortized cost per Make-Set and Find-Set is $O(1)$
- Amortized cost per Union is $O(\log n)$ since there can be at most $n - 1$ Union ops.
Tree Representation

- Can we improve on the linked list with weighted union representation?
- Use a collection of trees, one per set
- The rep is the root
- Each node has a pointer to its parent in the tree
Tree Representation

```
  a
 / \
d   e
```

```
  b
 / \
  f
```

```
  c
```

Analysis of Tree Implementation

- **Make-Set**: make a tree with one node
  - $O(1)$ time
- **Find-Set**: follow parent pointers to root
  - $O(h)$ time where $h$ is height of tree
- **Union(x,y)**: make the root of x's tree a child of the root of y's tree
  - $O(1)$ time
- So far, no better than original linked list implementation
Improved Tree Implementation

- Use a weighted union, so that smaller tree becomes child of larger tree
  - prevents long chains from developing
  - can show this gives $O(m \log n)$ time for a sequence of $m$ ops with $n$ Make-Sets

- Also do path compression during Find-Set
  - flattens out trees even more
  - can show this gives $O(m \log^* n)$ time!
What is $\log^* n$?

- The number of times you can successively take the log, starting with $n$, before reaching a number that is at most 1

- More formally:
  - $\log^* n = \min\{i \geq 0 : \log^{(i)} n \leq 1\}$
  - where $\log^{(i)} n = n$, if $i = 0$, and otherwise $\log^{(i)} n = \log(\log^{(i-1)} n)$
Examples of log*$n$

<table>
<thead>
<tr>
<th>$n$</th>
<th>log*$n$</th>
<th>why</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$0 &lt; 1$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$1 \leq 1$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>log 2 = 1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>log 3 &gt; 1, log(log 3) &lt; 1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>log(log 4) = log 2 = 1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>log(log 5) &gt; 1, log(log(log 5)) &lt; 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>log(log(log 16)) = log(log 4) = log 2 = 1</td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>log(log(log 17)) &gt; 1, log(log(log(log 17))) &lt; 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>65,536</td>
<td>4</td>
<td>log(log(log(log 65,536))) = log(log(log 16)) = log(log 4) = log 2 = 1</td>
</tr>
<tr>
<td>65,537</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>65,537</td>
<td>5</td>
</tr>
</tbody>
</table>
log*\(n\) Grows Slowly

- For all practical values of \(n\), log*\(n\) is never more than 5.
Make-Set

- **Make-Set(x):**
  - `parent(x) := x`
  - `rank(x) := 0  // used for weighted union`
Union

Union(x, y):

- r := Find-Set(x); s := Find-Set(y)
- if rank(r) > rank(s) then parent(s) := r
- else parent(r) := s
- if rank(r) = rank(s) then rank(s)++
Rank

- gives upper bound on height of tree
- is approximately the log of the number of nodes in the tree

Example:
- MS(a), MS(b), MS(c), MS(d), MS(e), MS(f),
- U(a,b), U(c,d), U(e,f), U(a,c), U(a,e)
End Result of Rank Example
Find-Set

Find-Set(x):

- if $x \neq \text{parent}(x)$ then
- \hspace{1em} parent(x) := \text{Find-Set(parent(x))}
- return parent(x)

Unroll recursion:

- first, follow parent pointers up the tree
- then go back down the path, making every node on the path a child of the root
Find-Set(a)
Amortized Analysis

- Show any sequence of $m$ Disjoint Set operations, $n$ of which are Make-Sets, takes $O(m \log^* n)$ time with the improved tree implementation.
- Use aggregate method.
- Assume Union always operates on roots
  - otherwise analysis is only affected by a factor of 3
Charging Scheme

- Charge 1 unit for each Make-Set
- Charge 1 unit for each Union

- Set 1 unit of charge to be large enough to cover the actual cost of these constant-time operations
Charging Scheme

- Actual cost of Find-Set(x) is proportional to number of nodes on path from x to its root.
  - Assess 1 unit of charge for each node in the path (make unit size big enough).

- Partition charges into 2 different piles:
  - block charges and
  - path charges
Overview of Analysis

- For each Find-Set, partition charges into block charges and path charges.
- To calculate all the block charges, bound the number of block charges incurred by each Find-Set.
- To calculate all the path charges, bound the number of path charges incurred by each node (over all the Find-Sets that it participates in).
Blocks

- Consider all possible ranks of nodes and group ranks into blocks
- Put rank $r$ into block $\log^* r$

| ranks: 0 | 1 | 2 | 3 | 4 | 5 | ... | 16 | 17 | ... | 65536 | 65537 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>blocks: 0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Charging Rule for Find-Set

- **Fact:** ranks of nodes along path from x to root are strictly increasing.
- **Fact:** block values along the path are non-decreasing
- **Rule:**
  - root, child of root, and any node whose rank is in a different block than the rank of its parent is assessed a block charge
  - each remaining node is assessed a path charge
Find-Set Charging Rule Figure

- block $b'' > b'$
  - 1 block charge (root)
  - 1 block charge
  - 1 path charge

- block $b' > b$
  - 1 path charge
  - 1 block charge

- block $b$
  - 1 path charge
  - 1 block charge
  - 1 path charge
Total Block Charges

- Consider any Find-Set(x).
- Worst case is when every node on the path from x to the root is in a different block.
- **Fact:** There are at most $\log^* n$ different blocks.
- So total cost per Find-Set is $O(\log^* n)$
- Total cost for all Find-Sets is $O(m \log^* n)$
Total Path Charges

- Consider a node $x$ that is assessed a path charge during a Find-Set.

- Just before the Find-Set executes:
  - $x$ is not a root
  - $x$ is not a child of a root
  - $x$ is in same block as its parent

- As a result of the Find-Set executing:
  - $x$ gets a new parent due to the path compression
Total Path Charges

- x could be assessed another path charge in a subsequent Find-Set execution.
- However, x is only assessed a path charge if it's in the same block as its parent.
- **Fact:** A node's rank only increases while it is a root. Once it stops being a root, its rank, and thus its block, stay the same.
- **Fact:** Every time a node gets a new parent (because of path compression), new parent's rank is larger than old parent's rank.
Total Path Charges

- So $x$ will contribute path charges in multiple `Find-Set` Sets as long as it can be moved to a new parent in the same block.
- Worst case is when $x$ has lowest rank in its block and is successively moved to a parent with every higher rank in the block.
- Thus $x$ contributes at most $M(b)$ path charges, where $b$ is $x$'s block and $M(b)$ is the maximum rank in block $b$. 
Total Path Charges

- **Fact:** There are at most $n/M(b)$ nodes in block $b$.

- Thus total path charges contributed by all nodes in block $b$ is $M(b) \times n/M(b) = n$.

- Since there are $\log^* n$ different blocks, total path charges is $O(n \log^* n)$, which is $O(m \log^* n)$. 
Even Better Bound

By working even harder, and using a potential function analysis, can show that the worst-case time is $O(m\alpha(n))$, where $\alpha$ is a function that grows even more slowly than $\log^*n$:

- for all practical values of $n$, $\alpha(n)$ is never more than 4.
An Application of Disjoint Sets

- In the next section of the course, on greedy algorithms, we will see how the disjoint sets data structure, implemented with weighted union and path compression, makes Kruskal’s minimum spanning tree algorithm very efficient.