CE2101/ CZ2101: Algorithm Design and Analysis

Union-Find Programs and Kruskal's Algorithm

Ke Yiping, Kelly

Contents

allow change

- Dynamic Equivalence Relations
  - Three basic operations
- Union-Find Programs
  - Improve algorithms step by step
    - QuickFind
    - QuickUnion
    - Weighted QuickUnion with Path Compression (WQUPC)
- Kruskal's Algorithm
  - Pseudocode
  - Correctness
  - Complexity

Learning Objectives

At the end of this lecture, students should be able to:

- Understand the concept of dynamic equivalence relations
- Understand and analyze various union-find programs
- Solve minimum spanning tree (MST) problem using Kruskal's algorithm
- Prove the correctness of Kruskal's algorithm

Dynamic Equivalence Relations

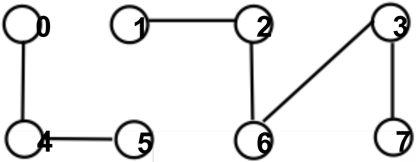
Dynamic Equivalence Relations

- An equivalence relation **R** on a set **S** is a binary relation, such that for every a, b, and c in **S**:
  - Reflexivity: R(a, a)
  - Symmetry: if R(a, b), then R(b, a)
  - Transitivity: if R(a, b) and R(b, c), then R(a, c)

- Dynamic equivalence relation: the equivalence relation will change with a number of operations
- Given a set of N objects in **S**, define three operations:
  - Initialize the objects
  - Connect two objects: add them into relation **R**
  - Is there a path connecting two objects?

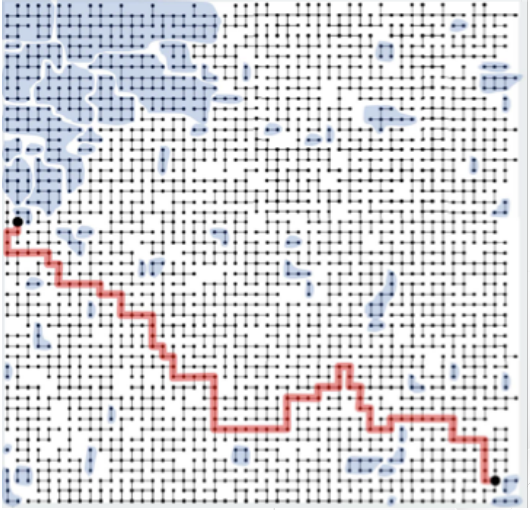
Running Example

- Initialize 8 objects: 0, …, 7
- Connect 4 and 5
- Connect 1 and 2
- Connect 6 and 3
- Connect 0 and 4
- Are 1 and 7 connected?
- Are 0 and 5 connected?
- Connect 2 and 6
- Connect 3 and 7
- Are 1 and 7 connected?



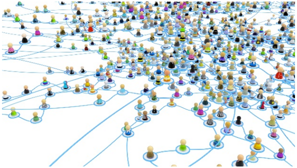
A Larger Example

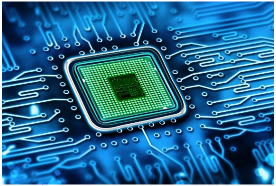
- Are the two big dots connected?
- Yes
- 63 components



Applications

- Friends in a social network
- Webpages on the Internet
- Statements in a Python program
- Computers in a computer network
- Transistors in a computer chip





Challenges

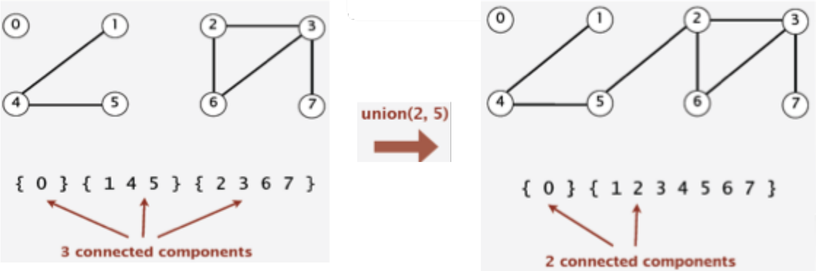
- Dynamics:
  - Operations may be intermixed.
- Number of operations M can be huge.
- Number of objects N can be huge.

