### Lecture 5

**Disjoint Sets**
- Equivalence Relations
- Union - Find

---

**Resources for this lecture**

- Roughgarden Algorithms 2 - weeks 1 & 2
  
  https://class.coursera.org/algo2-2012-001/lecture/63

  However, this requires you have seen
  Kruskal's algorithm for minimal spanning
  trees (coming soon!)

- Cormen, Leiserson, Rivest textbook (CLR)
  Chapter 22

---

The next part of the course, starting next lecture, will be about graphs.

\[ G = (V, E) \]

**Review Comp 250**
- graph traversal
  - breadth first search

---

The set of vertices \( V \) in a graph \( G \) is naturally partitioned into **connected components**, that is, sets of vertices that are "path connected".

Given two vertices, do they belong to the same connected component, that is, is there a path between the two vertices?

---

More generally, suppose we have a set of objects that is partitioned into disjoint subsets.

\[ S = S_1 \cup S_2 \cup \ldots \cup S_k \]

where \( S_i \neq \emptyset \) for all \( i \)

\[ S_i \cap S_j = \emptyset \text{ iff } i \neq j \]

**Notation:** set union \( \cup \)
set intersection \( \cap \)
empty set \( \emptyset \)
Do two objects belong to the same set in the partition?

What are good data structures and algorithms for solving this problem?

To keep the discussion simple, assume the set is:

\[ S = \{ 0, 1, 2, 3, \ldots, n-1 \} \]

In the graph application, these would be vertices \( V_0, V_1, \ldots, V_{n-1} \).

---

**Disjoint Sets**

- Equivalence Relations
- Union-Find

---

**Map vs. Relation**

Map or function

\[ \text{map } f : S \rightarrow S \]

Relation

\[ R \subseteq \{ (a, b) : a, b \in S \} \]

Any boolean matrix defines a relation

---

**Equivalence Relation (Much more constrained)**

Example: partition of a set

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
5 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\( i \) is equivalent to \( j \) if they belong in the same set

---

**Equivalence Relation (MATF 240)**


- Reflexivity
  
  for all \( a \in S \), \( (a, a) \in R \)

- Symmetry
  
  for all \( a, b \in S \), \( (a, b) \in R \implies (b, a) \in R \)

- Transitivity
  
  for all \( a, b, c \in S \), \( (a, b) \in R \) and \( (b, c) \in R \implies (a, c) \in R \)
Java

equals() defines an equivalence relation on objects

- reflexive: a.equals(a) returns true
- symmetric: a.equals(b) == b.equals(a)
- transitive: a.equals(b) and b.equals(c) => a.equals(c)

Example

For any undirected graph, path-connected(u,v) defines an equivalence relation on vertices.

- there is a path of length 0 from u to v, for all vertices v
- there is a path from u to v iff there is a path from v to u.
- if there is a path from vi to vj and there is a path from vj to vk then there is a path from vi to vk

Disjoint Sets ADT

Assume each set in the partition has a unique representative member.

- find(i) returns the representative of the set containing i (i.e. "findrep")
- sameSet(i,j) returns boolean value: find(i) == find(j)
- union(i,j)

Disjoint Sets ADT

- union(i,j) merges the sets containing i and j.
- we use it to build up the partition:
  - does nothing if i is already in the same set
  - otherwise, we need a policy for deciding who is the representative of the new (merged) set

Disjoint Sets data structures:

- quick find

Let rep[i] be the representative of the set containing i.

rep[1] = 1
rep[2] = 2
rep[3] = 1
rep[4] = 4
rep[5] = 5
rep[6] = 6
rep[7] = 1

N = 8
"quick find" (but slow union)

- **find(i)** \{ return\ rep[i] \}
- **union(i, j)** \{ merge i's set into j's \}

\[ O(n) \]

This seems to work. But there's an error!

union(i, j) \{
  if \ rep[i] \neq \ rep[j] 
    for each \ k \in 0,\ldots,n-1 
      if \ rep[k] = \ rep[i] 
        rep[k] = \ rep[j]
  \}

This is correct, but it's too slow.
\[ O(n) \] per union.

**Disjoint Sets data structures:**

1. "quick union"

Represent the disjoint sets by a "forest" of rooted trees.

The roots are the representatives, i.e., \( \text{find}(i) = \text{find} \text{rep}(i) = \text{find root}(i) \)

Each node points to its parent.

Use an array \( p[\ ] \) = parent[ ]

- Non-root nodes hold index of parent.
- Root nodes have value \(-1\)

\[ p[\ ] \]

```
  0 1 2 3 4 5 6 7
  |   |   |   |   |
  -1 1 7 0 2 -1 2 7
```

find(i) \{
  if \ p[i] = -1 
    return i 
  else 
    return find( p[i] )
\}

union(i, j) \{
  if \ find(i) \neq \ find(j) 
    \[ p[\ find(i)] = \ find(j) \]
  \}

// arbitrarily makes i's set merge into j's set
(j's rep is the rep of the merged set)
**Example**

union(i, j)

**Worst Case**

union(0, 1)
union(1, 2)
union(2, 3)
union(3, 4)
...
union(m-2, m-1)

\[\text{find}(0) \text{ is } O(m)\]

**Union by Size**

Merge tree with fewer nodes into tree with more nodes. (Break ties arbitrarily e.g. using previous union.)

union(i, j)

Claim: The depth of any node is at most \(\log n\) (no matter how many unions were performed).

**Proof:**

If union causes the depth of a node to increase, then this node must belong to the smaller tree. Thus, the size of tree containing this node will at least double. But you can double the size at most \(\log n\) times.

**Union by Height**

Merge tree with smaller height into tree with larger height.

Claim: the height of a union-by-height tree is at most \(\log n\).

**Proof (by induction)**

Base case: \(h = 0 \Rightarrow \text{tree has 1 node}\) \(\checkmark\)

Induction hypothesis: state claim for \(h = k\).

Induction step: show claim for \(h = k+1\)

\[\text{[See Exercises]}\]
Path Compression (Quicker union)

```
find(i) {
    if p[i] == -1
        return i
    else
        return find(p[i])
}
```

\[ p[i] = \text{find}(p[i]) \]
\[ \text{return } p[i] \]

[Advanced Topic: not on exams]

The worst case of find is \( O(\log n) \). However, it can be shown that, with path compression, in unions or finds, it takes \( O(m \log^* n) \) rather than \( O(m \log n) \).

What is that?

see Roughgarden Algorithms 2: Advanced Union-Find ("union by rank")

Announcements

- T.A. office hours for Al
- T.A. office hours in general (optional)
- Al should be relatively easy. A better reflection of how you are doing is: how easy are the exercises?
- Coming up... graph algorithms

(The algorithms are easy. The proofs are not.)

Define the "iterated logarithm" \( \log^* n \) to be the number of times you apply \( \log(\ ) \) until you get a value less than or equal to 1.

\[
\begin{align*}
\begin{array}{|c|c|}
\hline
n & \log^* n \\
\hline
0, 1 & 0 \quad \text{since it grows so slowly} \\
1, 2 & 1 \\
2, 2 & 2 \\
(4, 16) & 3 \\
(16, 256) & 4 \\
(2^4, 2^8) & 5 \\
\end{array}
\end{align*}
\]