CSE 100
Lazy Union and Find with Path Compression
Today’s Lecture

- Lazy Simple Union and Find (using up trees)
- Lazy Smart Union (union by size and union by height) and Find
- Path Compression
- Amortized Cost analysis
Eager Unions vs. Lazy Unions

Eager Union

Lazy Union

Perform these operations:

Union(1, 0) =
Find(4) =
Find(3) =

$O(\log n)$

$O(1)$

Union

find(4) ≠ 0

require to traverse
parent pointers

until root is reached
Array representation of Up-trees

- A compact and elegant implementation
- Each entry is the up index
- -1 for the roots
- Write the forest of trees, showing parent pointers and node labels, represented by this array

Find(4)

\[
\begin{array}{cccccccccc}
\text{parent} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & -1 & -1 & 0 & 1 & 2 & -1 & -1 & 7
\end{array}
\]

\[\text{find}(4) = 0\]
Performing a union operation

Union(6, 7) (Assume first element becomes the parent of the second)

Fill in 6, 7 and 8 in the array

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Disjoint subsets using up-trees (Lazy simple Union and Find)

Start with 4 items: 0, 1, 2, 3

Suppose \( \text{Union}(i, j) \) makes the node \( \text{Find}(i) \) the parent of the node \( \text{Find}(j) \)

Perform these operations:

\[
\begin{align*}
\text{Union}(2, 3) \\
\text{Union}(1, 2) \\
\text{Find}(0) = \\
\text{Find}(3) = \\
\text{Find}(1) = \\
\text{Find}(0) =
\end{align*}
\]

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
-1 & -1 & -1 & -1
\end{array}
\]
Run time of Lazy simple union and find (using up-trees)

If we have no rule in place for performing the union of two up trees, what is the worst case run time of find and union operations in terms of the number of elements N?

A. find: O(N), union: O(1)
B. find: O(1), union: O(N)
C. find: O(\(\log_2 N\)), union: O(1)
D. find: O(1), union: O(\(\log_2 N\))
Improving Union operations

• In the Union operation, a tree becomes like a linked list if, when joining two trees, the root of the smaller tree (e.g. a single node) always becomes the root of the new tree

• We can avoid this by making sure that in a Union operation, the *larger* of the two trees’ roots becomes the root of the new tree (ties are broken arbitrarily)
Smarter Union operations

- Avoid this problem by having the *larger* of the 2 trees become the root
- **Union-by-size**
  - Each root stores the size (# nodes) of its respective tree
  - The root with the larger size becomes the parent
  - Update its size = sum of its former size and the size of its new child
- Break ties arbitrarily
Smarter Union operations

- Union-by-size
- Union-by-height (also called union by rank)
  - Each root stores the height of its respective tree
  - If one root has a greater height than the other, it becomes the parent. Its stored height doesn’t need to be updated
  - If the roots show equal height, pick either one as the parent
  - Its stored height should be increased by one
- Break ties arbitrarily
Disjoint subsets using trees (Union-by-height and simple Find)

Start with 4 items: 0, 1, 2, 3

Assume that $\text{Union}(i,j)$ makes the root of the shorter tree a child of the root of the taller tree.

Perform these operations:

$\text{Union}(2,3)$
$\text{Union}(1,2)$
$\text{Find}(0) =$
$\text{Find}(3) =$
$\text{Union}(0,2)$
$\text{Find}(1) =$
$\text{Find}(0) =$
Cost of disjoint subsets operations with smarter Union and simple Find

Either union-by-size or union-by-height will guarantee that the height of any tree is no more than $\log_2 N$, where $N$ is the total number of nodes in all trees.

Q: What is the worst case run time of find and union operations in terms of the number of elements $N$?

A. find: $O(N)$, union: $O(1)$
B. find: $O(1)$, union: $O(N)$
C. find: $O(\log_2 N)$, union: $O(1)$
D. find: $O(1)$, union: $O(\log_2 N)$
Cost of disjoint subsets operations

Let’s compare all the methods so far for N-1 union operations (the maximum possible) and M find operations

<table>
<thead>
<tr>
<th>Union(single)</th>
<th>Find (single)</th>
<th>N-1 unions and M finds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Eager and simple</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Eager and smart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Lazy and simple union</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Lazy and smart union</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Path-compression

• In a disjoint subsets structure using parent-pointer trees, the basic Find operation is implemented as:
  • Go to the node corresponding to the item you want to Find the equivalence class for
  • Traverse parent pointers from that node to the root of its tree
  • Return the label of the root
  • This has worst-case time cost $O(\log_2 N)$
  • The time cost of doing another Find operation on the same item is the same
Path-compression

• The path-compression Find operation is implemented as:
  • Go to the node corresponding to the item you want to Find the equivalence class for
  • Traverse parent pointers from that node to the root of its tree
  • Return the label of the root
  • But …. 
Example of path compression

... as part of the traversal to the root, change the parent pointers of every node visited, to point to the root of the tree (all become children of the root) worst-case time cost is still $O(\log N)$

find(e)

Henry Kautz
Cost of find with path compression

What is the time cost of doing another Find operation on the same item, or on any item that was on the path to the root?