- Goal: design efficient data structure

Union-Find Programs

Union-Find

- Equivalence class: connected components
- Three operations to implement the dynamic equivalence relation
  - find(p): which component does an object p belong to?
  - connected(p, q): are two objects p and q connected?
  - union(p, q): connect p and q (compute the union of their components)



Union-Find APIs

- A class defined for union-find (in Java)

Can have different algos

`public class UF`

`UF(int N)`	*initialize union-find data structure with N singleton objects (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)`	*are p and q in the same component?*

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

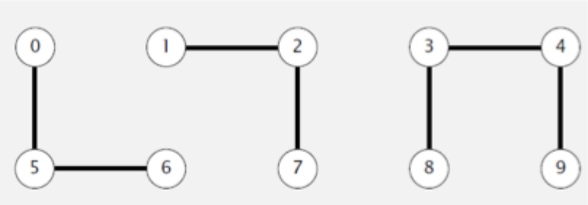
1-line implementation of connected()

QuickFind Algorithm

QuickFind [Eager Approach]

- Data structure:
  - Integer array id[ ] of length N
  - Interpretation: id[*p*] is the ID of the component that *p* belongs to

eager approach: immediate update



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

QuickFind Functions

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

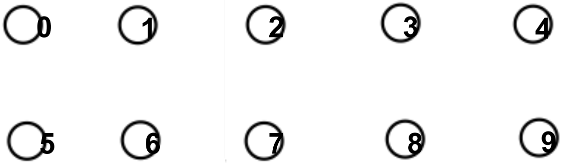
- find(*p*): what is the id of *p*?
  - id[6] = 0;   id[1] = 1
- connected(*p*, *q*): do *p* and *q* have the same id?
  - 6 and 1 are not connected
- union(*p*, *q*): change all entries whose id equals id[*p*] to id[*q*]
  - union(6, 1)

→ change id[6] to id[1]
content of

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

problem: many values can change

QuickFind Demo

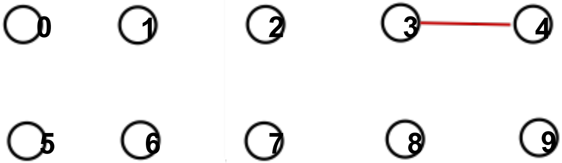


id									
0	1	2	3	4	5	6	7	8	9

Initially every component's index = position
(each a union)

QuickFind Demo

union(4, 3)



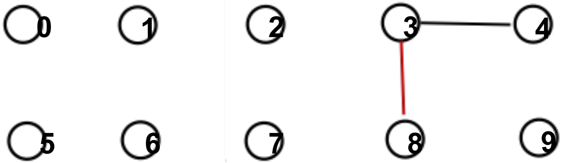
change content of id[4] to id[3]

id	0	1	2	3	3	5	6	7	8	9

QuickFind Demo

union(3, 8)

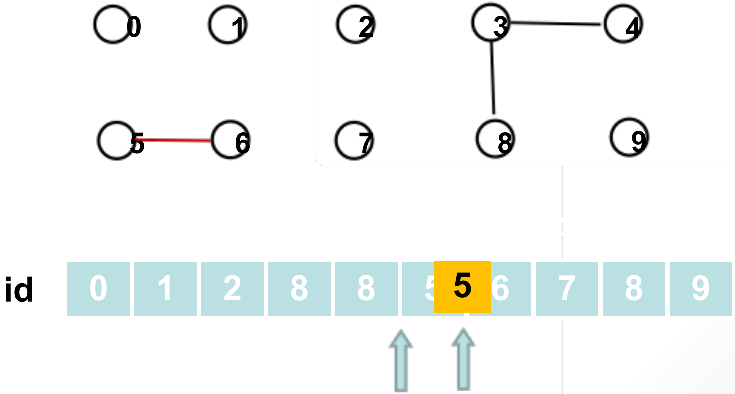
change Id[3], Id[4] to Id[8]



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

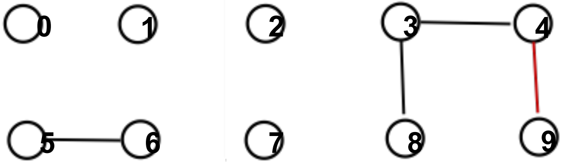
QuickFind Demo

union(6, 5)



QuickFind Demo

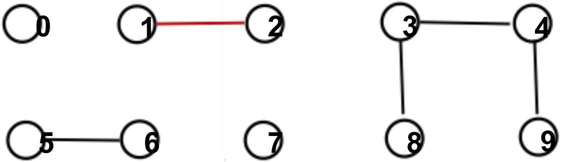
union(9, 4)



id	0	1	2	8	8	5	5	7	8	9

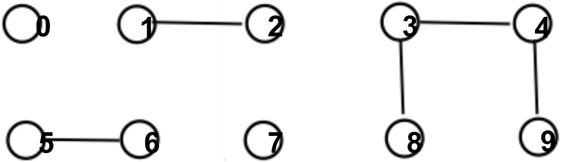
QuickFind Demo

union(2, 1)



id	0	1	1	8	8	5	5	7	8	8

connected(8, 9)

