Union-Find: A Data Structure for Disjoint Set Operations
The Union-Find Data Structure

- **Purpose:**
  - To manipulate disjoint sets (i.e., sets that don’t overlap)
  - Operations supported:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Union (x, y)</strong></td>
<td>Performs a union of the sets containing two elements x and y</td>
</tr>
<tr>
<td><strong>Find (x)</strong></td>
<td>Returns a pointer to the set containing element x</td>
</tr>
</tbody>
</table>

Q) Under what scenarios would one need these operations?
Some Motivating Applications for Union-Find Data Structures

Given a set $S$ of $n$ elements, $[a_1 \ldots a_n]$, compute all its equivalent classes

Example applications:
- Electrical cable/internet connectivity network
- Cities connected by roads
- Cities belonging to the same country
Equivalence Relations

- An *equivalence relation* $R$ is defined on a set $S$, if for every pair of elements $(a, b)$ in $S$,
  - $a R b$ is either false or true
- $a R b$ is true iff:
  - (**Reflexive**) $a R a$, for each element $a$ in $S$
  - (**Symmetric**) $a R b$ if and only if $b R a$
  - (**Transitive**) $a R b$ and $b R c$ implies $a R c$
- The *equivalence class of an element* $a$ (in $S$) is the subset of $S$ that contains all elements related to $a
Properties of Equivalence Classes

- **An observation:**
  - Each element must belong to exactly one equivalence class

- **Corollary:**
  - All equivalence classes are mutually disjoint

- What we are after is the set of all equivalence classes
Identifying equivalence classes

Legend:
- Equivalence class
- Pairwise relation

Cpt S 223. School of EECS, WSU
Disjoint Set Operations

To identify all equivalence classes

1. Initially, put each element in a set of its own

2. Permit only two types of operations:
   - \textbf{Find}(x): Returns the current equivalence class of \(x\)
   - \textbf{Union}(x, y): Merges the equivalence classes corresponding to elements \(x\) and \(y\) (assuming \(x\) and \(y\) are related by the eq.rel.)

This is same as:
\[
\text{unionSets( Find(x), Find(y) )}
\]
Steps in the Union \((x, y)\)

1. \(\text{EqClass}_x = \text{Find} \ (x)\)
2. \(\text{EqClass}_y = \text{Find} \ (y)\)
3. \(\text{EqClass}_{xy} = \text{EqClass}_x \cup \text{EqClass}_y\)
A Simple Algorithm to Compute Equivalence Classes

1. Initially, put each element in a set of its own i.e., \( \text{EqClass}_a = \{a\} \), for every \( a \in S \)

2. FOR EACH element pair \((a, b)\):
   1. Check \([a \mathbin{R} b == \text{true}]\)
   2. IF \( a \mathbin{R} b \) THEN
      1. \( \text{EqClass}_a = \text{Find}(a) \)
      2. \( \text{EqClass}_b = \text{Find}(b) \)
      3. \( \text{EqClass}_{ab} = \text{EqClass}_a \cup \text{EqClass}_b \)

\( \Theta(n^2) \) iterations
Specification for Union-Find

- **Find**(x)
  - Should return the id of the equivalence set that currently contains element x

- **Union**(a,b)
  - If a & b are in two different equivalence sets, then Union(a,b) should merge those two sets into one
    - Otherwise, no change
How to support Union() and Find() operations efficiently?

**Approach 1**
- *Keep the elements in the form of an array, where:* 
  
  \[ A[i] \text{ stores the current set ID for element } i \]

**Analysis:**
- Find() will take \( O(1) \) time
- Union() could take up to \( O(n) \) time
- Therefore a sequence of \( m \) (union and find) operations could take \( O(m \cdot n) \) in the worst case
  
  \[ \text{This is bad!} \]
How to support Union() and Find() operations efficiently?

