Union-Find Algorithms

- network connectivity
- quick union
- quick find
- weighted
- applications

Subtext of today's lecture (and this course)

Steps to developing an usable algorithm.

- Define the problem.
- Find an algorithm to solve it.
- Fast enough?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method

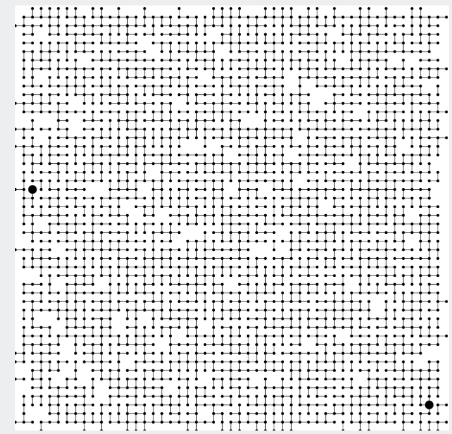
Mathematical models and computational complexity

READ Chapter One of Algs in Java

Network connectivity

Basic abstractions

- set of objects
- union command: connect two objects
- find query: is there a path connecting one object to another?



Objects

Union-find applications involve manipulating objects of all types.

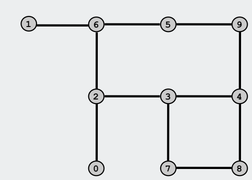
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Variable name aliases.
- Pixels in a digital photo.
- Metallic sites in a composite system.

stay tuned

When programming, convenient to name them 0 to N-1.

- Details not relevant to union-find.
- Integers allow quick access to object-related info.
- Could use symbol table to translate from object names

use as array index



Union-find abstractions

Simple model captures the essential nature of connectivity.

- Objects.

`0 1 2 3 4 5 6 7 8 9` — grid points

- Disjoint sets of objects.

`0 1 { 2 3 9 } { 5 6 } 7 { 4 8 }` — subsets of connected grid points

- Find query: are objects 2 and 9 in the same set?

`0 1 { 2 3 9 } { 5-6 } 7 { 4-8 }` — are two grid points connected?

- Union command: merge sets containing 3 and 8.

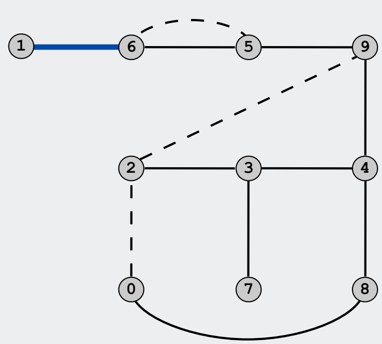
`0 1 { 2 3 4 8 9 } 7` — add a connection between two grid points

Network connectivity example

Input: sequence of object pairs

- do find query for each pair
- if connected, ignore
- otherwise, do union command (and print)

in	out	evidence
3 4	3 4	
4 9	4 9	
8 0	8 0	
2 3	2 3	
5 6	5 6	
2 9		(2-3-4-9)
5 9	5 9	
7 3	7 3	
4 8	4 8	
5 6		(5-6)
0 2		(2-3-4-8-0)
6 1	6 1	



Connected components

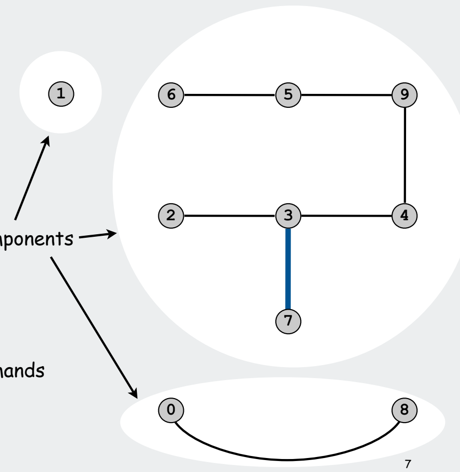
Connected component: set of mutually connected vertices

Each union command reduces by 1 the number of components

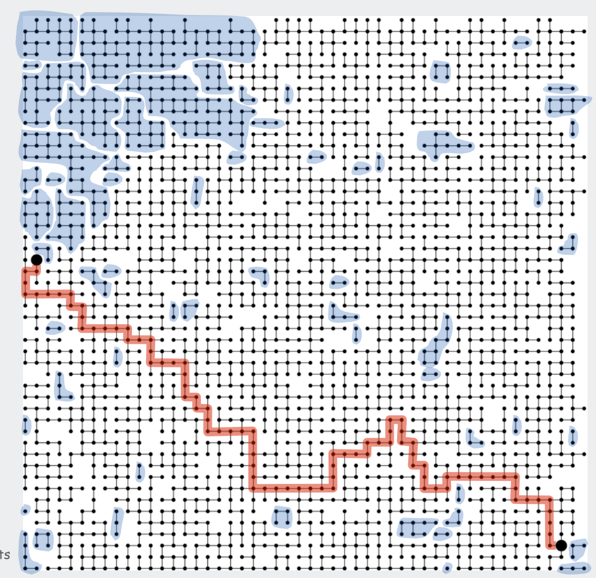
in	out
3 4	3 4
4 9	4 9
8 0	8 0
2 3	2 3
5 6	5 6
2 9	
5 9	5 9
7 3	7 3

