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: s R s
  2. Symmetric: s R t \iff t R s
  3. Transitive: s R t, t R u \Rightarrow s R 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 \equiv s ?
2. MAKE s \equiv t, (where s \equiv s 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

\[
R = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1
\end{bmatrix}
\]

1. R(i,j) = 1 \iff i \equiv 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
\[
R = [1 2 3 4 5]
\]

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[j] then R[i] = R[j]
3. Worst-case, a sequence of m MAKES and ISs will take O(mn).

- Matrix

\[
R = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1
\end{bmatrix}
\]
IS

MAKE

MAKE

MAKE

MAKE

MAKE

MAKE

Count

Unweighted

Weighted

 Union-Find: makeSet, find and union

In-tree representation of disjoint sets

In-tree operations:

Implementation of Union-Find

Weighted union

Because Union-Find is just an abstract data type, we need to provide concrete implementation.
Lemma 6.6: If trees grow by merging trees via \texttt{wUnion}, then any tree that has \( k \) nodes will have height at most \( \log k \).

Proof: induction on \( k \).

Base case (\( k = 1 \)): a tree with one node has height 0, and \( \log(1) = 0 \).

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 \( k_1 > 1 \) and \( k_2 > 1 \).

Since both \( T_1 \) and \( T_2 \) have nodes less than \( m \), we have \( h_1 \leq \log k_1 \) and \( h_2 \leq \log k_2 \).

The height \( h \) of the new tree \( T \) is determined as

\[
\max(h_1, h_2 + 1)
\]

where we assume \( k_1 < k_2 \).

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

\[
h = \max(h_1, h_2 + 1) < \max(\log k_1, \log k_2) + 1.
\]

Clearly, \( \log k' < \log k_1 \). Because \( k_1 < k_2 \), so \( \log k_2 < \log k' \).

Therefore, \( h < \log k' \).

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

\( \Theta(n \log n) \)

\( \log \) link operations in the worst case if \texttt{wUnion} is used.

Proof: With \( n \) elements, at most \( n-1 \) \texttt{wUnion}s can be done, building a tree with at most \( n \) nodes. Trees cannot be higher than \( \log(n) \).

Therefore, each \texttt{Find} takes at most \( \log(n) \). There can be at most \( m \) \texttt{Find}s. So the total number of link operations is less than \( \Theta(n + m \log(n)) \).

It can be shown that, in worst-case, it must take \( \Omega(n + m \log(n)) \) link operations.

Path Compression

With path compression, \texttt{Find} will make every encountered node directly point to the root.

```
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 \neq root)
7.   parent[v] = root;
8. return root;
```

Time Analysis: \texttt{wUnion} and \texttt{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 \( r \geq 0 \).

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

\[
\log^*(j) = \min \{ i | \log^*(i) \leq j \}
\]

where \( \log^*(1) = \log(j) \)

\[
\log^*(2) = \log(\log^*(1)) = \log(\log(j))
\]

\[
\log^*(3) = \log(\log^*(2)) = \log(\log(\log(j)))
\]

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 \( H(i) \) is at least \( j \).

\[\begin{array}{cccccc}
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
H(i) & 1 & 2 & 4 & 16 & 65536 & \text{max} & \text{??} \\
\log^*(H(i)) & 1 & 1 & 2 & 3 & 4 & 5 & 6
\end{array}\]

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 the 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
**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 $\$1$ 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.

---

**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. $D_0$ is the data structure after 0th operation.
- Potential $\Phi$: $D_i \rightarrow \Phi(D_i)$
- Amortized cost $c_i$ of the $i$th 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**
  $$c_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 0 = 1$$
- **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 of objects removed from the stack.
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 \((w_{i-1}, w_j)\) if nodes \( w_{i-1} \) and \( w_j \) 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.
- -2 is charged for a node w if w is in block \( i \) and 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 \( \mathcal{S} \) is at most \( \sum_{i=0}^{n} \frac{H(i)}{2^{H(i)}} \) (number of nodes in block \( i \))
- The number of nodes in block \( i \) is \( \frac{1}{2^{H(i)}} \) (number of nodes with rank \( i \))
- The number of withdrawals is \( \sum_{i=0}^{n} \frac{H(i)}{2^{H(i)}} \) (number of nodes in block \( i \))
- Total withdrawals \( \sum_{i=0}^{n} \frac{H(i)}{2^{H(i)}} = 2(n \lg^*(n+1)) + 2n \lg^*(n+1) \)
- Each withdrawal is -2
- The sum of accounting costs is 4n \( \lg^*(n+1) \) - 4n \( \lg^*(n+1) \) = 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 operations is at most 1+4\( \lg^*(n+1) \). 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 \( \lg^*n \) grows so slowly, \( O(n+m \lg^*(n)) \) is slower than \( O(n+m \log(n)) \). Therefore, improvement is achieved by using both wUnion and cFind.