Union Find Data Structure

- maintain a collection of disjoint sets subject to
  - Union (A, B) - unite two sets A and B (destroy A, B)
  - Find (e) - which set contains element e

A motivating problem: find all connected components in a graph

\[ \text{Figure: Connected Components in a Graph} \]

\[ n = \# \text{vertices} \]
\[ m = \# \text{edges} \]

Depth-first search in \( O(n+m) \) time

Dynamic Graph Connectivity - maintain connected components as the graph changes

answer queries: given vertices \( u, v \), are they connected?

Examples

- social networks, as relationships added/deleted

Problem is relevant but a bit hard.

Incremental Dynamic Graph Connectivity

- add edges to the graph, but don't delete any

This can be solved by Union-Find.
An application:
Kruskal's min-spanning tree algorithm
- Sort edges \( e_1, e_2 \ldots e_{|E|} \) by weight,
  \[ w(e_i) \leq w(e_{i+1}) \]
- Initial tree \( T \leftarrow \emptyset \) — each vertex is a component by itself
- For \( i = 1 \ldots |E| \)
  - If endpoints of \( e_i \) are in different components of \( T \) \( \leftarrow \) use two Find operations on endpoints of \( e_i \)
  - Then
    - \( T \leftarrow T \cup \{ e_i \} \)
    - Combine the 2 components \( \leftarrow \) Union

Note: in this application we care about a sequence of unions and finds
so an amortized analysis is appropriate
Today - implement Union Find, good amortized bound.

\[ n = \#\text{elements} \]
\[ m = \#\text{operations} = \#\text{Finds} + \#\text{Unions} \]
Note \( \#\text{Unions} \leq n-1 \)

First approach

Keep a list of elements in each set and
Use array \( S[i..n] \) \( S[i] = \text{name of set containing i} \)
Find \( O(1) \)
Union \( O(n) \) worst case

Tiny improvement:

\( \text{Union}(A,B) - \text{update} \ S[i] \text{ for i in smaller of A,B} \)

eg.
\[
\begin{align*}
A : & 1, 3 \\
B : & 2, 7, 6, 5 \\
C : & 4 \\
\end{align*}
\]

\[
\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
A & B & A & C & B & B & B & B \\
B & B & B & B & B & B & B & B \\
\end{array}
\]

For \( \text{Union}(A,B) \), update for \( A \)

Cost of all unions \( \leq O(n \log n) \) because each element changes its set \( \leq \log n \) times

Cost of \( m \) operations \( O(m + n \log n) \)

If \( m \geq n \log n \quad (m \in \Omega(n \log n)) \)
then this is best possible
A better method in case #Finds is small.

Represent each set as a tree of parent pointers.

Find(e) — walk up the tree from e & get set name at root.

Union — link the 2 trees — make "smaller" tree's root point to root of other tree.

Path compression: After Find(e), for every node x on path v → root, set parent(x) ← root.

This doubles work of Find, but same O(α(n)).

Last detail: how to define "smaller".

Want a tree with smaller height:

But we cannot easily keep track of height (path compression changes it).
Idea: will use "rank" \( r \), defined as follows:

- \( r \) (single node) = 0
- new rank of root
  \[
  r = \max \{ r_1, r_2 + 1 \}
  \]
  \[
  \begin{cases} 
  r_1 & \text{if } r_1 > r_2 \\
  r_1 + 1 & \text{if } r_1 = r_2 
  \end{cases}
  \]

- rank = height if no-path compressions are done.
- Note: every node will have a rank. Only rank (root) changes.

Algorithm (above) is simple. But analysis is hard.

**Theorem** [Tarjan 1975]
The cost of \( m \) operations is \( \Theta(m \cdot \alpha(m,n)) \)
i.e. amortized cost per operation is \( \Theta(\alpha(m,n)) \)

\( \alpha(m,n) \) — inverse Ackerman function
- very slow growing. \( \leq 5 \) for all practical purposes.

This bound is tight — there is an infinite class of examples where the alg. takes this time.

We will prove a slightly weaker bound of \( O(m \log^* n) \)
using a charging scheme.

**Notes:**
- for \( \Theta(m \cdot \alpha(m,n)) \) bound see CLRS
- for alternate pf. of \( O(m \log^* n) \) see Jeff Erickson's notes.
\[
\log^* n = \min_i \left( \log \underbrace{\log \cdots (\log n)}_{i \text{-times}} \right) \leq 1 \frac{2}{3}
\]

i.e. define \( 2^\uparrow n = \frac{2^{2^{n}}}{n!} \)

Then \( \log^* (2^\uparrow n) = n \)

\[
\begin{array}{c|ccccccc}
 n & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 \log^* n & 0 & 1 & 2 & 3 & 4 & 5 & \infty \\
\end{array}
\]

cost of Find \( (u) \) = distance from \( u \) to root

Idea: change some of this cost to the Find and some to the nodes along the path. Then sum up.

\textbf{Claim 1} when a vertex is assigned rank \( r \) it has \( \geq 2^r \) descendants.

\textbf{pf.} by induction.

\textbf{Claim 2} \( \text{rank}(u) < \text{rank}(\text{parent}(u)) \)

\textbf{Claim 3} \# vertices of rank \( r \) \( \leq \frac{n}{2^r} \)

If when vertex is assigned rank \( r \) it has \( \geq 2^r \) descendants.

Vertices of rank \( r \) have disjoint descendants because when a vertex changes its parent (by path compression) it gets a parent of higher rank.

Divide vertices into groups based on rank

vertex of rank \( r \) goes in group \( \log^* r \)

\# groups = \( \log^* n \)

group \( g \) contains ranks \( 2^\uparrow (g-1) + 1 \cdots 2^\uparrow g \)

as in the table above
Charge for Find (v)
- for each vertex u on path v → root
  - if u has parent x and grandparent y and group(u)=group(p(u))
    then charge 1 to u
  - else charge 1 to Find (v)
Note that this covers cost.

Total charges to Find (v) ≤ log⁺ n + 1
since group changes ≤ log⁺ n - 1 times
+ 2 for root and child of root.

Now add up the charges to vertices

Total charge to vertex u in group g:
- if u is charged then path compression will give it a new parent of higher rank than the old parent by claim 2
- so u in group g is charged
  \[ c(g) = \left( \# \text{ ranks in group } g \right) - 1 \]
  times before it acquires a parent in a higher group — and after that it is not charged.
  \[ c(g) ≤ 2 \uparrow g \] from above

Total charge to all vertices in group g
\[ c(g) \cdot N(g) \] where \[ N(g) = \# \text{ vertices in group } g \]
\[ N(g) ≤ \sum_{r=2 \uparrow (g-1)+1}^{2 \uparrow g} \frac{n}{2^r} ≤ \frac{n}{2^{2 \uparrow (g-1)+1}} \sum_0^{2 \uparrow g} \frac{1}{2^i} = \frac{n}{2 \uparrow g} \]

Thus \[ c(g) \cdot N(g) ≤ n \]
Total charge to all vertices

\[ \text{charge to 1 group} \cdot \frac{n}{\log^* n} \]

Total charge to `Finds` and vertices

\[ O\left(\frac{m}{\log^* n + 1} \cdot \log^* n + n \cdot \log^* n\right) = O\left(m \cdot \log^* n\right) \]

Note standard assumption that \( m \geq n \)