Case Study: Union-Find

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

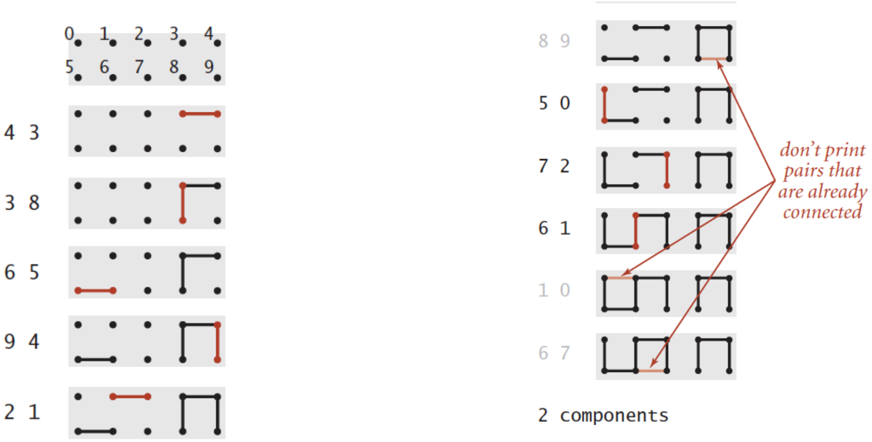
Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 1.5) and Prof. Janicki's course slides

Dynamic Connectivity Problem

You are given a sequence of pairs of integers, where each integer represents an object of some type. The pair p, q is interpreted as "p is connected to q".

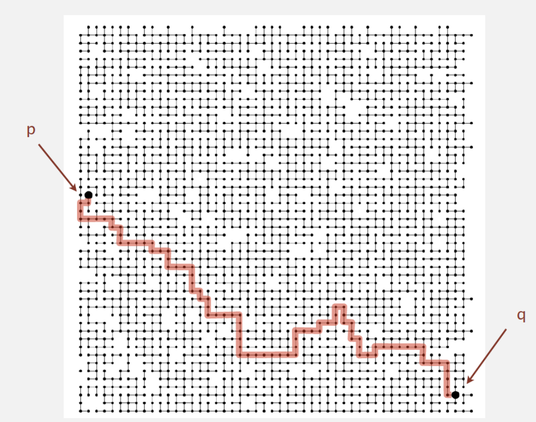
Goal:

- To write a program to filter out extraneous pairs. Particularly, when the program reads a pair p, q from the input, it should write the pair to the output only if the pairs it has seen to that point do not imply that p is connected to q.
- If the previous pairs do imply that p is connected to q, then the program should ignore the pair p, q and proceed to read in the next pair.



0 1 2 3 4
5 6 7 8 9
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
don't print pairs that are already connected
2 components

Question: Is there a path connecting p and q

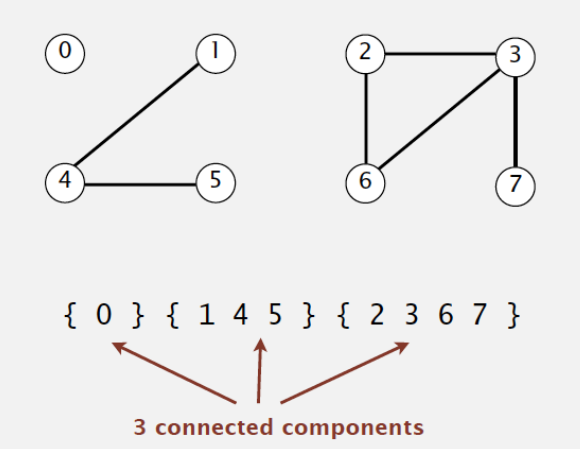


Yes!

- The "is connected to" is an *equivalence relation*, which means that it is
  - Reflexive : p is connected to p.
  - Symmetric : If p is connected to q, then q is connected to p.
  - Transitive : If p is connected to q and q is connected to r, then p is connected to r.
- An equivalence relation partitions the objects into equivalence classes.
- In this case, two objects are in the same equivalence class if and only if they are connected.

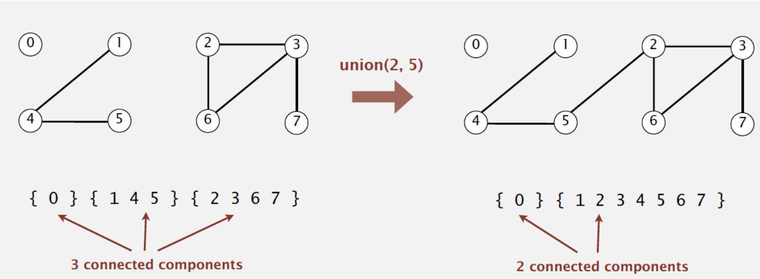
Connected Component

Connected Components: Maximal set of objects that are mutually connected.



