Union-Find

Week 6

COMP 1201 (Algorithmics)

Dr Andrew Sogokon a.sogokon@soton.ac.uk
ECS, University of Southampton

4 March 2020

Previously...

Data structures with simple entities:

- Linked lists, trees, heaps, hash tables,
- Operations: **`search`**, **`insert`**, **`remove`**,
- Goal: build data structures with fast (e.g. logarithmic) operation running time.

More complex data?

- Dynamic sets of entities.

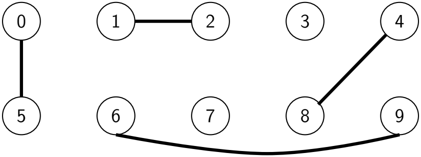
The Union-Find Problem

- **Example: dynamic connectivity**
- Equivalence classes and relations
- Quick-Find
- Quick-Union
- Improvements

Dynamic Connectivity

Given a set of N objects:

- **`union`** command: *connect two objects*
- **`isConnected`** query: *check whether two objects are connected via a path*



- **`isConnected(0,5)=True`**,
- **`isConnected(1,7)=False`**,
- Connectivity can change over time (hence *dynamic*).

The Union-Find Problem

- Example: dynamic connectivity
- **Equivalence classes and relations**
- Quick-Find
- Quick-Union
- Improvements

Equivalence classes and relations

Recall (from the Foundations course in Semester 1):

Given a set of N elements $S = \{S_1, S_2, \ldots, S_N\}$, **equivalence** is a binary relation $\sim$:

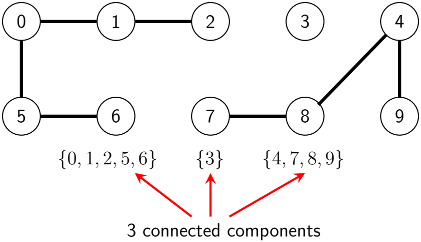
- Reflexive: $S_i \sim S_i$,
- Symmetric: $S_i \sim S_j \rightarrow S_j \sim S_i$,
- Transitive: $S_i \sim S_j,\ S_j \sim S_k \rightarrow S_i \sim S_k$

Equivalence classes and relations

Recall (from the Foundations course in Semester 1):

Given a set of N elements $S = \{S_1, S_2, \ldots, S_N\}$, **equivalence** is a binary relation $\sim$:

- Reflexive: $S_i \sim S_i$,
- Symmetric: $S_i \sim S_j \rightarrow S_j \sim S_i$,
- Transitive: $S_i \sim S_j,\ S_j \sim S_k \rightarrow S_i \sim S_k$



Applications of Equivalence Classes

- Pixels in a digital photograph
  (*equivalence class: pixels from the same object*)
- Computer networks
  (*equivalence class: machines from the same cluster*)
- Social networks
  (*equivalence class: people from the same group of friends*)
- Transistors in hardware
  (*equivalence classes: connected components*)

Historical note: the original paper on Union-Find is from 1964:
"An improved **equivalence algorithm**", by Galler & Fisher.

Union-Find (Disjoint Sets) Data Structure

Find operation: check which *class* a given element belongs to.
Union operation: merge two given equivalence classes into one.

Question: *how do we design a data structure for this type of problem?*

Union-Find (Disjoint Sets) data structure.

```
public class UF {

 public UF(int numElements) // Constructor

 boolean isConnected(int p, int q)

 public int find(int p) // Find class of p

 public void union(int p, int q) // Union
}
```

The Union-Find Problem

- Example: dynamic connectivity
- Equivalence classes and relations
- **Quick-Find**
- Quick-Union
- Improvements

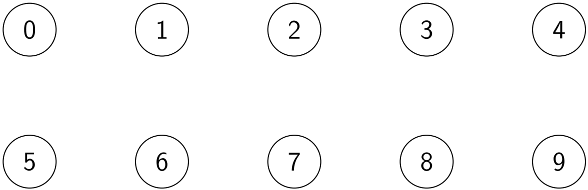
Quick-Find

- Integer array `id[]` of size N

	0	1	2	3	4	5	6	7	8	9	⟵ elements
`id[]`	0	1	2	3	4	5	6	7	8	9	⟵ classes

- Elements `p` and `q` are connected/equivalent if and only if their `id`s are the same.
- **`isConnected(p,q)`** : check whether `p` and `q` have the same `id`.
- **`find(p)`** : simply return `p`'s `id`.
- **`union(p,q)`** : merge components containing `p` and `q` by *changing the* `id` *of all those elements with* `id[p]` *to* `id[q]`.

