

# Disjoint Sets and Dynamic Equivalence Relations

CSE 373  
Data Structures and Algorithms

10/22/2012

1

## Today's Outline

- **Announcements**
  - Assignment #3 due Thurs 10/25 at 11pm.
- **Today's Topics:**
  - Disjoint Sets & Dynamic Equivalence

10/22/2012

2

## Motivation

Some kinds of data analysis require keeping track of transitive relations.

Equivalence relations are one family of transitive relations.

Grouping pixels of an image into colored regions is one form of data analysis that uses “dynamic equivalence relations”.

Creating mazes without cycles is another application.

Later we'll learn about “minimum spanning trees” for networks, and how the dynamic equivalence relations help out in computing spanning trees.

10/22/2012

3

## Disjoint Sets

- Two sets  $S_1$  and  $S_2$  are disjoint if and only if they have **no elements in common**.
- $S_1$  and  $S_2$  are disjoint iff  $S_1 \cap S_2 = \emptyset$   
(the intersection of the two sets is the empty set)

For example  $\{a, b, c\}$  and  $\{d, e\}$  are disjoint.

But  $\{x, y, z\}$  and  $\{t, u, x\}$  are not disjoint.

10/22/2012

4

## Equivalence Relations

- A binary relation  $R$  on a set  $S$  is an **equivalence relation** provided it is reflexive, symmetric, and transitive:
- Reflexive -  $R(a,a)$  for all  $a$  in  $S$ .
- Symmetric -  $R(a,b) \rightarrow R(b,a)$
- Transitive -  $R(a,b) \wedge R(b,c) \rightarrow R(a,c)$

Is  $\leq$  an equivalence relation on integers?

Is “is connected by roads” an equivalence relation on cities?

10/22/2012

5

## Induced Equivalence Relations

- Let  $S$  be a set, and let  $P$  be a partition of  $S$ .  
 $P = \{ S_1, S_2, \dots, S_k \}$   
 $P$  being a partition of  $S$  means that:  
 $i \neq j \rightarrow S_i \cap S_j = \emptyset$  and  
 $S_1 \cup S_2 \cup \dots \cup S_k = S$
- $P$  induces an equivalence relation  $R$  on  $S$ :  
 $R(a,b)$  provided  $a$  and  $b$  are in the **same subset** (same element of  $P$ ).

So given any partition  $P$  of a set  $S$ , there is a corresponding equivalence relation  $R$  on  $S$ .

10/22/2012

6

## Example

- $S = \{a, b, c, d, e\}$   
 $P = \{S_1, S_2, S_3\}$   
 $S_1 = \{a, b, c\}, S_2 = \{d\}, S_3 = \{e\}$   
P being a partition of S means that:  
 $i \neq j \rightarrow S_i \cap S_j = \emptyset$  and  
 $S_1 \cup S_2 \cup \dots \cup S_k = S$
- P induces an equivalence relation R on S:  
 $R = \{ (a,a), (b,b), (c,c), (a,b), (b,a), (a,c), (c,a),$   
 $(b,c), (c,b),$   
 $(d,d),$   
 $(e,e) \}$

10/22/2012

7

## Introducing the UNION-FIND ADT

- Also known as the Disjoint Sets ADT or the Dynamic Equivalence ADT.
- There will be a set S of elements that does not change.
- We will start with a partition  $P_0$ , but we will modify it over time by combining sets.
- The combining operation is called "UNION"
- Determining which set (of the current partition) an element of S belongs to is called the "FIND" operation.

10/22/2012

8

## Example

- Maintain a set of **pairwise disjoint**\* sets.  
–  $\{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}$
- Each set has a unique name: one of its members  
–  $\{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}$

\*Pairwise Disjoint: For any two sets you pick, their intersection will be empty)

10/22/2012

9

## Union

- Union(x,y) – take the union of two sets named x and y  
–  $\{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}$   
– Union(5,1)  
 $\{3,5,7,1,6\}, \{4,2,8\}, \{9\}$

To perform the union operation, we replace sets x and y by  $(x \cup y)$

10/22/2012

10

## Find

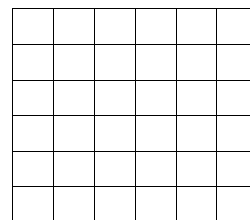
- Find(x) – return the name of the set containing x.  
–  $\{3,5,7,1,6\}, \{4,2,8\}, \{9\}$   
– Find(1) = 5  
– Find(4) = 8

10/22/2012

11

## Application: Building Mazes

- Build a random maze by erasing edges.

