CISC 320 Introduction to Algorithms
Fall 2003

Lecture 9
Union-Find

Problem: to maintain dynamic equivalence relations
Applications:
- minimum spanning tree
- Equivalence declarations in Fortran
Equivalence relation $R$ on a set $S$
- $R$ is a binary relation
- $R$ satisfies three properties for all $s, u, t \in S$
  1. Reflexive: $sR\!t$
  2. Symmetric: $sR\!t \iff tR\!s$
  3. Transitive: $sR\!t, tR\!u \iff sR\!u$

Task: a data structure (and algorithms) to support efficient operations w.r.t. equivalence relations, i.e., to represent, modify, and answer certain questions about an equivalence relation that changes during computation.

Operations
1. IS $s_i \equiv s_j$?
2. MAKE $s_i \equiv s_j$ (where $s_i \equiv s_j$ is not already true).

Example: $S = \{1, 2, 3, 4, 5\}$
equivalence classes to start: $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}$
1. IS 2 $\equiv$ 4?  No
2. IS 3 $\equiv$ 5?  No
3. MAKE 3 $\equiv$ 5.  $\{1\}, \{2\}, \{3,5\}, \{4\}$
4. MAKE 2 $\equiv$ 5.  $\{1\}, \{2,3,5\}, \{4\}$
5. IS 2 $\equiv$ 3?  Yes
6. MAKE 4 $\equiv$ 1.  $\{1,4\}, \{2,3,5\}$
7. IS 2 $\equiv$ 4?  No
### Implementations

**Matrix**

- \[ R = \begin{bmatrix}
  1 & 2 & 3 & 4 & 5 \\
  1 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 1 & 0 \\
  0 & 0 & 1 & 0 & 1 \\
  0 & 0 & 1 & 0 & 1 \\
\end{bmatrix} \]

1. \( R(i,j) = 1 \) if \( i \neq j \), \( R(i,j) = 0 \) otherwise
2. IS takes \( O(1) \) time
3. MAKE requires copying rows, may be up to \( O(n^2) \)
4. A sequence of \( m \) MAKES and ISs at worst-case takes \( O(mn) \)
5. Space usage is \( O(n^2) \).

**Array**

- \[ R = \begin{bmatrix}
  1 & 2 & 3 & 4 & 5 \\
\end{bmatrix} \]

1. IS \( i \equiv j \)?
   - Yes, if \( R[i] = R[j] \)
   - No, otherwise
2. MAKE \( i \equiv j \)
   - for \( k = 1 \) to \( n \), if \( R[k] = R[i] \) then \( R[k] = R[j] \)
3. Worst-case, a sequence of \( m \) MAKES and ISs will take \( O(mn) \).
1. IS 2 \equiv 4? No \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}
2. IS 3 \equiv 5? No \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}
3. MAKE 3 \equiv 5. \quad \{1\}, \{2\}, \{3,5\}, \{4\} \quad \begin{bmatrix} 1 & 2 & 5 & 4 & 5 \end{bmatrix}
4. MAKE 2 \equiv 5. \quad \{1\}, \{2,3,5\}, \{4\} \quad \begin{bmatrix} 1 & 5 & 5 & 4 & 5 \end{bmatrix}
5. IS 2 \equiv 3? Yes \quad \begin{bmatrix} 1 & 5 & 5 & 4 & 5 \end{bmatrix}
6. MAKE 4 \equiv 1. \quad \{1,4\}, \{2,3,5\} \quad \begin{bmatrix} 4 & 5 & 5 & 4 & 5 \end{bmatrix}
7. IS 2 \equiv 4? No \quad \begin{bmatrix} 4 & 5 & 5 & 4 & 5 \end{bmatrix}

- Union-Find: makeSet, find and union
  - makeSet is run on each element of S to make n singleton sets.
  - IS and MAKE are implemented as
    
    IS \, s_i \equiv s_j
    \quad t = \text{find}(s_i)
    \quad u = \text{find}(s_j)
    \quad \text{if } (t=u) \text{ then yes else no}

    MAKE \, s_i \equiv s_j
    \quad t = \text{find}(s_i)
    \quad u = \text{find}(s_j)
    \quad \text{union}(t,u)

    Because Union-Find is just an abstract data type, we need to provide concrete implementation.
In-tree representation of disjoint sets

In-tree operations:

- makeNode: construct a tree of one node
- setParent: change the parent of a node
- setNodeData: set an integer data value for the node
- isRoot: return true if the node has no parent
- parent: return the parent of the node
- nodeData: return the data value

Implementation of Union-Find

- makeSet: O(1)
- find: O(d) where d is the tree height
- union:

<table>
<thead>
<tr>
<th>#</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Union(1,2)</td>
</tr>
<tr>
<td>2</td>
<td>Union(2,3)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td>Union(n-1,n)</td>
</tr>
<tr>
<td>n</td>
<td>Find(1)</td>
</tr>
<tr>
<td>n+1</td>
<td>Find(1)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n+m</td>
<td>Find(1)</td>
</tr>
</tbody>
</table>

\[ T = n + (n-1) + (m+1)n = O(nm) \]
- Weighted union
  - Keep tree as short as possible
  - When do union of two trees, make the tree with fewer nodes a subtree of the root of the other tree.
  - At root, assign a weight (total # of nodes, or height of the tree)

```c
wUnion(i, j)
    t = find(i);
    u = find(j);
    if(t.weight < u.weight)
        then union(t,u);
    else union(u,t);
```

<table>
<thead>
<tr>
<th>#</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>wUnion(1,2)</td>
</tr>
<tr>
<td>2</td>
<td>wUnion(2,3)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td>wUnion(n-1,n)</td>
</tr>
<tr>
<td>n</td>
<td>Find(1)</td>
</tr>
<tr>
<td>n+1</td>
<td>Find(1)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n+m</td>
<td>Find(1)</td>
</tr>
</tbody>
</table>

```
T = n + (n-1) + (m+1) = n + (n-1) + 2(n-1) + (m+1) = O(n+m)
```

```c
wUnion  find  union  find  find
```
Lemma 6.6  If trees grow by merging trees via wUnion, then any tree that has k nodes will have height at most $\lfloor \lg k \rfloor$.

Proof: induction on k.
Base case ($k = 1$): a tree with one node has height 0, and $\lg(1) = 0$.
Assume the lemma holds for trees that have size up to any arbitrary $k > 1$.
Let’s merge two trees $T_1$ ($k_1$ nodes) and $T_2$ ($k_2$ nodes).
we have $k_1 \leq k$ and $k_2 \leq k$, but $k_1 + k_2 > k$.
Because both $T_1$ and $T_2$ have nodes less than m, so $h_1 \leq \lg(k_1)$ and $h_2 \leq \lg(k_2)$.
The height $h$ of the new tree $T$ as a result of wUnion($T_1$, $T_2$) is determined as
max($h_1$, $h_2+1$)
where we assume $k_2 \leq k_1$.

$T$ has $k' = (k_1 + k_2)$ nodes, and its height

$h = \max(h_1, h_2+1) < \max(\lg(k_1), \lg(k_2) + 1)$.

Clearly, $\lg(k_2) \leq \lg(k')$. Because $k_2 \leq k'/2$, so $\lg(k_2) \leq \lg(k') - 1$.
Therefore: $h \leq \lg(k')$.  QED

Definition: Because disjoint sets are implemented as in-trees, the in-tree operations are called link operations.

Theorem 6.7  A Union-Find program of size $m$, on a set of $n$ elements, performs

$\Theta(n+m \log(n))$ link operations in the worst case if wUnion is used.

Proof: With $n$ elements, at most $n-1$ wUnions can be done, building a tree with at most $n$ nodes. Trees can not be higher than $\lg(n)$.
Therefore, each find takes at most $\lg(n)$. There can be at most $m$ finds. So the total number of link operations is less than $O(n+m \log(n))$.

It can be shown that, in worst-case, it must take $\Omega(n + m \log(n))$ link operations.
Path Compression
With path compression, find will make every encountered node directly point to the root.

After Find(x)
int cFind(int v)
    int root;
    1. int oldParent = parent[v];
    2. if(oldParent == -1)
        3. root = v;
    4. else
        5. root = cFind(oldParent);
        6. if(oldParent != root)
            7. parent[v] = root;
    8. return root;

Time Analysis: wUnion and cFind

Definition:
The height of node v, also called its rank, is the height of the subtree rooted at v.

Lemma 6.8 In the set S there are at most n/2^r node with rank r, for r ≥ 0.