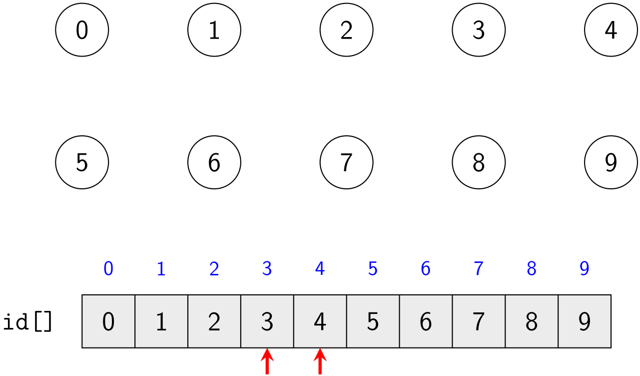
Quick-Find



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

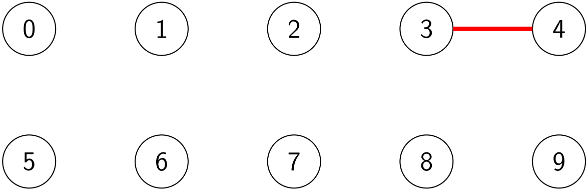
Quick-Find

`union(4,3)`



Quick-Find

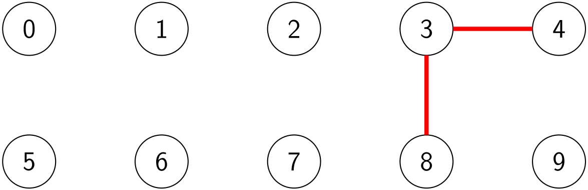
`union(4,3)`



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

Quick-Find

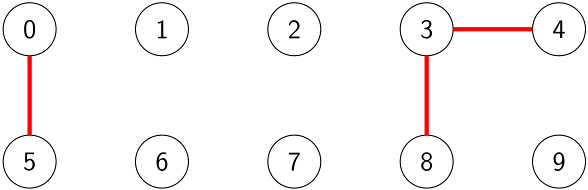
`union(3,8)`



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

Quick-Find

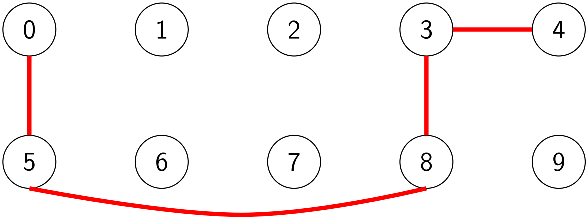
`union(0,5)`



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

Quick-Find

`union(5,8)`



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

Quick-Find implementation

```java
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 boolean isConnected(int p, int q){
     return id[p] == id[q];
 }

 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;
 }

 }
```

Time-complexity of Quick-Find

- Initialisation: $O(N)$
- **`find`**:

Time-complexity of Quick-Find

- Initialisation: $O(N)$
- **`find`**: $O(1)$

Time-complexity of Quick-Find

- Initialisation: $O(N)$
- **`find`**: $O(1)$
- **`union`**:

Time-complexity of Quick-Find

- Initialisation: $O(N)$
- **`find`**: $O(1)$
- **`union`**: $O(N)$

What if N is *very large* and **`union`** is called frequently?

We need a faster **`union`** !

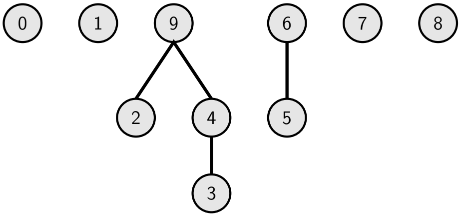
The Union-Find Problem

- Example: dynamic connectivity
- Equivalence classes and relations
- Quick-Find
- **Quick-Union**
- Improvements

Quick-Union

- Integer array `id[]` of size N.
- Components are stored as *trees* (the whole thing is a *forest*).
- The `id` of an element is its *parent* (if not root), or itself (if root); e.g. `id[3]=4` means “4 is a parent of 3”.