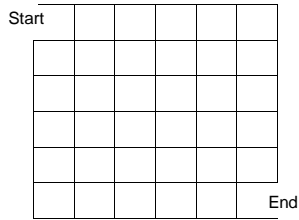


10/22/2012

12

## Building Mazes (2)

- Pick Start and End

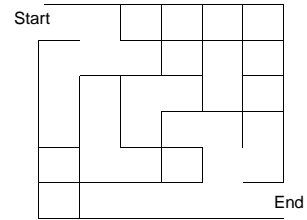


10/22/2012

13

## Building Mazes (3)

- Repeatedly pick random edges to delete.



10/22/2012

14

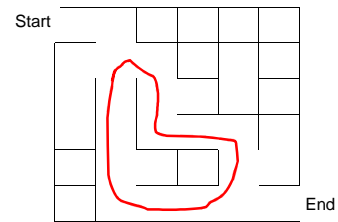
## Desired Properties

- None of the boundary is deleted
- Every cell is reachable from every other cell.
- Only one path from any one cell to another (There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.)

10/22/2012

15

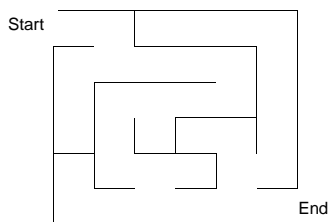
## A Cycle



10/22/2012

16

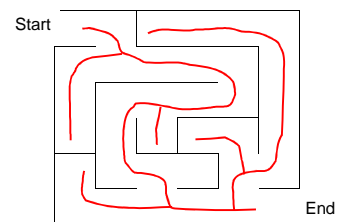
## A Good Solution



10/22/2012

17

## A Hidden Tree



10/22/2012

18

## Number the Cells

We have **disjoint sets**  $P = \{ \{1\}, \{2\}, \{3\}, \{4\}, \dots, \{36\} \}$  each cell is unto itself.  
 We have all possible edges  $E = \{ (1,2), (1,7), (2,8), (2,3), \dots \}$  60 edges total.

Start	1	2	3	4	5	6
	7	8	9	10	11	12
	13	14	15	16	17	18
	19	20	21	22	23	24
	25	26	27	28	29	30
	31	32	33	34	35	36
						End

10/22/2012

19

## Basic Algorithm

- $P$  = set of **disjoint sets** of connected cells
- $E$  = set of edges
- $Maze$  = set of maze edges (initially empty)

```

While there is more than one set in  $P$  {
  pick a random edge (x,y) and remove from  $E$ 
  u := Find(x);
  v := Find(y);
  if u ≠ v then // removing edge (x,y) connects previously non-
                // connected cells x and y - leave this edge removed!
    Union(u,v)
  else // cells x and y were already connected, add this
        // edge to set of edges that will make up final maze.
    add (x,y) to  $Maze$ 
}
    
```

All remaining members of  $E$  together with  $Maze$  form the maze

10/22/2012

20

## Example Step

Pick (8,14)

Start	1	2	3	4	5	6
	7	8	9	10	11	12
	13	14	15	16	17	18
	19	20	21	22	23	24
	25	26	27	28	29	30
	31	32	33	34	35	36
						End

$P$   
 {1,2,7,8,9,13,19}  
 {3}  
 {4}  
 {5}  
 {6}  
 {10}  
 {11,17}  
 {12}  
 {14,20,26,27}  
 {15,16,21}  
 .  
 .  
 {22,23,24,29,30,32}  
 {33,34,35,36}