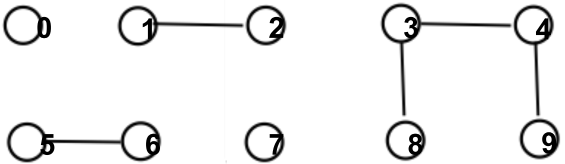


id									
0	1	1	8	8	5	5	7	8	8

Connected

QuickFind Demo

connected(5, 0)

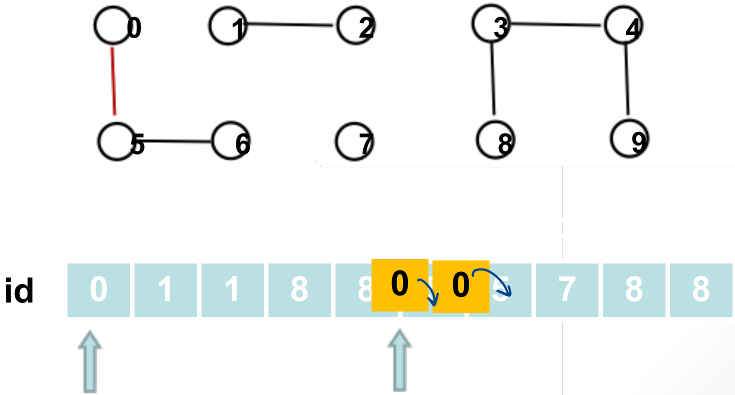


id	0	1	1	8	8	5	5	7	8	8

Not connected

QuickFind Demo

union(5, 0)



NANYANG TECHNOLOGICAL UNIVERSITY

QuickFind Implementation (Java)

```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 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;
   }
}
```

Initialization

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

return the id of p
(1 array access)

n+2.

Scan entire array.

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

QuickFind - Time Complexity

- Measured by: number of array accesses (read or write)

Algorithm	Initialization	union	find	connected
QuickFind	O(N)	O(N)	O(1)	O(1)

- union() is too expensive:
  - For N union operations on N objects, it takes O(N2) time.

NANYANG TECHNOLOGICAL UNIVERSITY

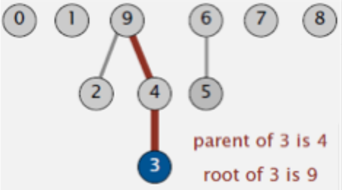
Quadratic is not scalable

- Rough standard (in 2020)
  - 1011 operations per second
  - 1011 words of main memory (400GB)
  - Access all words in ~1 second
- Huge problem for QuickFind
  - 1011 union operations on 1011 objects
  - QuickFind takes more than 1022 operations.
  - 3000+ years to complete!
- Quadratic algorithms don’t scale with technology
  - We need better algorithms!

QuickUnion Algorithm

QuickUnion [Lazy Approach]

- Data structure:
  - Integer array id[ ] of length N
  - Interpretation: id[*p*] is the parent of *p*
  - Root of p is id[id[id[…id[i]…]]]: component ID
    - Keep id-ing until the value doesn't change (cycles?)

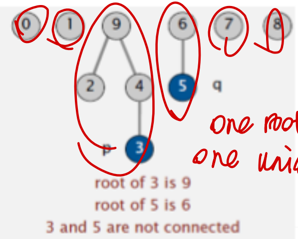


	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9

QuickUnion Functions

- find(*p*): what is the root of *p*?
- connected(*p*, *q*): do *p* and *q* have the same root?
- union(*p*, *q*): set the id of *p*'s root to the id of *q*'s root.