A. $O(\log N)$
B. $O(N)$
C. $O(1)$
Disjoint subsets using trees
(Union-by-height and path-compression Find)

• Start with 4 items: 0, 1, 2, 3

• $\text{Union}(i,j)$ makes the root of the shorter tree a child of the root of the taller tree

• We perform path compression

• If an array element contains a negative $\text{int}$, then that element represents a tree root, and the value stored there is $-1$ times (the height of the tree plus 1)

Perform these operations:
  
  $\text{Union}(2,3)$
  $\text{Union}(0,1)$
  $\text{Find}(0) =$
  $\text{Find}(3) =$
  $\text{Find}(3) =$
  $\text{Union}(0,2)$
  $\text{Find}(1) =$
  $\text{Find}(3) =$
  $\text{Find}(3) =$
Self-adjusting data structures

• Path-compression Find for disjoint subset structures is an example of a *self-adjusting* structure

• Other examples of self-adjusting data structures are splay trees, self-adjusting lists, skew heaps, etc

• In a self-adjusting structure, a find operation occasionally incurs high cost because it does extra work to modify (adjust) the data structure, with the hope of making subsequent operations much more efficient

• Does this strategy pay off? *Amortized cost analysis* is the key to the answering that question...
Amortized cost analysis

• Amortization corresponds to spreading the cost of an infrequent expensive item (car, house) over the use period of the item.

• The amortized cost should be comparable to alternatives such as renting the item or taking a loan.

• Amortized analysis of a data structure considers the average cost over many actions.
Amortized cost analysis results for path compression Find

- It can be shown (Tarjan, 1984) that with Union-by-size or Union-by-height, using path-compression Find makes any combination of up to N-1 Union operations and M Find operations have a worst-case time cost of $O(N + M \log^* N)$

- This is very good: it is almost constant time per operation, when amortized over the N-1 + M operations!
Amortized cost analysis results for path compression Find

- $\log^* N = \text{“log star of } N\text{”} = \text{smallest } k \text{ such that } \log^{(k)} n \leq 1 \text{ or } # \text{ times you can take the log base-2 of } N, \text{ before we get a number } \leq 1$
- Also known as the “single variable inverse Ackerman function”

\[
\begin{align*}
\log^* 2 &= 1 \\
\log^* 4 &= 2 \\
\log^* 16 &= 3 \\
\log^* 65536 &= 4 \\
\log^* 2^{65536} &= 5
\end{align*}
\]

$4 = (2)^2$  $16 = (2)^4$  $65536 = (2)^{16}$  $2^{65536} = \text{a huge number of 20000 digits}$

- $\log^{(k)} n = \log (\log (\log \ldots (\log n)))$

- $k \text{ logs}$

- $\log^* N \text{ grows extremely slowly as a function of } N$
- It is not constant, but for all practical purposes, $\log^* N \text{ is never more than 5}$
Union/Find Code

• Using the array representation for disjoint subsets, the code for implementing the Disjoint Subset ADT’s methods is compact

```cpp
class DisjSets{
    int *array;

    /**
     * Construct the disjoint sets object
     * numElements is the initial number of disjoint sets
     */
    DisjSets( int numElements )
    {
        array = new int [ numElements ];
        for( int i = 0; i < numElements; i++ )
            array[ i ] = -1;
    }
}
```
/**
 * Union two disjoint sets using the height heuristic.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set labels.
 * root1 is the root of set 1
 * root2 is root of set 2
 * returns the root of the union
 */
int union ( int root1, int root2 ){
    if( array[ root2 ] < array[ root1 ] )
    {
        array[ root1 ] = root2; // root2 is higher
        return root2;            // Make root2 new root
    } else {
        if( array[ root1 ] == array[ root2 ] )
        {
            array[ root1 ]--;        // Update height if same
            array[ root2 ] = root1;   // Make root1 new root
            return root1;
        }
    }
}
Find with path compression

/**
 * Perform a find with path compression
 * Error checks omitted again for simplicity
 * @param x the label of the element being searched for
 * @return the label of the set containing x
 */
int find( int x ) {

    if( array[ x ] < 0 )
    [
        else return x;
        array[ x ] = find( array[ x ] );
    }

• Note that this path-compression find method does not update the disjoint subset tree heights; so the stored heights (called “ranks”) will overestimate of the true height

• A problem for the cost analysis of the union-by-height method (which now is properly called union-by-rank)
Building a random maze

• Suppose you want to construct a maze on a $N \times M$ grid
  Connected, random, only 1 path between any 2 rooms
• Start with an array of $N \times M$ rooms, each isolated from its neighbors by 4 ‘walls’
• Pick a wall at random. If knocking it down would join two unconnected compartments in the maze, then knock it down, else leave it alone
• Continue until all the rooms are connected
• There is a path between any two rooms in the maze

[Diagram of maze]

Henry Kautz
Building a random maze with union-find

- Approach: computing with equivalence classes
- Given: the equivalence relation \( E(i, j) \) is true iff you can get from room \( i \) to room \( j \)
- Initially each room is in its own singleton equivalence class
- Pick a wall at random. If knocking it down would join two distinct equivalence classes of room, do so; otherwise leave it standing
- Continue until all the rooms form one equivalence class, then stop
- Question: Given a \( N \times M \) maze constructed in this way, how many distinct simple paths are there from the “upper left corner” room to the “lower right corner” room?