CMSC 341

Disjoint Sets

Disjoint Set Definition

- Suppose we have an application involving N distinct items. We will not be adding new items, nor deleting any items. Our application requires us to partition the items into a collection of sets such that:
  - each item is in a set
  - no item is in more than one set
- Examples
  - UMBC students according to class rank
  - CMSC 341 students according to GPA
- The resulting sets are said to be ***disjoint sets*.**

Disjoint Set Terminology

- We identify a set by choosing a ***representative element*** of the set. It doesn't matter which element we choose, but once chosen, it can't change.
- There are two operations of interest:
  - find (x) -- determine which set x is in. The return value is the representative element of that set
  - union (x, y) -- make one set out of the sets containing x and y.
- Disjoint set algorithms are sometimes called ***union-find*** algorithms.

Disjoint Set Example

Given a set of cities, C, and a set of roads, R, that connect two cities (x, y) determine if it's possible to travel from any given city to another given city

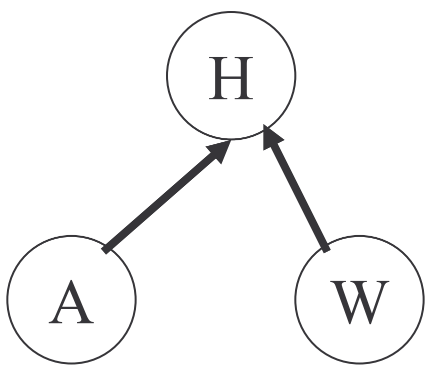
```
for (each city in C)
    put each city in its own set
for (each road (x,y) in R)
    if (find( x ) != find( y ))
        union(x, y)
```

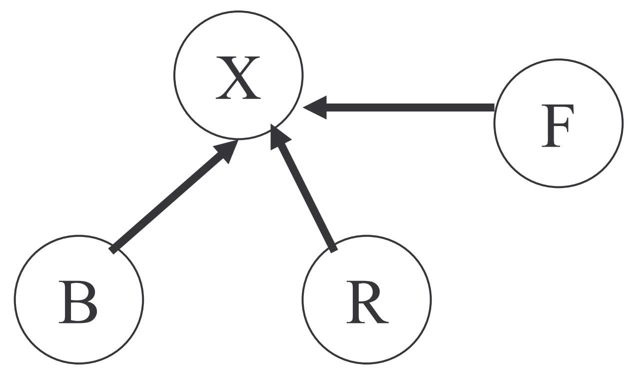
Now we can determine if it's possible to travel by road between two cities c_1 and c_2 by testing

```
find(c1) == find(c2)
```

Up-Trees

- A simple data structure for implementing disjoint sets is the *up-tree*.





H, A and W belong to the same set. H is the representative

X, B, R and F are in the same set. X is the representative

Operations in Up-Trees

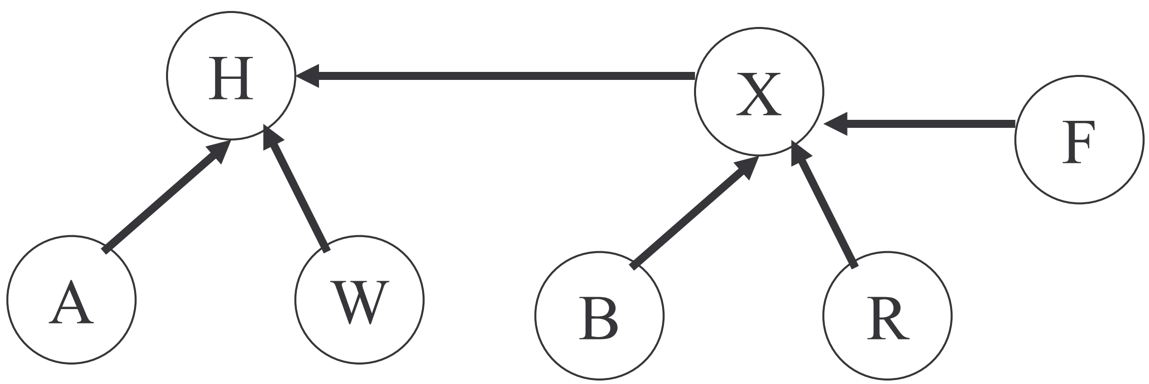
find( ) is easy. Just follow pointer to representative element. The representative has no parent.

```
find(x)
{
  if (parent(x))     // not the representative
     return(find(parent(x));
  else
     return (x);     // representative
}
```

