lecture 5

Disjoint Sets

- Equivalence Relations
- Union - Find

Resources for this lecture

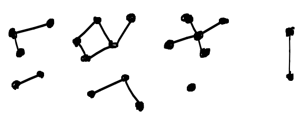
- Roughgarden Algorithms 2 - weeks 1 & 2

  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, Rivet 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 / depth first search)

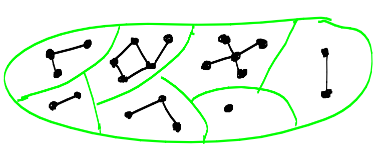
Suppose we have an undirected graph and we would like to know:

given two vertices, is there a path between them?

But, and here's the interesting part, we might not care what the path is.

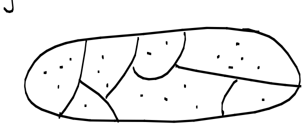
Is there a faster way to solve the above problem than graph traversal (BFS, DFS)?

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.

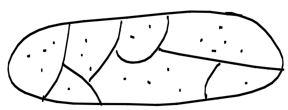


$S = S_1 \cup S_2 \cup \dots \cup S_k$

where
- $S_i \neq \emptyset$ for all i
- $S_i \cap S_j = \emptyset$ iff $i \neq j$

"partition" (math formalism)

Notation: set union $\cup$, set intersection $\cap$, empty set $\emptyset$

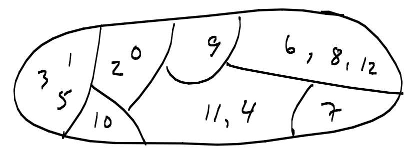


Do two objects belong to the same set in the partition?

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 $v_0, v_1 \ldots v_{n-1}$.

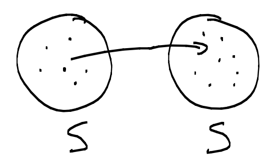
lecture 5

Disjoint Sets

- Equivalence Relations
- Union - Find

Map vs. Relation

map or function $f:$



$\{(a, f(a))\}$

relation $R \subseteq \{(a,b) : a, b \in S\}$

b

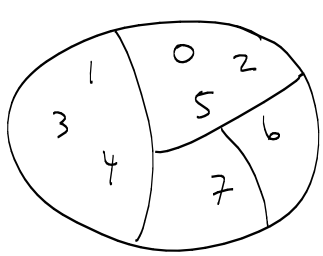
a						
	0	0	1	1	0	1
	0	0	0	0	0	1
	1	1	0	1	0	0
	1	0	1	1	0	0
	0	1	0	0	1	0
	0	0	0	0	1	1

any boolean matrix defines a relation

Equivalence Relation

(much more constrained)

Example: partition of a set



	0	1	2	3	4	5	6	7
0	1	0	1	0	0	1	0	0
1	0	1	0	1	1	0	0	0
2	1	0	1	0	0	1	0	0
3	0	1	0	1	1	0	0	0
4	0	1	0	1	1	0	0	0
5	1	0	1	0	0	1	0	0
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1

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

Equivalence Relation (MATH 240)

http://en.wikipedia.org/wiki/Equivalence_relation

reflexivity

for all $a \in S$, $(a, a) \in R$

symmetry

for all $a, b \in S$, $(a, b) \in R \Rightarrow (b, a) \in R$

transitivity

for all $a, b, c \in S$, $(a, b) \in R$ and $(b, c) \in R$
$\Rightarrow (a, c) \in R$

Java

equals( ) defines an equivalence relation on objects

http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)

reflexive — a.equals(a) returns true

symmetric — a.equals(b) == b.equals(a)

transitive — a.equal(b) and b.equals(c) $\Longrightarrow$ a.equals(c)

Example

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

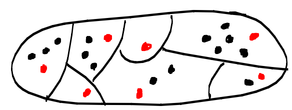
- there is a path of length 0 from u to u, for all vertices u
- there is a path from u to v iff there is a path from v to u.
- if {there is a path from v_i to v_j and there is a path from v_j to v_k} then there is a path from v_i to v_k

lecture 5

Disjoint Sets

- Equivalence Relations
- Union - Find (dynamic equivalence relation)

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.