3 = 10-7 components

7 union commands



Network Connectivity



63 components

Union-find abstractions

- Objects.
- Disjoint sets of objects.
- Find queries: are two objects in the same set?
- Union commands: replace sets containing two items by their union

Goal. Design efficient data structure for union-find.

- Find queries and union commands may be intermixed.
- Number of operations M can be huge.
- Number of objects N can be huge.

Quick-Find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Quick-Find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

id[3] = 9; id[6] = 6
3 and 6 not connected

Union. To merge components containing `p` and `q`, change all entries with `id[p]` to `id[q]`.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	6	6	6	6	6	7	8	6

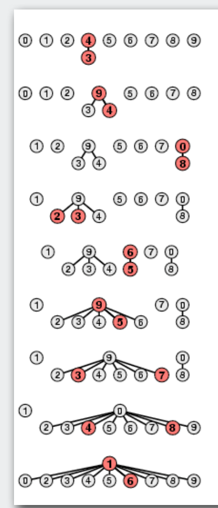
union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

problem: many values can change

Quick-Find: Example

3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	9	9	5	6	7	8	9
8-0	0	1	2	9	9	5	6	7	0	9
2-3	0	1	9	9	9	5	6	7	0	9
5-6	0	1	9	9	9	6	6	7	0	9
5-9	0	1	9	9	9	9	9	7	0	9
7-3	0	1	9	9	9	9	9	9	0	9
4-8	0	1	0	0	0	0	0	0	0	0
6-1	1	1	1	1	1	1	1	1	1	1

problem: many values can change



Quick-Find: Java Implementation


```java
public class QuickFind
{
   private int[] id;

   public QuickFind(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++)
         id[i] = i;
   }

   public boolean find(int p, int q)
   {
      return id[p] == id[q];
   }

   public void unite(int p, int q)
   {
      int pid = id[p];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = id[q];
   }
}
```

set id of each object to itself

1 operation

N operations

Quick-find is too slow

Quick-find algorithm may take ~MN steps to process M union commands on N objects

Rough standard (for now).
- 10^9 operations per second.
- 10^9 words of main memory.
- Touch all words in approximately 1 second.

a truism (roughly) since 1950 !

Ex. Huge problem for quick-find.
- 10^{10} edges connecting 10^9 nodes.
- Quick-find takes more than 10^{18} operations.
- 30+ years of computer time!

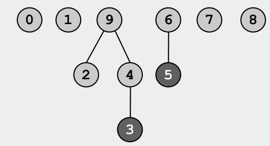
Paradoxically, quadratic algorithms get worse with newer equipment.
- New computer may be 10x as fast.
- But, has 10x as much memory so problem may be 10x bigger.
- With quadratic algorithm, takes 10x as long!

Quick-Union [lazy approach]

Data structure.
- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



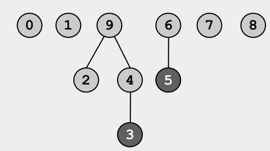
3's root is 9; 5's root is 6

Quick-Union [lazy approach]

Data structure.
- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



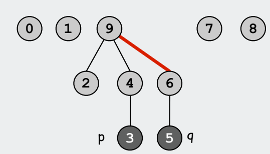
Find. Check if p and q have the same root.

3's root is 9; 5's root is 6
3 and 5 are not connected

Union. Set the id of q's root to the id of p's root.

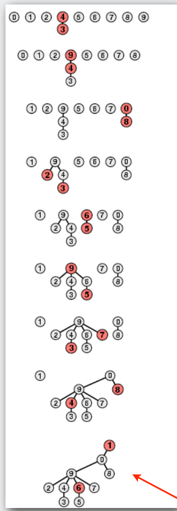
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	9	7	8	9

only one value changes



Quick-Union: Example

3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



problem: trees can get tall

Quick union is also too slow

Quick-find defect.

