Union/Find
Aka: Disjoint-set forest

Alon Efrat

Problem definition

Given: A set of atoms \( S = \{1, 2, \ldots, n\} \)
E.g. each represents a commercial name of a drugs.
This set consists of different disjoint subsets.

Problem: suggest a data structures that efficiently supports two operations

- **Find** \((i, j)\) – reports if the atom \(i\) and atom \(j\) belong to the same set.
- **Union** \((i, j)\) – unify (merged) all elements of the set containing \(i\) with the set containing \(j\).

*Example – on the board.

Naïve attempts

Idea: Each element “knows” to which set it belongs
(recall – each atom belongs to exactly one set)

Bad idea: once two sets are merged, we need to scan all elements of one set and “tell” them that they belong to a different set – requires lots of work if the set is large.

A Promising Attempt

- Store a forest of trees.
- Each set is stored as a tree (each node is an atom)
- Every node points to the parent
(different than standard trees)

Only the root “knows” the name of the set.

So the ‘name’ of the set \(\{2, 3, 4, 1\}\) is 2.
The name of the set \(\{5, 6, 7, 8\}\) is 8.
The name of the set \(\{9\}\) is 9.
The name of the set \(\{11, 12\}\) is 12.

To find if two atoms belong to the same set, just check if they belong to same tree: Follow the parents pointers from each of them up all the way to the roots. Check if reached the same root.
A forest of disjoint sets – merging trees

Find_root(j){
    If (p[j] ≠ j) return Find_root(p[j]);
    // p[] - points to the parent
    Else return j;
}

Find(i,j){
    return (Find_root(i) == Find_root(j));
}

Union(i,j){
    Let r = Find_root(j);
    p[r] = i;
}

Example – Union(5,11)

It this efficient?

Time per ans operation depends on the height of the tree. Might be \( \Theta(n) \) in the worst case.

So \( n \) operations takes \( \Theta(n^2) \)

Could we do better?

First improvement

Improved union operation – version 1

Example – Union(5,11)

Note that we can also do

Improved union operation – version 1
**Keeping tracks of # nodes**

Every root (only roots) stores the number of nodes in its tree. Let \( n_r \) denote this field in the root \( r \).

```plaintext
Union(i,j) {
    Let \( r_1 = \text{Find}_\text{root}(i) \);  Let \( r_2 = \text{Find}_\text{root}(j) \);
    /* connect the root of the small tree as a child of the root of
       the larger tree */
    if (\( r_1.n < r_2.n \)) {    p[\( r_1 \)] = \( r_2 \);
        \( r_2.n += r_1.n \);  }
    else {   p[\( r_2 \)] = \( r_1 \);
        \( r_1.n += r_2.n \)  }
}
```

Example: \( \text{Union}(9,3) \)

**Proving bounds on the height**

Assume we start from a forest where each node is a singleton (a set of one element), and we perform a sequence of union operations.

**Lemma:** The height of every tree is \( \leq \log_2 n \). (\( n \) – number of atoms)

**Proof:** Show by induction that every tree of height \( h \) has \( \geq 2^h \) nodes.

Assume true for every tree of height \( h' < h \), and assume that after merging trees \( T_1, T_2 \), we created a tree of height exactly \( h \).

\( T_2 \) has height exactly \( h-1 \), so it has \( \geq 2^{h-1} \) nodes.

\( T_1 \) also has \( 2^{h-1} \) nodes, (why?)

Together they have \( 2^{h-1} + 2^{h-1} = 2^h \) nodes.

**Further improvement: path compression**

So far we know that every tree has height \( O(\log n) \), so this bounds the time for each operation.

**Path compression:** during either union or find operation, we scan a sequence of nodes on our way from a node \( j \) to the root.

Idea: set the parent pointer of all these node to points to the root.
(Slightly more work to perform it, but pays off in next operations)

```plaintext
\text{Find}_\text{root}(j ) {  
    \text{If } p[j] \neq j \text{ then } p[j]=\text{Find}_\text{root}(p[j]);  
    \text{return } p[j]  
}
```

**Make sense – but how fast is it?**

Consider a set of \( n \) atoms.

**Thm:** Any sequence of \( m \) U/F operations takes \( O(m a(n)) \).

Here \( a(n) \) is the inverse function of Ackerman function, and is approaching infinity as \( n \) approaching infinity.

However, it does it very slowly.

\( a(n) <4 \) when \( n < 10^{100} \).
Connected Components in Undirected graphs

Let $G(V, E)$ be a graph.

**Definition:**
We say that a subset $C$ of $V$ is a connected component (CC) iff for every pair $u, v \in C$, there is a path connecting $u$ to $v$, and no path connects $u$ to a node $\notin C$.

**Examples:**
1. If $G(V, E)$ is connected then $V$ is a (single) CC.
2. If $G(V, E)$ contains no edges, then every node is CC, which contains only itself.
3. If $G(V, E)$ is a tree, and we deleted an edge from $E$, then the resulting graph has 2 CCs.

Minimum Spanning Trees

**Given:** a graph $G(V, E)$ with positive weights on its edges.

A Minimum spanning tree (MST) is any graph $T$ such that

1. Every vertex of $V$ appears in $T$, and
2. $T$ is connected (has a path between every two vertices)
3. $T$ is a subset of $E$
4. Sum of weights of its edges are as small as possible

Application: Kruskal algorithm

**Kruskal algorithm for finding a MST.**

**Input:** Graph $G(V, E)$. **Output:** Minimal Spanning Tree for $G$.

1. Assume $E = \{e_1, ..., e_m\}$ is sorted from cheapest edge to most expensive edge.
2. Set $S = \text{EmptySet}$. $S$ determines or more trees, each is a subtree of $\text{MST}(G)$.
3. For $i = 1..m$
   - if $e_i \cup S$ does not contain a cycle, add $e_i$ to $S$
     // We use U/F structure to answer last test

We need a mechanism to determine if a new edge $e=(u,v)$ connects two vertices from different trees, or both $u$ and $v$ belong to the same tree.

Idea: Use U/F data structure.

**If** $E$ is sorted, then the time is $O(|E| \alpha(|E|))$. 

Application: Kruskal algorithm

**Input:** Graph $G(V, E)$. **Output:** Minimal Spanning Tree for $G$.

Assume $E = \{e_1, ..., e_m\}$ is sorted from cheapest edge to most expensive edge.

The algorithm maintains a set $S$ of disjoint trees, each is a subtree of $\text{MST}(G)$

1. Set $S = \text{EmptySet}$. $S$ determines or more trees, each is a subtree of $\text{MST}(G)$
2. Assume $E = \{e_1, ..., e_m\}$ is sorted from cheapest edge to most expensive edge.
3. For $i = 1..m$
   - if $e_i \cup S$ does not contain a cycle, add $e_i$ to $S$
     // We use U/F structure to answer last test

We need a mechanism to determine if a new edge $e=(u,v)$ connects two vertices from different trees, or both $u$ and $v$ belong to the same tree.

Idea: Use U/F data structure.

**If** $E$ is sorted, then the time is $O(|E| \alpha(|E|))$. 