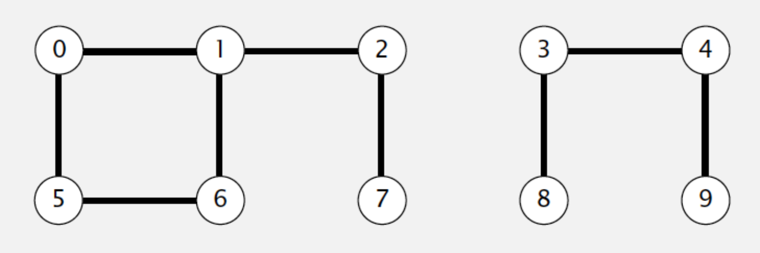
Connected Component/Union-Find Operations

- **Find:** In which component is object p?
- **Connected:** Are objects p and q in the same component?
- **Union:** Replace components containing objects p and q with their union.



UF-API

Viewing the example on slide# 2 as connected components



UF-API

public class UF		
	UF(int N)	*initialize N sites with integer names (0 to N-1)*
void	union(int p, int q)	*add connection between p and q*
int	find(int p)	*component identifier for p (0 to N-1)*
boolean	connected(int p, int q)	*return true if p and q are in the same component*
int	count()	*number of components*

Union-find API

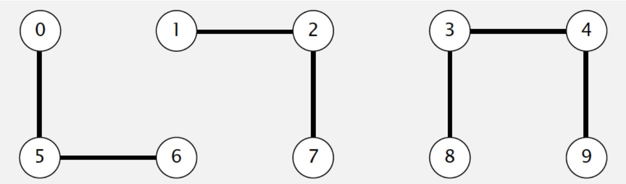
Union-Find: Eager Approach

- Data Structure: Integer array $id[]$ of length N.
- Interpretation: $id[p]$ is the ID of the component containing p.
- Initialization: Initialize $id[i] = i$
- Find: What is the id of p or what component is p in?
- Connected: Do p and q have the same id?
- Union: To merge components containing p and q, change all entries whose id equals $id[p]$ to $id[q]$.

Union-Find: Eager Approach - Example I

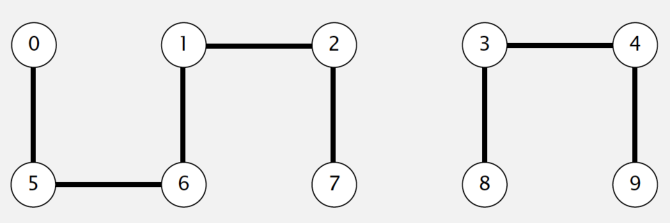
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	1	8	8	0	0	1	8	8

0, 5 and 6 are connected
1, 2, and 7 are connected
3, 4, 8, and 9 are connected



- Now we want to see if $6, 1$ are connected? – NO, as $id[6] = 0$ and $id[1] = 1$.
- Perform Union – Change all entries whose id equals $id[6] = 0$ to $id[1] = 1$.

Quick-Find - Example II



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

Quick-Find and Union

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

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

   public int find(int p)
   {  return id[p];  }

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

set id of each object to itself
(N array accesses)

return the id of p
(1 array access)

change all entries with id[p] to id[q]
(at most 2N + 2 array accesses)

Time complexity of Quick-Find and Union

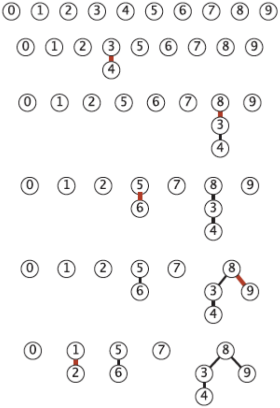
- Initialization: $O(N)$
- Find: $O(1)$
- Union: $O(N)$
- Are p and q connected?: $O(1)$
- Union is too expensive. It takes N^2 array accesses to process a sequence of N union operations on N objects.

Quick-union: Lazy Approach

- Data Structure: Integer array $id[]$ of length N.
- Interpretation: $id[i]$ is parent of i.
- Root of i is $id[id[id[\ldots id[i]\ldots]]]$ - keep going until the value does not change.
- Initialization: Initialize $id[i] = i$
- Find: what is the root of p?
- Connected: do p and q have the same root?
- Union: To merge components containing p and q, set the id of p's root to id of q's root.