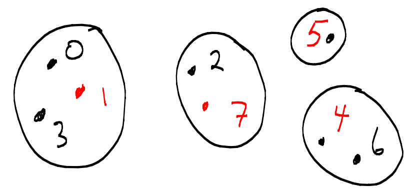
We use it to build up the partition.

- does nothing if i & j already belong to the same set
- otherwise, we need a policy for deciding who is the representative of the new (merged) set

Disjoint Sets data structures:
(1) "quick find"

Let rep[i] ∈ {0, 1, 2, ... n-1} be the representative of the set containing i.

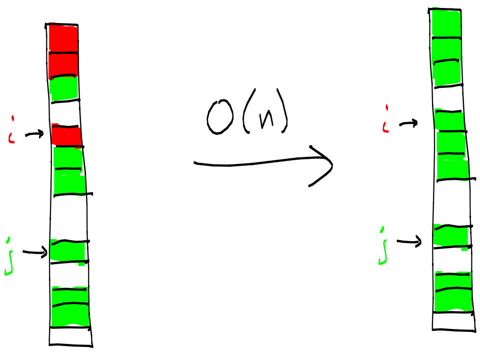
	rep[ ]
0	1
1	1
2	7
3	1
4	4
5	5
6	4
7	7



n = 8

"quick find" (but slow union)

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




```
union(i, j) {
   if rep[i] ≠ rep[j]
      for each k in 0,..n-1
         if rep[k] == rep[i]
            rep[k] = rep[j]
}
```

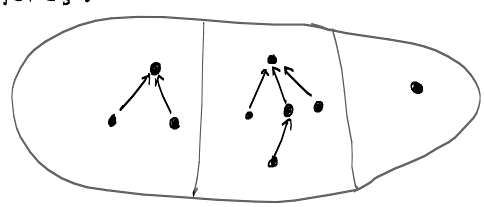
This seems to work. But there's an error!

```
union(i, j) {
   if rep[i] ≠ rep[j]
      old rep i = rep[i]
      for each k in 0,..n-1
         if rep[k] == old rep i
            rep[k] = rep[j]
}
```

This is correct, but its too slow. O(n) per union.

Disjoint Sets data structures: (2) "quick union"

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



The roots are the representatives.

i.e. find(i) == findrep(i) == findroot(i)

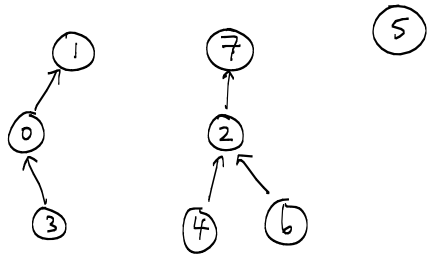
Each node points to its parent.

Use an array p[ ] i.e. parent[ ]

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

p[ ]

