Union Find
For a given graph $G$, can we determine whether or not two vertices are connected in $G$?

Can also be viewed as checking subset membership

Important for many practical applications

We will solve this problem using a union/find data structure
A simple approach

- Have an id array simply store the component id for each item in the union/find structure
  - Find simply returns its id
  - What about union?
Example

U(2, 0)
U(4, 7)
U(1, 2)
U(3, 2)
U(4, 5)
U(5, 7)
U(6, 3)

ID:

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
6 & 6 & 6 & 6 & 4 & 4 & 6 & 4 \\
\end{array}
\]
Analysis of our simple approach

● Runtime?
  ○ For find():
    ■ $\Theta(1)$
  ○ For union():
    ■ $\Theta(n)$
Union Find API

- `UF (int n)`
  - Initialize with n items numbered 0 to n-1

- `void union(int p, int q)`
  - Connect p with q

- `int find (int p)`
  - Return id of the connected component that p is in

- `boolean connected (int p, int q)`
  - True if p and q are connected

- `int count()`
  - Number of connected components
public int count() {
    return count;
}

public boolean connected(int p, int q) {
    return find(p) == find(q);
}
public UF(int n) {
    count = n;
    id = new int[n];
    for (int i = 0; i < n; i++) { id[i] = i; }
}

public int find(int p) { return id[p]; }

public void union(int p, int q) {
    int pID = find(p), qID = find(q);
    if (pID == qID) return;
    for(int i = 0; i < id.length; i++)
        if (id[i] == pID) id[i] = qID;
    count--;
}
Can we improve on union()’s runtime?

- What if we store our connected components as a forest of trees?
  - Each tree representing a different connected component
  - Every time a new connection is made, we simply make one tree the child of another
Tree example
public int find(int p) {
    while (p != id[p]) p = id[p];
    return p;
}

public void union(int p, int q) {
    int i = find(p);
    int j = find(q);
    if (i == j) return;
    id[i] = j;
    count--;
}

Implementation using the same id array
Forest of trees implementation analysis

- Runtime?
  - find():
    - Bound by the height of the tree
  - union():
    - Bound by the height of the tree
- What is the max height of the tree?
  - Can we modify our approach to cap its max height?
Weighted tree example
public UF(int n) {
    count = n;
    id = new int[n];
    sz = new int[n];
    for (int i = 0; i < n; i++) { id[i] = i; sz[i] = 1; }
}

public void union(int p, int q) {
    int i = find(p), j = find(q);
    if (i == j) return;
    if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
    else               { id[j] = i; sz[i] += sz[j]; }
    count--;
}
Weighted tree approach analysis

- Runtime?
  - find():
    - $\Theta(\log n)$
  - union():
    - $\Theta(\log n)$

- Can we do any better?
With this knowledge of union/find, how, exactly can it be used as a part of Kruskal’s algorithm?

- What is the runtime of Kruskal’s algorithm?