10/22/2012

21

## Example

$P$		$P$
{1,2,7,8,9,13,19}		{1,2,7,8,9,13,19,14,20,26,27}
{3}	Find(8) = 7	{3}
{4}	Find(14) = 20	{4}
{5}	→	{5}
{6}	Union(7,20)	{6}
{10}		{10}
{11,17}		{11,17}
{12}		{12}
{14,20,26,27}		{15,16,21}
{15,16,21}		.
.		.
.		{22,23,24,29,30,32}
{22,23,24,29,30,32}		{33,34,35,36}
33,34,35,36		

10/22/2012

22

## Example

Pick (19,20)

Start	1	2	3	4	5	6
	7	8	9	10	11	12
	13	14	15	16	17	18
	19	20	21	22	23	24
	25	26	27	28	29	30
	31	32	33	34	35	36
						End

$P$   
 {1,2,7,8,9,13,19,  
 14,20,26,27}  
 {3}  
 {4}  
 {5}  
 {6}  
 {10}  
 {11,17}  
 {12}  
 {15,16,21}  
 .  
 .  
 {22,23,24,29,30,32}  
 {33,34,35,36}

10/22/2012

23

## Example at the End

Start	1	2	3	4	5	6
	7	8	9	10	11	12
	13	14	15	16	17	18
	19	20	21	22	23	24
	25	26	27	28	29	30
	31	32	33	34	35	36
						End

$P$   
 {1,2,3,4,5,6,7,... 36}

—  $E$   
 —  $Maze$

10/22/2012

24

## Implementing the Disjoint Sets ADT

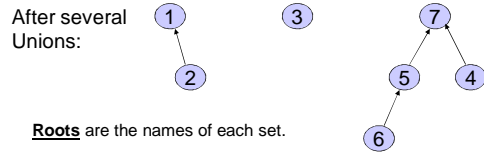
- $n$  elements,  
Total Cost of:  $m$  finds,  $\leq n-1$  unions
- Target complexity:  $O(m+n)$   
*i.e.  $O(1)$  amortized*
- $O(1)$  worst-case for find as well as union would be great, but...  
*Known result: both find and union cannot be done in worst-case  $O(1)$  time*

10/22/2012

25

## Up-Tree for Disjoint Union/Find

Initial state: ① ② ③ ④ ⑤ ⑥ ⑦

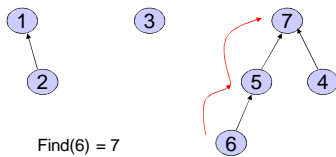


10/22/2012

26

## Find Operation

**Find(x)** - follow  $x$  to the root and return the root

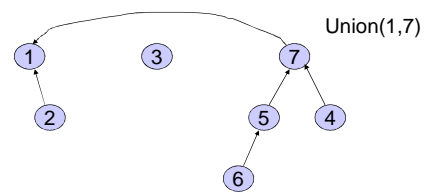


10/22/2012

27

## Union Operation

**Union(x,y)** - assuming  $x$  and  $y$  are roots, point  $y$  to  $x$ .



10/22/2012

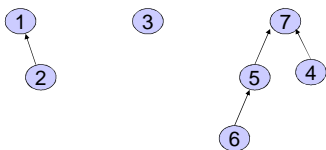
28

## Simple Implementation

- Array of indices

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

Up[x] = 0 means  
 $x$  is a root.



10/22/2012

29

## Implementation

```
int Find(int x) {
    while(up[x] != 0) {
        x = up[x];
    }
    return x;
}
```

```
void Union(int x, int y) {
    up[y] = x;
}
```

*runtime for Union():*

*runtime for Find():*

*runtime for  $m$  Finds and  $n-1$  Unions:*

10/22/2012

30

## Find Solutions

### Recursive

```
Find(up[] : integer array, x : integer) : integer {
  //precondition: x is in the range 1 to size//
  if up[x] = 0 then return x
  else return Find(up,up[x]);
}
```

### Iterative

```
Find(up[] : integer array, x : integer) : integer {
  //precondition: x is in the range 1 to size//
  while up[x] ≠ 0 do
    x := up[x];
  return x;
}
```