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



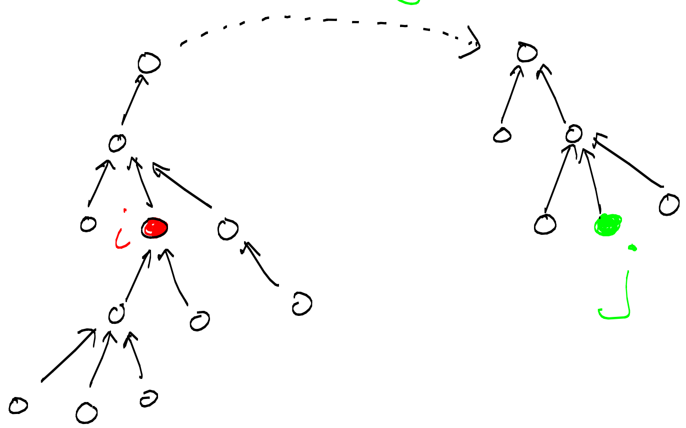

```
find(i) {
   if p[i] == -1
      return i
   else return find(p[i])
}

union(i, j) {
   if find(i) != 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)

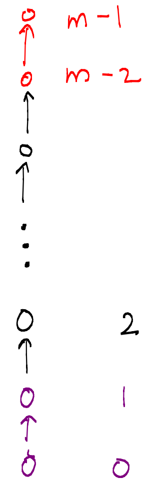
union (2,3)

union (3,4)

⋮

union (m-2, m-1)

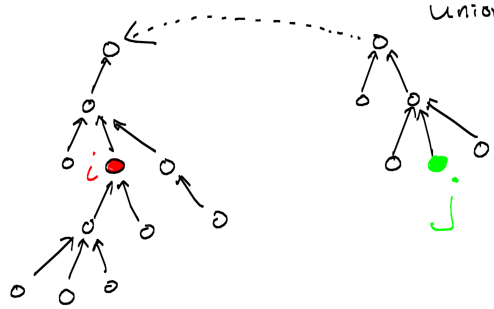
find(0) is $O(m)$



"Union by size"

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

union(i, j)



"Union by size"

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 $\log n$.

Equivalent Claim: a union-by-height tree of height h has at least $n \geq 2^h$ nodes.

Claim A union-by-height tree of height h has at least $n \geq 2^h$ nodes.

Proof (by induction)

Base case: $h = 0 \Rightarrow$ tree has 1 node ✓

Induction hypothesis: state claim for $h = k$.

Induction step: show claim for $h = k+1$

[See Exercises]

		find(i)	union(i,j)
quick find		O(1)	O(n)
quick union	union by size	O(log n)	O(log n)
	union by height	O(log n)	O(log n)

recall: union makes 2 calls to find

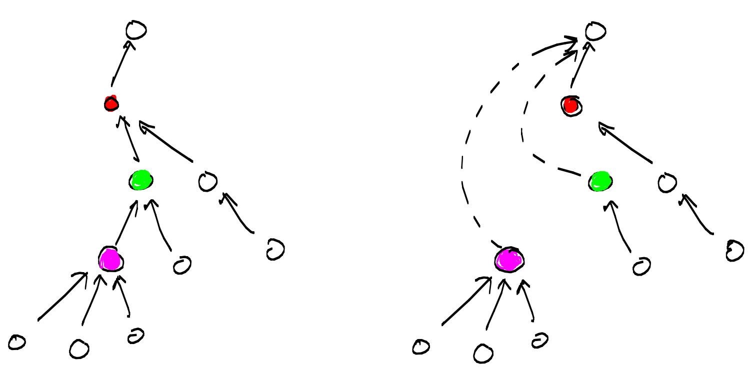
Path Compression (Quicker union)

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

```
p[i] = find(p[i])
return p[i]
```

find(●)

BEFORE   AFTER



[Advanced Topic: not on exams]

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

What is that?

see Roughgarden Algorithms 2: Advanced Union-find
https://class.coursera.org/algo2-2012-001/lecture/115 ("Union by rank")

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.

n	$\log^* n$	
$[0,1]$	0	
$(1,2]$	1	
$(2,2^2]$	2	
$(4,16] = (2^2, 2^{(2^2)}]$	3	2^{16}
$(16, 2^{16}] = (2^{2^2}, 2^{2^{2^2}}]$	4	$\approx 65{,}000$
$(2^{16}, 2^{(2^{16})}] = (2^{2^{2^2}}, 2^{2^{2^{2^2}}}]$	5	
	⋮	

$\log^* n \approx O(1)$ since it grows so slowly

Announcements

- T.A. office hours for A1
- T.A. office hours in general (options)
- A1 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.)