Proof: Any tree with height r has at least 2^r nodes (Lemma 6.6). Since subtrees with rank r are disjoint, there can be at most n/2^r such subtrees.
Lemma 6.9  No node of S has rank greater than \( \lg(n) \).

Lemma 6.10  The ranks of the nodes on a path from a leaf to a root of a tree form a strictly increasing sequences. When a cFind operation changes the parent of a node, the new parent has higher rank than the old parent of that node.

---

Definition: log-star  
\[
\log^*(j) = \{ \min i \mid \log^{(i)} j \leq 1 \}
\]
where  
\[
\log^{(1)} j = \log j \\
\log^{(2)} j = \log(\log^{(1)} j) = \log(\log(j)) \\
\log^{(3)} j = \log(\log^{(2)} j) = \log(\log(\log(j))) \\
\ldots
\]

If function \( H \) is defined as follows  
\[
H(0) = 1 \\
H(i) = 2^{H(i-1)} \quad \text{for } i > 0.
\]
then \( \log^*(j) \) is the least \( i \) such that \( H(i) \geq j \).

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H(i) )</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>65536</td>
<td>2^{65536}</td>
<td>??</td>
</tr>
<tr>
<td>( \log^*(i) )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>( \log^*(H(i)) )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Definition: Node blocks (called node groups in the text)
If a node has rank $r$, then it belongs to Block $\lg^*(1+r)$

- Block(0) has nodes of rank 0
- Block(1) has nodes of rank 1
- Block(2) has nodes of rank 2 to 3
- Block(3) has nodes of rank 4 to 15
- Block(4) has nodes of rank 16 to 65535
- Block(5) has nodes of rank 65536 to $(2^{65536} - 1)$

...  
- Block(i) has nodes of rank $H(i-1)$ to $H(i) - 1$.

- For the set $S$ of $n$ nodes, how many blocks? $\lg^*(n+1)$

  Since no node in $S$ can rank higher than $\lg(n)$, the maximum block index must be less than $\lg^*(1 + \lg(n)) = \lg^*(\lg(n+1) - 1) = \lg^*(n+1) - 1$.

  And the minimum block index is 0. Therefore, the # of blocks is $\lg^*(n+1)$

---

Amortized Cost

- Why use amortized cost?

  Remember, we use the number of (basic) operations as a measure of running time. However, how can such a measure be useful if same operation (e.g., each cFind) may cost differently, depending when it is applied in a sequence of operations?

- In an amortized analysis, the time required to perform a sequence of data structure operations is averaged over all the operations performed.
  - Amortized cost differs from average-case cost, because there is no probability involved
  - Amortized cost is the average performance of each operation in the worst case.
Three techniques for amortized cost analysis

- **Aggregate method**
  - Amortized cost = total actual cost / number of operations

- **Accounting method**
  - Amortized cost = actual cost + accounting cost
  - The sum of the accounting cost is nonnegative

- **Potential method**
  - Amortized cost = actual cost + increment of potential
  - Potential never below zero

---

**A simple example**

stack operations
- `Push(S, x)`
- `Pop(S)`
- `MultiPop(S, k)`: pop k top objects of S.
  - If S has less than k objects, then pop all in S.

a sequence of n Push, Pop and MultiPop operations on initially empty stack.

Worst-case cost:
- For each operation:
  - Push: O(1)
  - Pop: O(1)
  - MultiPop: O(n) since the stack size is at most n

There are n operations (possibly O(n) MultiPop operations), the upper bound is O(n²).

Problem: O(n²) upper bound is not tight.
- **Aggregate method**
  Although a single MultiPop can be expensive, any sequence of n Push, Pop, and MultiPop operations on an initially empty stack can cost at most $O(n)$.
  Proof: Pop (either directly or from inside MultiPop) can be called n times since there can be at most n objects (by n Push operations).
  The amortized cost of an operation is the average: $O(n)/n = O(1)$

- **Accounting method**
  - Data structure comes with a “bank account”
  - Every operation allotted a fixed $\$ cost (its amortized cost)
  - If actual cost less than allotted amount, the difference is deposited into bank
  - If actual cost more than allotted amount, withdraw from bank to pay for the operation
  - Catch: always have a non-negative balance
  - Benefit: we can use an operation’s amortized cost, which is a fixed number, and we know that n times the amortized cost is the upper bound of the actual cost of n operations.
Actual cost:
- Push: 1
- Pop: 1
- MultiPop: min(k,s)

Amortized cost:
- Push: 2
- Pop: 0
- MultiPop: 0

Will the bank account be balanced?
Yes. A stack of plates in a cafeteria. We start with an empty stack. Push a plate on the stack and pop a plate off the stack cost $1 each. Now, when push, we pay $2. One dollar for the actual cost, and one dollar as a credit. Since every plate on the stack has a dollar of credit on it, pop is free.