	0	1	2	3	4	5	6	7	8	9	← elements
`id[]`	0	1	9	4	9	6	6	7	8	9	← parents



Quick-Union

- **`find(p)`** : return the `id` of `p`'s *root*.
- **`isConnected(p,q)`** : true if and only if `p` and `q` have the same root.
- **`union(p,q)`** : add `p`'s root as a direct child of `q`'s root.

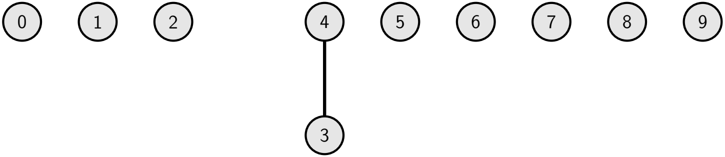
Quick-Union

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

Quick-Union

`union(3,4)`

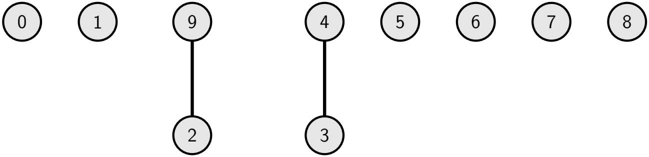
	0	1	2	3	4	5	6	7	8	9
`id[]`	0	1	2	4	4	5	6	7	8	9



Quick-Union

`union(2,9)`

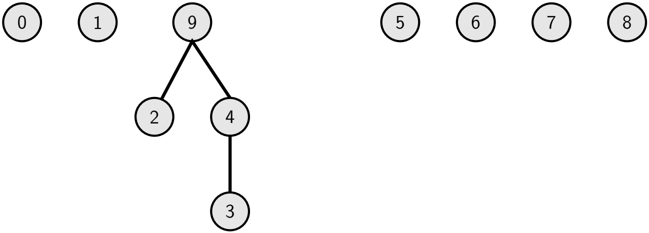
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	4	5	6	7	8	9



Quick-Union

`union(3,2)`

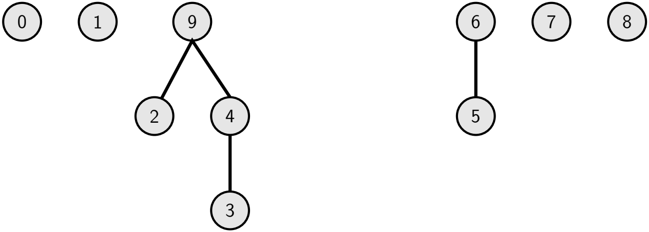
	0	1	2	3	4	5	6	7	8	9
`id[]`	0	1	9	4	9	5	6	7	8	9



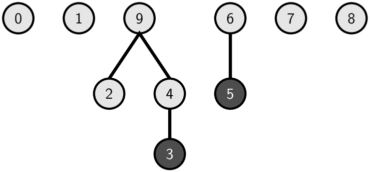
Quick-Union

`union(5,6)`

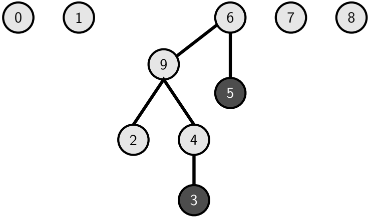
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9