10/22/2012

31

## Now this doesn't look good ☹️

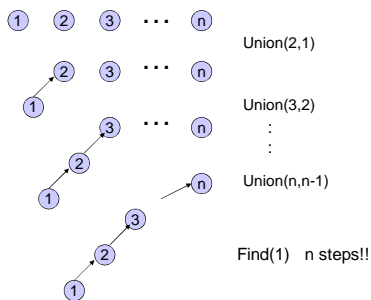
Can we do better? *Yes!*

1. Improve **union** so that **find** only takes  $\Theta(\log n)$ 
  - **Union-by-size**
  - Reduces complexity to  $\Theta(m \log n + n)$
2. Improve **find** so that it becomes even better!
  - **Path compression**
  - Reduces complexity to almost  $\Theta(m + n)$

10/22/2012

32

## A Bad Case

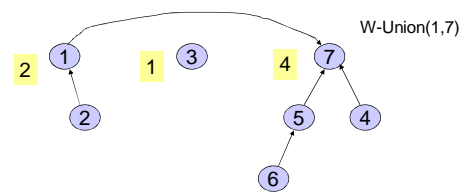


10/22/2012

33

## Weighted Union

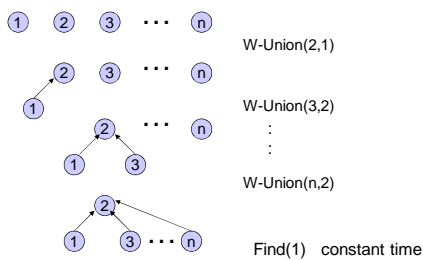
- **Weighted Union**
  - Always point the *smaller* (total # of nodes) tree to the root of the larger tree



10/22/2012

34

## Example Again



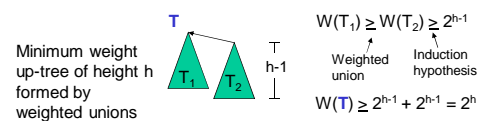
10/22/2012

35

## Analysis of Weighted Union

With weighted union an up-tree of height  $h$  has weight *at least*  $2^h$ .

- **Proof by induction**
  - **Basis:**  $h = 0$ . The up-tree has one node,  $2^0 = 1$
  - **Inductive step:** Assume true for all  $h' < h$ .



10/22/2012

36

## Analysis of Weighted Union (cont)

Let T be an up-tree of weight n formed by weighted union. Let h be its height.

$$n \geq 2^h$$

$$\log_2 n \geq h$$

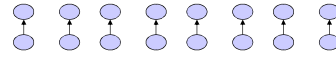
- Find(x) in tree T takes  $O(\log n)$  time.
  - Can we do better?

10/22/2012

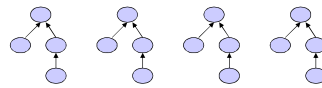
37

## Worst Case for Weighted Union

n/2 Weighted Unions



n/4 Weighted Unions

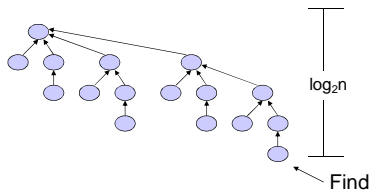


10/22/2012

38

## Example of Worst Case (cont')

After  $n/2 + n/4 + \dots + 1$  Weighted Unions:

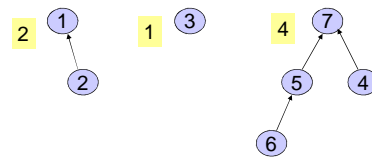


If there are  $n = 2^k$  nodes then the longest path from leaf to root has length k.

10/22/2012

39

## Array Implementation



	1	2	3	4	5	6	7
up	-1	1	-1	7	5	-1	
weight	2		1				4

10/22/2012

40

## Weighted Union

```

W-Union(i, j : index){
  //i and j are roots           new runtime for Union():
  wi := weight[i];
  wj := weight[j];
  if wi < wj then
    up[i] := j;                 new runtime for Find():
    weight[j] := wi + wj;
  else
    up[j] := i;
    weight[i] := wi + wj;
}

```

runtime for m finds and n-1 unions =

10/22/2012

41

## Nifty Storage Trick

- Use the same array representation as before
- Instead of storing -1 for the root, simply store **-size**