- Approach 2
  - Keep all equivalence sets in separate linked lists:
    1 linked list for every set ID

- Analysis:
  - Union() now needs only $O(1)$ time
    (assume doubly linked list)
  - However, Find() could take up to $O(n)$ time
    - Slight improvements are possible (think of Balanced BSTs)
  - A sequence of $m$ operations takes $\Omega(m \log n)$
  - Still bad!
How to support Union() and Find() operations efficiently?

- Approach 3
  - *Keep all equivalence sets in separate trees: 1 tree for every set*
  - *Ensure (somehow) that Find() and Union() take very little time (<< O(log n))*

- That is the Union-Find Data Structure!

The Union-Find data structure for n elements is a forest of k trees, where 1 ≤ k ≤ n
Initialization

- Initially, each element is put in one set of its own
  - Start with $n$ sets $==$ $n$ trees

```
0
1
2
3
4
5
6
7
```
Union(4,5)

Union(6,7)
Union(5,6)

Link up the roots
The Union-Find Data Structure

- **Purpose:** To support two basic operations efficiently
  - Find \((x)\)
  - Union \((x, y)\)

- **Input:** An array of \(n\) elements

- Identify each element by its array index
  - Element label = array index
Union-Find Data Structure

```cpp
class DisjSets {
public:
explicit DisjSets( int numElements );

int find( int x ) const;
/ int find( int x );
void unionSets( int root1, int root2 );
void union(int x, int y);
private:
vector<int> s;
};
```

Note: This will always be a vector<int>, regardless of the data type of your elements. WHY?
Union-Find D/S: Implementation

- Entry $s[i]$ points to $i^{th}$ parent
- $-1$ means root

This is WHY

vector<int>

Cpt S 223. School of EECS, WSU
Union-Find: "Simple Version"

### Union-Find: "Simple Version"

**Simple Find" implementation**

```cpp
/**
 * Construct the disjoint sets object.
 * numElements is the initial number of disjoint sets.
 */
DisjSets::DisjSets( int numElements ) : s( numElements )
{
    for( int i = 0; i < s.size(); i++ )
        s[ i ] = -1;
}

// Union performed arbitrarily
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * root1 is the root of set 1.
 * root2 is the root of set 2.
 */
void DisjSets::unionSets( int root1, int root2 )
{
    s[ root2 ] = root1;
}

int DisjSets::find( int x ) const
{
    if( s[ x ] < 0 )
        return x;
    else
        return find( s[ x ] );
}
```

This could also be:

```
s[root1] = root2
```

void DisjSets::union(int a, int b)
{
    unionSets( find(a), find(b) );
}
```

a & b could be arbitrary elements (need not be roots)
Analysis of the simple version

- Each `unionSets()` takes only $O(1)$ in the worst case.
- Each `Find()` could take $O(n)$ time.
  - Therefore, each `Union()` could also take $O(n)$ time.
- Therefore, $m$ operations, where $m \gg n$, would take $O(mn)$ in the worst-case.

Pretty bad!
Smarter Union Algorithms

Problem with the arbitrary root attachment strategy in the simple approach is that:
- The tree, in the worst-case, could just grow along one long \((O(n))\) path

**Idea:** Prevent formation of such long chains
- \(\Rightarrow\) Enforce \(\text{Union}()\) to happen in a “balanced” way
Heuristic: Union-By-Size

- Attach the root of the "smaller" tree to the root of the "larger" tree

Find (3)
Size=4

Find (7)
Size=1

Union(3,7)

So, connect root 3 to root 4
Union-By-Size:

An arbitrary union could end up unbalanced like this:
Also known as “Union-By-Rank”

Another Heuristic: Union-By-Height

- Attach the root of the “shallower” tree to the root of the “deeper” tree

Find (3)

Find (7)

Height=2

Height=0

Union(3,7)

So, connect root 3 to root 4

Cpt S 223. School of EECS, WSU
Let us assume union-by-rank first

How to implement smart union?

Old method:

<table>
<thead>
<tr>
<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>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

But where will you keep track of the heights?

- \( s[i] \) = parent of \( i \)
- \( S[i] = -1 \), means root

New method:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-3</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

What is the problem if you store the height value directly?

- instead of roots storing -1, let them store a value that is equal to: -1-(tree height)
New code for union by rank?