Potential method
- Prepaid work as potential that can be released to pay for future operations
- Initial data structure $D_0$, on which n operations are to be performed. $D_i$ is the data structure after ith operation.
- Potential $\Phi$: $D_i \rightarrow \Phi(D_i)$
- Amortized cost $c_i$ of the ith operation is its actual cost plus the increase in potential due to the operation
  $c_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- The total amortized cost of the n operations
  $\sum c_i = \sum (c_i + \Phi(D_i) - \Phi(D_{i-1}))$
  $= \sum (c_i) + \Phi(D_n) - \Phi(D_0)$
Potential $\Phi$: number of objects on the stack.
- $D_0$ is empty stack, and $\Phi(D_0) = 0$
- $\Phi(D_i) \geq 0 = \Phi(D_0)$

Amortized cost:
- **Push**
  $$c_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$
- **MultiPop(S, k)**
  $$c_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$$
  where $k' = \min(k, s)$ is the actual number objects removed from the stack.

**makeSet:**
- Accounting cost is assigned as $4 \lg^*(n+1)$
- Actual cost is 1
- Amortized cost = $1 + 4 \lg^*(n+1)$

**wUnion:**
- Accounting cost is assigned as 0
- Actual cost is 1
- Amortized cost = 1
- **cFind(v)**
  - Path from v to the root contains nodes: \( w_0, w_1, \ldots, w_k \).
  - where \( w_0 \) is v and \( w_k \) is the root.
  - If \( k = 0 \) or \( 1 \), accounting cost is assigned as zero.
  - For \( k \geq 2 \), the accounting cost is \(-2\) for each pair \((w_{i-1}, w_i)\), \(1 \leq i \leq k-1\), if nodes \( w_{i-1} \) and \( w_i \) are in the same block.
  - Actual cost is \( 2k \).
  - Amortized cost = \( 2 \lg^*(n+1) \).
Lemma 6.12  The sum of accounting costs is never negative
Proof:
- The sum of accounting costs of the initial makeSet operations is 4n lg*(n+1)
- -2 is charged for a node w if w is traversed by a cFind and w is in the same block as its parent and the parent is not a root. If w is in block i, then after cFind's traversal, w is assigned a new parent and if this new parent is in block i+1, then w can not be further associated with any negative charge. Therefore, w can not be associated with more withdrawals than there are ranks in its block. The number of ranks in block i is less than H(i).
- The number of withdrawals for all w in S is at most
  \[ \sum_{i=0}^{\lfloor \log^* (n+1) \rfloor} H(i) \] (number of nodes in block i)
- Number of nodes in block i
  \[ = \sum_{r=1}^{H(i)} \frac{(n/2)^r}{r} \leq \sum_{r=1}^{H(i)} \frac{n}{r} \leq n \sum_{r=1}^{H(i)} \frac{1}{r} \approx 2nH(i). \]
- Total withdrawals \[ \leq \sum_{i=0}^{\lfloor \log^* (n+1) \rfloor} H(i) \] (number of nodes in block i)
- Each withdrawal is -2
- The sum of accounting costs is \[ \geq 4n \log^*(n+1) - 4n \log^*(n+1) = 0. \]

Theorem 6.13  The number of link operations done by a Union-Find program implemented with wUnion and cFind, of length m on a set of n elements is in O((n+m) lg*(n)) in the worst case.

Proof: as analyzed above, amortized cost for each Union-Find operation is at most 1+4lg*(n+1). There are n-m operations including makeSets. So the total amortized cost is O((n+m)lg*(n+1)). This is also the upper bound for the total actual cost, because according to Lemma 6.12 the total actual cost never exceeds the total amortized cost.
Summary:
- Since $\log^* n$ grows so slowly, $O((n+m)\log^*(n))$ is slower than $O(n+m \log(n))$. Therefore, improvement is achieved by using both wUnio and cFind.