- Union too expensive (N steps).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N steps)

Data Structure	Union	Find
Quick-find	N	1
Quick-union	1	N ← worst case

assumes find already done

Quick-Union: Java Implementation


```
public class QuickUnion
{
   private int[] id;

   public QuickUnion(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   private int root(int i)
   {
      while (i != id[i]) i = id[i];
      return i;
   }


   public boolean find(int p, int q)
   {
      return root(p) == root(q);
   }

   public void unite(int p, int q)
   {
      int i = root(p);
      int j = root(q);
      id[i] = j;
   }
}
```

time proportional to depth of i

time proportional to depth of p and q

time proportional to depth of p and q

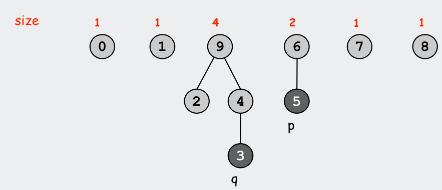
Weighted Quick-Union

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each component.
- Balance by linking small tree below large one.

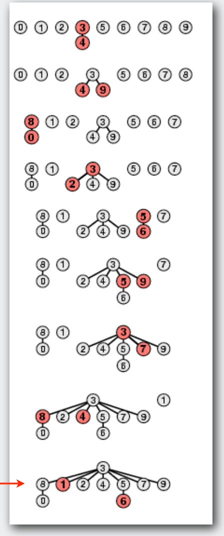
Ex. Union of 5 and 3.

- Quick union: link 9 to 6.
- Weighted quick union: link 6 to 9.



Weighted quick-union example

3-4	0 1 2 3 3 5 6 7 8 9
4-9	0 1 2 3 3 5 6 7 8 3
8-0	8 1 2 3 3 5 6 7 8 3
2-3	8 1 3 3 3 5 6 7 8 3
5-6	8 1 3 3 3 5 5 7 8 3
5-9	8 1 3 3 3 3 5 7 8 3
7-3	8 1 3 3 3 3 5 3 8 3
4-8	8 1 3 3 3 3 5 3 3 3
6-1	8 3 3 3 3 3 5 3 3 3



no problem: trees stay flat

- connectivity
- quick union
- quick find
- qfwpc
- applications

Weighted Quick-Union: Java Implementation

Java implementation.

- Almost identical to quick-union.
- Maintain extra array `sz[]` to count number of elements in the tree rooted at i.

Find. Identical to quick-union.

Union. Modify quick-union to

- merge smaller tree into larger tree
- update the `sz[]` array.

```
if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else sz[i] < sz[j]  { id[j] = i; sz[i] += sz[j]; }
```


- connectivity
- quick union
- quick find
- qfwpc
- applications

Weighted quick-union analysis

Analysis.

- Find: takes time proportional to depth of `p` and `q`.
- Union: takes constant time, given roots.
- Fact: depth is at most lg N. [needs proof]

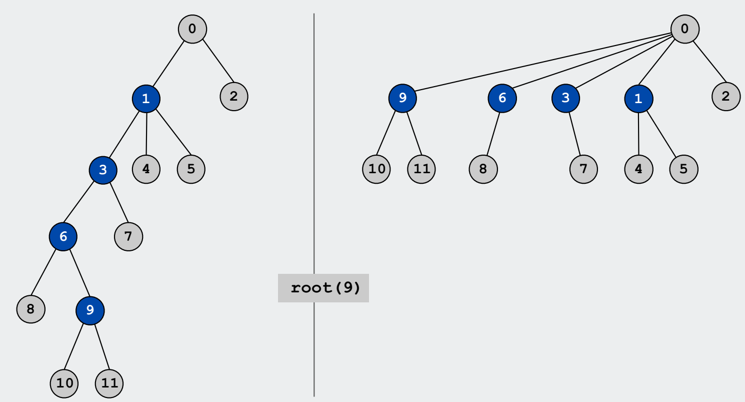
Data Structure	Union	Find
Quick-find	N	1
Quick-union	1	N
Weighted QU	lg N	lg N

Stop at guaranteed acceptable performance? No, easy to improve further.

- connectivity
- quick union
- quick find
- qfwpc
- applications

Path Compression

Path compression. Just after computing the root of `i`, set the `id` of each examined node to `root(i)`.



Weighted Quick-Union with Path Compression

Path compression.