```c
void DisjSets::unionSets(int root1, int root2) {
    // first compare heights

    // link up shorter tree as child of taller tree
    // if equal height, make arbitrary choice

    // then increment height of new merged tree if height has changed – will happen if merging two equal height trees
}
```
void DisjSets::unionSets(int root1, int root2) {
    assert(s[root1]<0);
    assert(s[root2]<0);
    if(s[root1]<s[root2]) s[root2]=root1;
    if(s[root2]<s[root1]) s[root1]=root2;
    if(s[root1]==s[root2]) {
        s[root1]=root2;
        s[root2]--;
        }
}
Note: All nodes, except root, store parent id. The root stores a value = negative(height) -1

Code for Union-By-Rank

```c++
    /*
    * Union two disjoint sets.
    * For simplicity, we assume root1 and root2 are distinct
    * and represent set names.
    * root1 is the root of set 1.
    * root2 is the root of set 2.
    */

    void DisjSets::unionSets( int root1, int root2 )
    {
        if( s[ root2 ] < s[ root1 ] ) // root2 is deeper
            s[ root1 ] = root2;       // Make root2 new root
        else
            { // root1 is deeper
                if( s[ root1 ] == s[ root2 ] )
                    s[ root1 ]--;      // Update height if same
                s[ root2 ] = root1;    // Make root1 new root
            }
    }
```

Similar code for union-by-size
How Good Are These Two Smart Union Heuristics?

- Worst-case tree

Maximum depth restricted to $O(\log n)$
Analysis: Smart Union Heuristics

- For smart union (by rank or by size):
  - Find() takes $O(\log n)$;
    - $\Rightarrow$ union() takes $O(\log n)$;
  - unionSets() takes $O(1)$ time
- For $m$ operations: $O(m \log n)$ run-time

- Can it be better?
  - What is still causing the $(\log n)$ factor is the distance of the root from the nodes
  - **Idea**: Get the nodes as close as possible to the root

Path Compression!
**Path Compression Heuristic**

- During find(x) operation:
  - Update all the nodes along the path from x to the root point directly to the root
  - A two-pass algorithm

How will this help?

Any future calls to find on x or its ancestors will return in constant time!
New code for find() using path compression?

```cpp
void DisjSets::find(int x) {

}
```
New code for find() using path compression?

```cpp
int DisjSets::find(int x) {
    // if x is root, then just return x
    if(s[x]<0) return x;

    // otherwise simply call find recursively, but.. 
    //  make sure you store the return value (root index) 
    //  to update s[x], for path compression 

    return s[x]=find(s[x]);
}
```
Path Compression: Code

```cpp
/**
 * Perform a find with path compression.
 * Error checks omitted again for simplicity.
 * Return the set containing x.
 */

int DisjSets::find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return s[ x ] = find( s[ x ] );
}
```

It can be proven that path compression alone ensures that find(x) can be achieved in $O(\log n)$.

Spot the difference from old find() code!
Union-by-Rank & Path-Compression: Code

Init()

```c++
/**
  * Construct the disjoint sets object.
  * numElements is the initial number of disjoint sets.
  */
void DisjSets::DisjSets( int numElements ) : s( numElements )
{
    for( int i = 0; i < s.size( ); i++ )
        s[ i ] = -1;
}
```

Union()

```c++
void DisjSets::union(int a, int b)
{
    unionSets( find(a), find(b) );
}
```

unionSets()

