Disjoint Sets I
Chapter 8 in Weiss

CSE 326
Data Structures
Ruth Anderson

Today's Outline

- **Announcements**
  - **Project 3 partner selection due Mon Feb 22 by 11pm, DO NOT WAIT UNTIL THEN TO START!**
  - **Written Homework #5 due Friday 2/19**
- **Today's Topics:**
  - **Hash Tables**
  - **Disjoint Sets**

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.

Disjoint Sets

- Two sets S_1 and S_2 are disjoint if and only if they have no elements in common.

(the intersection of the two sets is the empty set)

- S_1 and S_2 are disjoint iff $S_1 \cap S_2 = \varnothing$

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

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

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?

Induced Equivalence Relations

- Let S be a set, and let P be a partition of S.

  $P = \{ S_1, S_2, \ldots, S_k \}$

  P being a partition of S means that:

  $i \neq j \rightarrow S_i \cap S_j = \varnothing$ and

  $S_1 \cup S_2 \cup \ldots \cup S_k = S$
- 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.

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:

  $i \neq j \rightarrow S_i \cap S_j = \varnothing$ and

  $S_1 \cup S_2 \cup \ldots \cup S_k = S$
- 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) }

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.

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)

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},

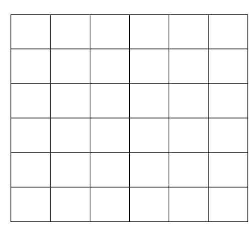
To perform the union operation, we replace sets x and y by $(x \cup y)$

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

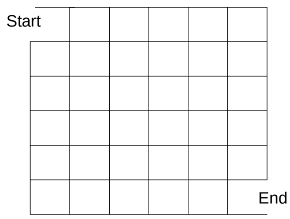
Application: Building Mazes

- Build a random maze by erasing edges.



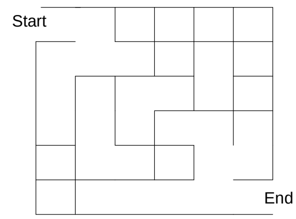
Building Mazes (2)

- Pick Start and End



Building Mazes (3)

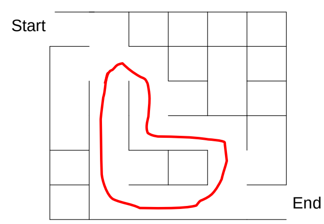
- Repeatedly pick random edges to delete.



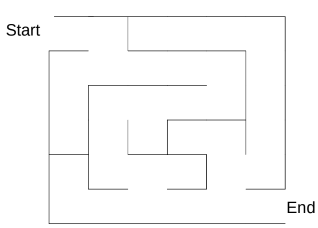
Desired Properties

- 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.)

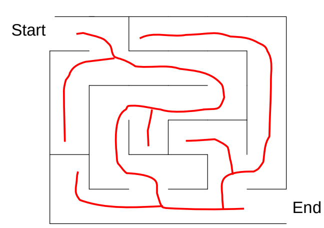
A Cycle



A Good Solution



A Hidden Tree



Number the Cells

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

Algorithm - idea

1. Choose wall at random.
   → ***Boundary walls are not in wall list, because we cannot delete them***
2. Erase wall if the neighbors are in disjoint sets.
   → ***Avoids cycles***
3. Take union of those sets.
4. Repeat until there is only one set.
   → ***Every cell reachable from every other cell.***

We want to check if two nodes x and y are in the same set.
How can I do this using unions and finds?

Activity

Basic Algorithm

- P = set of 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
```

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}

Activity

Example

P
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Find(8) = 7
Find(14) = 20

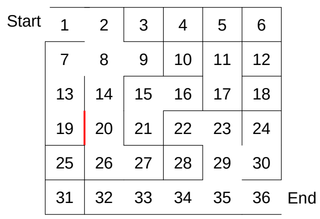
→

Union(7,20)

P
{1,2,7,8,9,13,19,14,20 26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Example

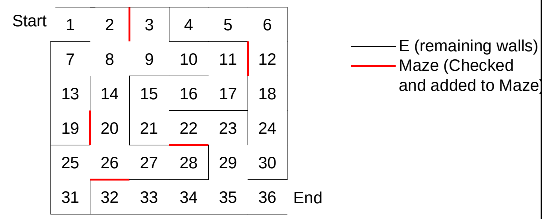
Pick (19,20)



P
{1,2,7,8,9,13,19
14,20,26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Example at the End

P
{1,2,3,4,5,6,7,… 36}



Implementing the DS ADT

- n elements,
  Total Cost of: m finds, $\leq n$-1 unions

can there be more unions?

- Target complexity: $O(m+n)$
  i.e. $O(1)$ amortized

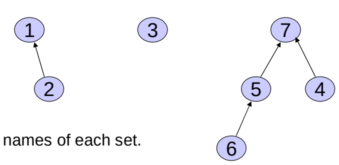
- $O(1)$ worst-case for find as well as union would be great, but…
  Known result: both find and union *cannot* be done in worst-case $O(1)$ time

Data Structure for the DS ADT

Up-Tree for Disjoint Union/Find

Initial state:

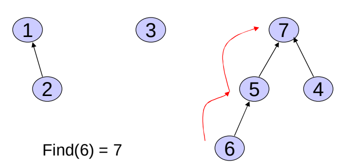
After several Unions:



Roots are the names of each set.

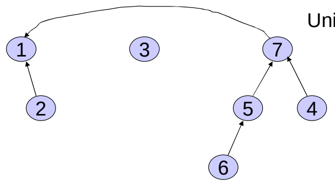
Find Operation

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



Union Operation

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

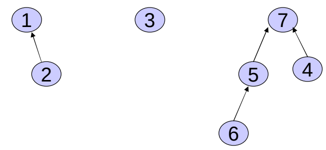


Simple Implementation

- 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.



Implementation

```
int Find(int x) {

  while(up[x] != 0) {
    x = up[x];
  }

  return x;
}
```

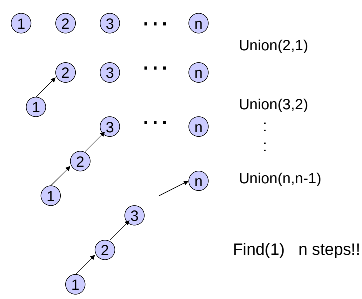
```
void Union(int x, int y) {
  up[y] = x;
}
```

runtime for Union():

runtime for Find():

runtime for m Finds and n-1 Unions:

A Bad Case



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;
}
```


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 $\Theta(m + n)$