Quick-Union



`union(3,5)`



Quick-Union implementation

```
public class QuickUnionUF {

 private int[] id;

 public QuickUnionUF(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 isConnected(int p, int q){
     return root(p) == root(q);
 }

 public int find(int p){
     return root(p);
 }

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

Complexity of Quick-Union

- Initialisation : $O(N)$
- `find` :

Complexity of Quick-Union

- Initialisation : $O(N)$
- `find` : $O(N)$

Complexity of Quick-Union

- Initialisation : $O(N)$
- **`find`** : $O(N)$
- **`union`** :

Complexity of Quick-Union

- Initialisation : $O(N)$
- **`find`** : $O(N)$
- **`union`** : $O(N)$ tree can be imbalanced

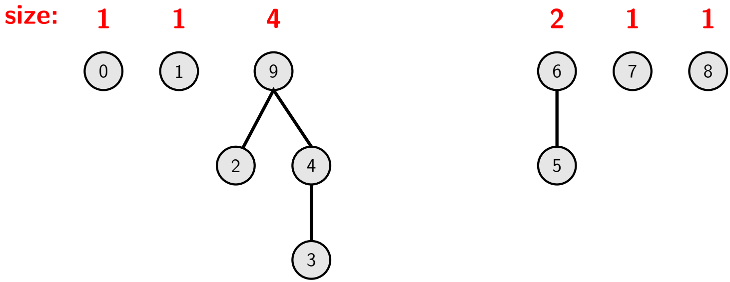
This is *even worse than Quick-Find!*

More sophisticated algorithms needed.

Improvement 1: **Weighted** Quick-Union

Main idea: Keep the tree *balanced* at union.

- Maintain a `size[]` array to keep track of the *number of items* in each tree.
- Merge the smaller tree into the larger one (i.e. link the root of the smaller tree directly to the root of the larger tree).



Improvement 1: **Weighted** Quick-Union

Implementation of **`find`** is exactly the same as in Quick-Union.

We need to modify **`union`** to :

- merge the smaller tree into the larger tree, and
- keep the `size[]` array updated.

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

Proposition 1

The depth of any tree in the forest formed by **weighted** *Quick-Union is* $\log_2(N)$.

Time complexity of **`find`** and **`union`** become $O(\log_2(N))$.

Improvement 2: Path compression

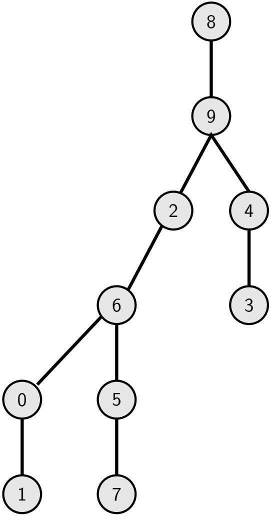
Observation: when we perform **`find(p)`** in Quick-Union, we actually look for a *path* from element `p` to its *root*.

Idea:

- Set the `id` of the items on this path to be the root each time a root is computed.

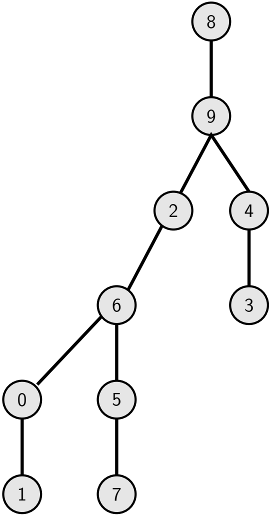
This should flatten the tree significantly.

Improvement 2: Path compression



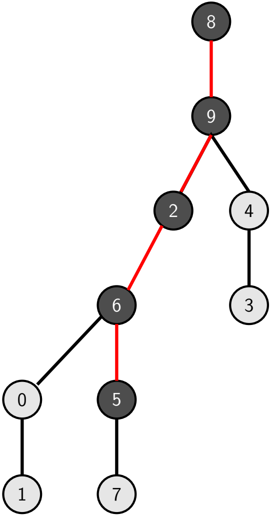