Union

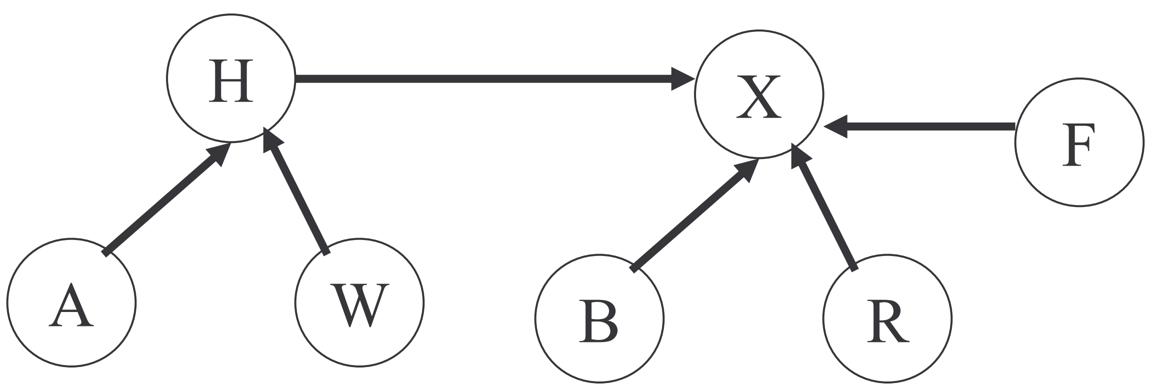
- Union is more complicated.
- Make one representative element point to the other, but which way? Does it matter?
- In the example, some elements are now twice as deep as they were before

Union(H, X)



X points to H

B, R and F are now deeper

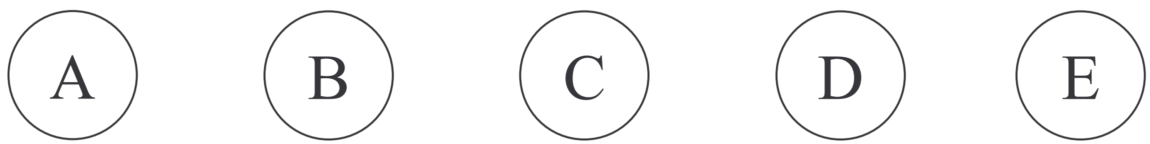


H points to X

A and W are now deeper

A worse case for Union

Union can be done in O(1), but may cause find to become O(n)



Consider the result of the following sequence of operations:

Union (A, B)
Union (C, A)
Union (D, C)
Union (E, D)

Array Representation of Up-tree

- Assume each element is associated with an integer i = 0…n-1. From now on, we deal only with i.
- Create an integer array, A[n]
- An array entry is the element's parent
- A[i] = -1 signifies that element i is the representative element.

Union/Find with an Array

Now the union algorithm might be:

```
Union(x,y) {
    A[y] = x;   // attaches y to x
}
```

The find algorithm would be

```
find(x) {
    if (A[x] < 0)
          return(x);
    else
          return(find(A[x]));
}
```

Improving Performance

- There are two heuristics that improve the performance of union-find.
  - Path compression on find
  - Union by weight

Path Compression

Each time we find( ) an element E, we make all elements on the path from E to the root be immediate children of root by making each element's parent be the representative.

```
find(x) {
    if (A[x]<0)
          return(x);
    A[x] = find(A[x]);  // one new line of code
    return (A[x]);
}
```

When path compression is used, a sequence of m operations takes O(m lg n) time. Amortized time is O(lg n) per operation.

“Union by Weight” Heuristic

Always attach the smaller tree to larger tree.

```
union(x,y) {
    rep_x = find(x);
    rep_y = find(y);
    if (weight[rep_x] < weight[rep_y]) {
         A[rep_x] = rep_y;
         weight[rep_y] += weight[rep_x];
    }
    else {
         A[rep_y] = rep_x;
         weight[rep_x] += weight[rep_y];
    }
}
```

Performance w/ Union by Weight

- If unions are performed by weight, the depth of any element is never greater than lg N.
- Intuitive Proof:
  - Initially, ever element is at depth zero.
  - An element's depth only increases as a result of a union operation if it's in the smaller tree in which case it is placed in a tree that becomes at least twice as large as before (union of two equal size trees).
  - Only lg N such unions can be performed until all elements are in the same tree
- Therefore, find( ) becomes O(lg n) when union by weight is used -- even without path compression.

Performance with Both Optimizations

- When both optimizations are performed a sequence of m ($m \geq n$) operations (unions and finds), takes no more than $O(m \lg^* n)$ time.
  - $\lg^* n$ is the iterated (base 2) logarithm of n -- the number of times you take lg n before n becomes ≤ 1.

- Union-find is essentially O(m) for a

A Union-Find Application

- A random maze generator can use union-find. Consider a 5x5 maze:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Maze Generator

- Initially, 25 cells, each isolated by walls from the others.
- This corresponds to an equivalence relation -- two cells are equivalent if they can be reached from each other (walls been removed so there is a path from one to the other).

Maze Generator (cont'd)

- To start, choose an entrance and an exit.

IN →

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

→ OUT

Maze Generator (cont’d)

- Randomly remove walls until the entrance and exit cells are in the same set.
- Removing a wall is the same as doing a union operation.
- Do not remove a randomly chosen wall if the cells it separates are already in the same set.

MakeMaze

```
MakeMaze(int size) {
  entrance = 0; exit = size-1;
  while (find(entrance) != find(exit)) {
     cell1 = a randomly chosen cell
     cell2 = a randomly chosen adjacent cell
     if (find(cell1) != find(cell2)
          union(cell1, cell2)
     }
  }
```