```c++
/**
  * Union two disjoint sets.
  * For simplicity, we assume root1 and root2 are distinct
  * and represent set names.
  * root1 is the root of set 1.
  * root2 is the root of set 2.
  */
void DisjSets::unionSets( int root1, int root2 )
{
    if( s[ root2 ] < s[ root1 ] ) // root2 is deeper
        s[ root1 ] = root2;    // Make root2 new root
    else
    {
        if( s[ root1 ] == s[ root2 ] )
            s[ root1 ]--;        // Update height if same
        s[ root2 ] = root1;     // Make root1 new root
    }
}
```

Find()

```c++
/**
  * Perform a find with path compression.
  * Error checks omitted again for simplicity.
  */
int DisjSets::find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return s[ x ] = find( s[ x ] );
}
```

Amortized complexity for m operations:
$$O(m \text{ Inv. Ackerman } (m,n)) = O(m \log^*n)$$
# Heuristics & their Gains

<table>
<thead>
<tr>
<th>Heuristic Description</th>
<th>Worst-case run-time for m operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary Union, Simple Find</td>
<td>$O(m \cdot n)$</td>
</tr>
<tr>
<td>Union-by-size, Simple Find</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td>Union-by-rank, Simple Find</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td>Arbitrary Union, Path compression Find</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td><strong>Union-by-rank, Path compression Find</strong></td>
<td>$O(m \text{Inv.Ackermann}(m,n)) = O(m \log^*n)$</td>
</tr>
</tbody>
</table>

*Extremely slow*  
*Growing function*
What is Inverse Ackermann Function?

- $A(1,j) = 2^j$ for $j \geq 1$
- $A(i,1) = A(i-1,2)$ for $i \geq 2$
- $A(i,j) = A(i-1,A(i,j-1))$ for $i,j \geq 2$

- $\text{InvAck}(m,n) = \min\{i \mid A(i,\text{floor}(m/n)) > \log N\}$

- $\text{InvAck}(m,n) = O(\log^* n)$ (pronounced “log star n”)
  
Even Slower! A very slow function
How Slow is Inverse Ackermann Function?

- What is log*n?
  - A very slow function

- log*n = log log log log log ........ n
  - How many times we have to repeatedly take log on n to make the value to 1?
  - log*65536=4, but log*2^{65536}=5
Some Applications
A Naïve Algorithm for Equivalence Class Computation

1. Initially, put each element in a set of its own
   i.e., EqClass\(_a\) = \{a\}, for every \(a \in S\)

2. FOR EACH element pair \((a,b)\):
   1. Check \([a \sim b == true]\)
   2. IF \(a \sim b\) THEN
      1. \(\text{EqClass}_a = \text{Find}(a)\)
      2. \(\text{EqClass}_b = \text{Find}(b)\)
      3. \(\text{EqClass}_{ab} = \text{EqClass}_a \cup \text{EqClass}_b\)

\(\Theta(n^2)\) iterations

Run-time using union-find: \(O(n^2 \log^* n)\)

Better solutions using other data structures/techniques could exist depending on the application
An Application: Maze
Strategy:

- As you find cells that are connected, collapse them into equivalent set.
- If no more collapses are possible, examine if the Entrance cell and the Exit cell are in the same set.
  - If so => we have a solution.
  - O/w => no solutions exists.
Strategy:

- As you find cells that are connected, collapse them into equivalent set.
- If no more collapses are possible, examine if the Entrance cell and the Exit cell are in the same set.
  - If so => we have a solution
  - O/w => no solutions exists

\{0, 1\} \{2\} \{3\} \{4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22\} \{5\} \{10, 11, 15\} \{12\} \{19\} \{20\} \{21\} \{23\} \{24\}
Another Application: Assembling Multiple Jigsaw Puzzles at once

Merging Criterion: Visual & Geometric Alignment

Picture Source: http://ssed.gsfc.nasa.gov/lepedu/jigsaw.html
Summary

- Union Find data structure
  - Simple & elegant
  - Complicated analysis
- Great for disjoint set operations
  - Union & Find
  - In general, great for applications with a need for “clustering”