Improvement 2: Path compression

`find(5)`



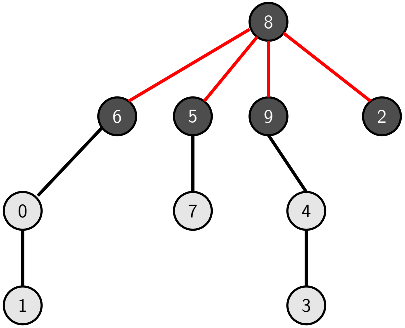
Improvement 2: Path compression

`find(5)`



Improvement 2: Path compression

`find(5)`



Weighted Quick-Union + Path Compression

- Initialisation: $O(N)$
- **`find`**: $O(\log^*(N))$ [Hopcroft & Ullman, 1973]
- **`union`**: $O(\log^*(N))$ [Hopcroft & Ullman, 1973]

$\log^*$ is *iterated logarithm*: *the number of times you need to apply the logarithm function before you get a number less than or equal to* 1.

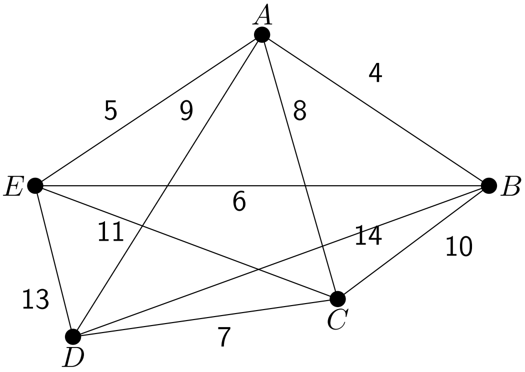
$$\log^*(N) = \begin{cases} 0 & \text{if } N \leq 1 \\ 1 + \log^*(\log(N)) & \text{if } N > 1 \end{cases}$$

$\log^*(2^{100}) = 5, \quad \log^*(2^{1,000}) = 5, \quad \log^*(2^{1,000,000,000}) = 6$

In practice, $\log^*(N) \leq 5$.

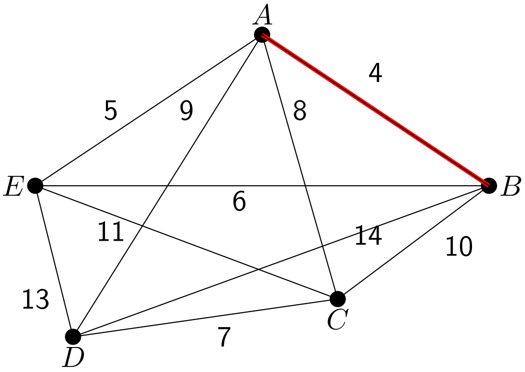
Important applications of Union-Find

Kruskal's Algorithm (1956) for finding **minimum spanning trees**.



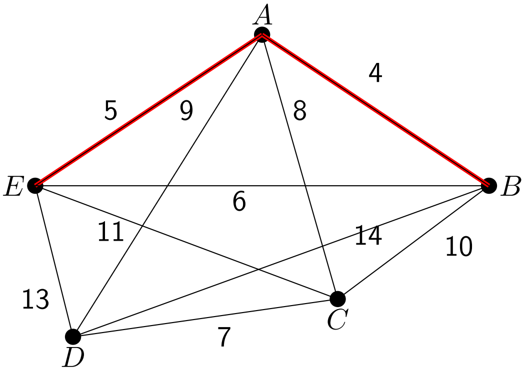
Important applications of Union-Find

Kruskal's Algorithm (1956) for finding **minimum spanning trees**.



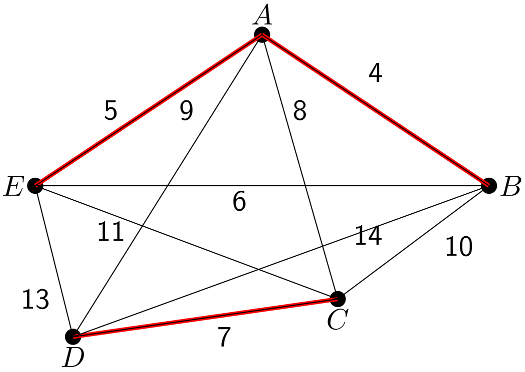
Important applications of Union-Find

Kruskal's Algorithm (1956) for finding **minimum spanning trees**.



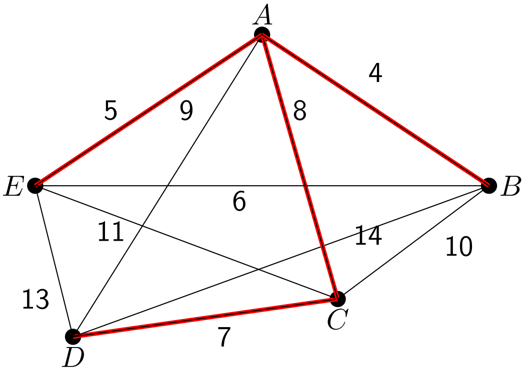
Important applications of Union-Find

Kruskal's Algorithm (1956) for finding **minimum spanning trees**.



Important applications of Union-Find

Kruskal's Algorithm (1956) for finding **minimum spanning trees**.



Important applications of Union-Find

Union-Find can also be used to efficiently detect **cycles in graphs** (tutorial problem).

Further Reading:

1. Chapter 1, §1.5 in Sedgewick and Wayne **Algorithms (4th Edition)**.
2. Java code, problems, and case studies with Union-Find, companion to the Sedgewick and Wayne textbook: https://algs4.cs.princeton.edu/15uf/
3. Chapter 21 in Cormen (CLRS) **Introduction to Algorithms**.

Acknowledgements: Partly based on earlier COMP 1201 slides by Dr Long Tran-Thanh, University of Southampton.