[Read section 8.4, page 299]

10/22/2012

43

## How about Union-by-height?

- Can still guarantee  $O(\log n)$  worst case depth

*Left as an exercise!*

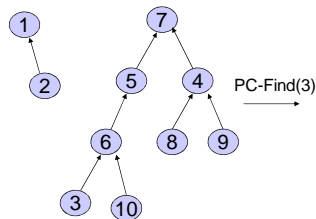
- Problem: Union-by-height doesn't combine very well with the new find optimization technique we'll see next

10/22/2012

44

## Path Compression

- On a Find operation point all the nodes on the search path directly to the root.

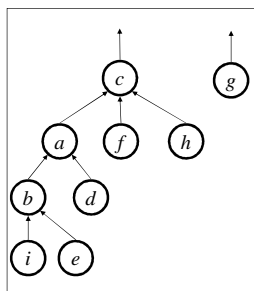


10/22/2012

45

### Student Activity

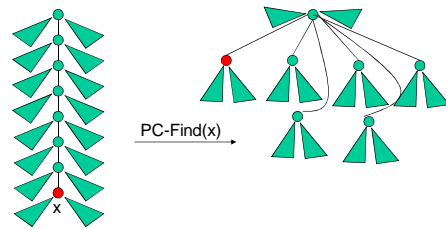
## Draw the result of Find(e):



10/22/2012

47

## Self-Adjustment Works



10/22/2012

48

## Path Compression Find

```

PC-Find(i : index) {
  r := i;
  while up[r] ≠ -1 do //find root
    r := up[r];

  // Assert: r= the root, up[r] = -1
  if i ≠ r then // if i was not a root

    temp := up[i];
    while temp ≠ r do //compress path
      up[i] := r;
      i := temp;
      temp := up[temp]

  return(r)
}
    
```

*(New?) runtime for Find:*

10/22/2012

49

## Interlude: A Really Slow Function

**Ackermann's function** is a really big function  $A(x, y)$  with inverse  $\alpha(x, y)$  which is really small

How fast does  $\alpha(x, y)$  grow?

$\alpha(x, y) = 4$  for  $x$  far larger than the number of atoms in the universe ( $2^{300}$ )

$\alpha$  shows up in:

- Computation Geometry (surface complexity)
- Combinatorics of sequences

10/22/2012

50



## A More Comprehensible Slow Function

**$\log^* x$  = number of times you need to compute  
log to bring value down to at most 1**

E.g.  $\log^* 2 = 1$   
 $\log^* 4 = \log^* 2^2 = 2$   
 $\log^* 16 = \log^* 2^{2^2} = 3$  ( $\log \log \log 16 = 1$ )  
 $\log^* 65536 = \log^* 2^{2^{2^2}} = 4$  ( $\log \log \log \log 65536 = 1$ )  
 $\log^* 2^{65536} = \dots\dots\dots = 5$

Take this:  $\alpha(m,n)$  grows even slower than  $\log^* n$  !!

10/22/2012

51

## Complex Complexity of Union-by-Size + Path Compression

Tarjan proved that, with these optimizations,  $p$  union and find operations on a set of  $n$  elements have worst case complexity of  $O(p \cdot \alpha(p, n))$

For *all practical purposes* this is amortized constant time:  
 $O(p \cdot 4)$  for  $p$  operations!

- Very complex analysis – worse than splay tree analysis etc. that we skipped!

10/22/2012

52

## Disjoint Union / Find with Weighted Union and PC

- Worst case time complexity for a W-Union is  $O(1)$  and for a PC-Find is  $O(\log n)$ .
- Time complexity for  $m \geq n$  operations on  $n$  elements is  $O(m \log^* n)$  where  $\log^* n$  is a very slow growing function.
  - $\log^* n < 7$  for all reasonable  $n$ . Essentially constant time per operation!

10/22/2012

53

## Amortized Complexity

- For disjoint union / find with weighted union and path compression.
  - average time per operation is essentially a constant.
  - worst case time for a PC-Find is  $O(\log n)$ .
- An individual operation can be costly, but over time the average cost per operation is not.

10/22/2012

54