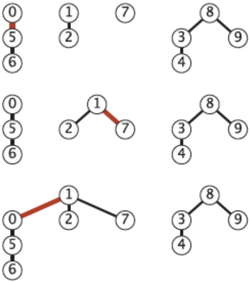
Quick-union: Lazy Approach Example

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

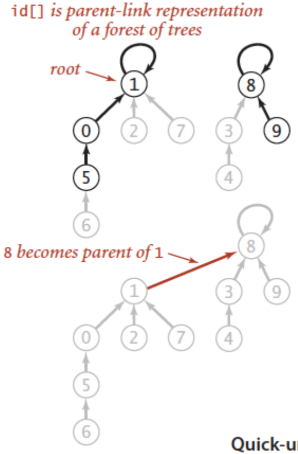


Quick-union: Lazy Approach Example - I

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



Quick-union: Lazy Approach Example - Connect $5, 9$



find has to follow links to the root

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	3	0	5	1	8	8

find(5) *is* id[id[id[5]]]

find(9) *is* id[id[9]]

union changes just one link

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	3	0	5	1	8	8
		1	8	1	8	3	0	5	1	8	8

Quick-union overview

Some Tree data structure definitions

- The **size** of a tree is its number of nodes.
- The **depth** of a node in a tree is the number of links on the path from it to the root.
- The **height** of a tree is the maximum depth among its nodes.

Quick-union: Lazy Approach II

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

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

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


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

set id of each object to itself
(N array accesses)

chase parent pointers until reach root
(depth of i array accesses)

change root of p to point to root of q
(depth of p and q array accesses)

Time complexity of Quick-Find + Union and Quick-union+Find

- Cost Model: Number of array accesses (for read and write).

algorithm	initialize	union	find	connected	
quick-find	N	N	1	1	
quick-union	N	N †	N	N	← worst case

† includes cost of finding roots

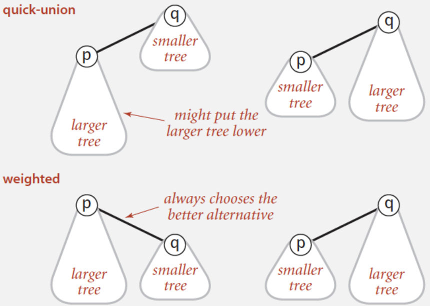
- Quick-find defect.
  - Union too expensive (N array accesses).
  - Trees are flat, but too expensive to keep them flat.
- Quick-union defect.
  - Trees can get tall.
  - Find too expensive (could be N array accesses).

Weighted Quick-union

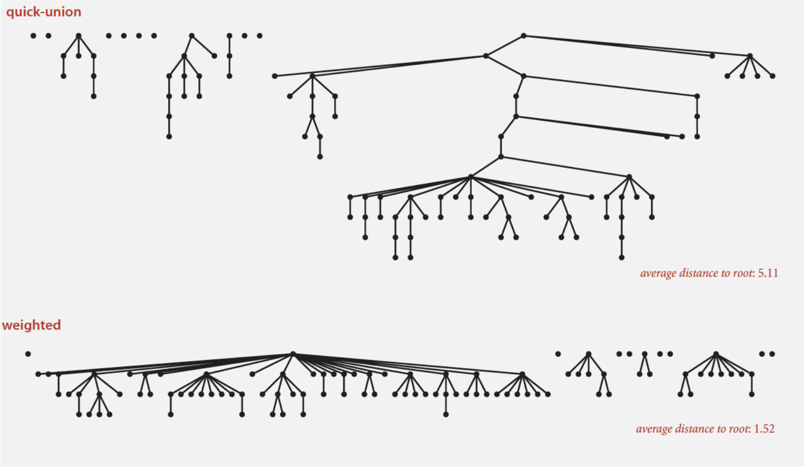
Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree.

reasonable alternatives:
union by height or "rank"



Weighted Quick-union



Quick-union and weighted quick-union (100 sites, 88 `union()` operations)

Weighted Quick-union: code

- Data structure: Same as quick-union, but maintain extra array $sz[i]$ to count number of objects in the tree rooted at i.
- Find/connected: Identical to quick-union.
- Union: Modify quick-union to:
  - Link root of smaller tree to root of larger tree.
  - Update the $sz[]$ array.

```
public void union(int p, int q)
{
   int i = find(p);
   int j = find(q);
   if (i == j) return;

   // Make smaller root point to larger one.
   if   (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
   else                 { id[j] = i; sz[i] += sz[j]; }
   count--;
}
```

Time Complexity of Weighted Quick-union+Find

- Below: $lgN = \log_2 N$.
- Proposition: Depth of any node x is at most lgN.
- Running time:
  - Find: takes time proportional to depth of p.
  - Union: takes constant time, given roots.

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N †	N	N
weighted QU	N	lg N †	lg N	lg N

† includes cost of finding roots