- Standard implementation: add second loop to `root()` to set the id of each examined node to the root.
- Simpler one-pass variant: make every other node in path point to its grandparent.

```
public int root(int i)
{
   while (i != id[i])
   {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

only one extra line of code !

In practice. No reason not to! Keeps tree almost completely flat.

WQUPC performance

Theorem. Starting from an empty data structure, any sequence of M union and find operations on N objects takes O(N + M lg* N) time.

- Proof is very difficult.
- But the algorithm is still simple!

number of times needed to take the lg of a number until reaching 1

Linear algorithm?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

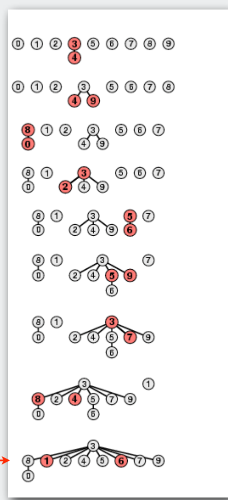
because lg* N is a constant in this universe

N	lg* N
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

Amazing fact: No algorithm can do better!

Weighted Quick-Union with Path Compression

3-4	0	1	2	3	3	5	6	7	8	9
4-9	0	1	2	3	3	5	6	7	8	3
8-0	8	1	2	3	3	5	6	7	8	3
2-3	8	1	3	3	3	5	6	7	8	3
5-6	8	1	3	3	3	5	5	7	8	3
5-9	8	1	3	3	3	3	5	7	8	3
7-3	8	1	3	3	3	3	5	3	8	3
4-8	8	1	3	3	3	3	5	3	3	3
6-1	8	3	3	3	3	3	3	3	3	3



no problem: trees stay VERY flat

Summary

Algorithm	Worst-case time
Quick-find	M N
Quick-union	M N
Weighted QU	N + M log N
Path compression	N + M log N
Weighted + path	(M + N) lg* N

M union-find ops on a set of N objects

Ex. Huge practical problem.

- 10^{10} edges connecting 10^9 nodes.
- WQUPC reduces time from 3,000 years to 1 minute.
- Supercomputer won't help much.
- Good algorithm makes solution possible.

WQUPC on Java cell phone beats QF on supercomputer!

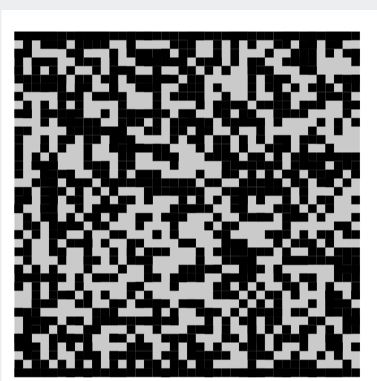
Bottom line.
WQUPC makes it possible to solve problems that could not otherwise be addressed

Union-find applications

✓ Network connectivity.
- Percolation.
- Image processing.
- Least common ancestor.
- Equivalence of finite state automata.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm.
- Games (Go, Hex)
- Compiling equivalence statements in Fortran.

Percolation

Percolation phase-transition.
- Two parallel conducting bars (top and bottom).
- Electricity flows from a site to one of its 4 neighbors if both are occupied by conductors.
- Model: each site is a conductor with probability p.



Q. What is percolation threshold p* at which charge carriers can percolate from top to bottom?

UF solution for percolation

- Initialize whole grid to be insulators
- Make top and bottom row conductors
- Make random sites conductors until `find(top, bottom)`
- conductor percentage estimates p*

top

0	0	0	0	0	0	0	0	0	0	0	0
2	3	4	5	6	0	8	9	10	11	12	0
14	15	0	0	0	0	20	21	22	23	24	0
14	14	28	29	30	31	32	33	34	35	36	0
14	39	40	1	42	43	32	45	46	1	1	49
50	1	52	1	54	55	56	57	58	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1

bottom

conductor

insulator

Percolation

Q. What is percolation threshold p* at which charge carriers can percolate from top to bottom?

A. ~ 0.592746 for square lattices.

percolation constant known only via simulation

top

0	0	0	0	0	0	0	0	0	0	0	0
2	3	4	5	6	0	8	9	10	11	12	0
14	15	0	0	0	0	20	21	22	23	24	0
14	14	28	29	30	31	32	33	34	35	36	0
14	39	40	1	42	43	32	45	46	1	1	49
50	1	52	1	54	55	56	57	58	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1

bottom

conductor

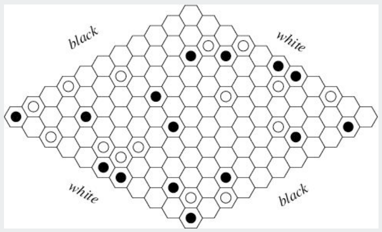
insulator

Why is UF solution better than solution in IntroProgramming 2.4?

Hex

Hex. [Piet Hein 1942, John Nash 1948, Parker Brothers 1962]

- Two players alternate in picking a cell in a hex grid.
- Black: make a black path from upper left to lower right.
- White: make a white path from lower left to upper right.



Reference: http://mathworld.wolfram.com/GameofHex.html

Goal. Algorithm to detect when a player has won.

Subtext of today's lecture (and this course)

Steps to developing an usable algorithm.

- Define the problem.
- Find an algorithm to solve it.
- Fast enough?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method

Mathematical models and computational complexity

READ Chapter One of Algs in Java