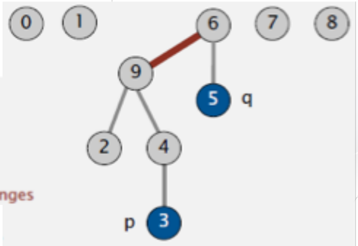
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9



one root, one union

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

only one value changes



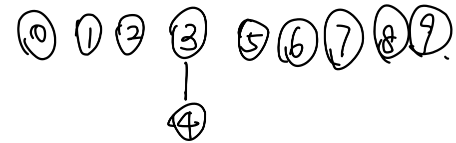
Initial id

0	1	2	3	4	5	6	7	8	9

Union (4, 3)

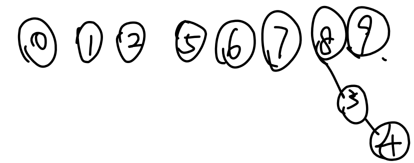
root of 4 to 3

0	1	2	3	3	5	6	7	8	9



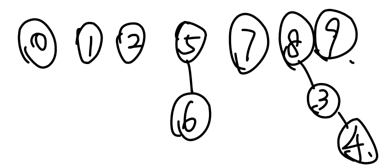
Union (3, 8)

0	1	2	8	3	5	6	7	8	9



Union (6, 5)

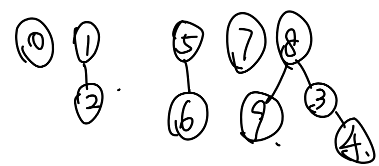
0	1	2	8	3	5	5	7	8	9



Union (9, 4)
Union (2, 1)

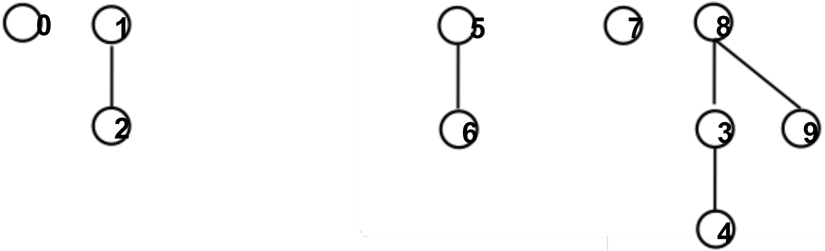
0	1	1	8	3	5	5	7	8	8

find root of 4.



QuickUnion Demo

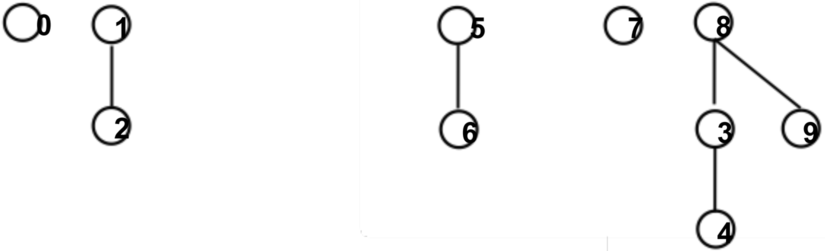
connected(8, 9) Same root ✓ connected



id										
id	0	1	1	8	3	5	5	7	8	8

QuickUnion Demo

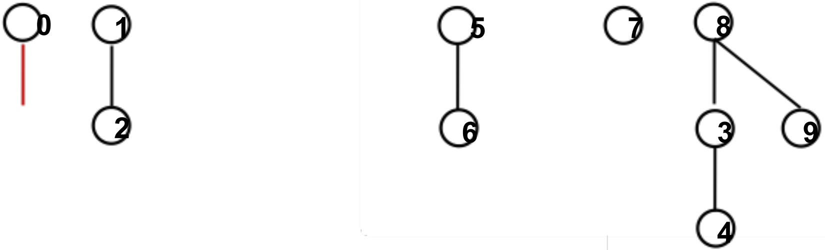
connected(5, 4)



id										
	0	1	1	8	3	5	5	7	8	8

QuickUnion Demo

union(5, 0)



id	0	1	1	8	3	0	5	7	8	8

QuickUnion Implementation (Java)

```
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)

NANYANG TECHNOLOGICAL UNIVERSITY

QuickUnion – Time Complexity

- Measured by: number of array accesses (read or write)

Algorithm	Initialization	union	find	connected
QuickFind	O(N)	O(N)	O(1)	O(1)
QuickUnion	O(N)	O(N)	O(N)	O(N)

worst case

- QuickFind:
  - union() is too expensive.
  - Trees are flat, but need many id updates to keep them flat
- QuickUnion:
  - Trees can be tall.
  - find() and connected() are too expensive.

Weighted Quick Union

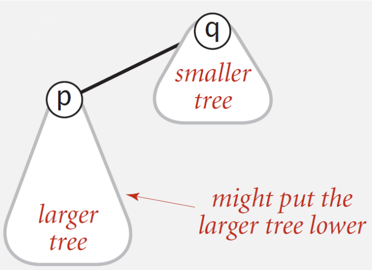
WQUPC Algorithm

NANYANG TECHNOLOGICAL UNIVERSITY

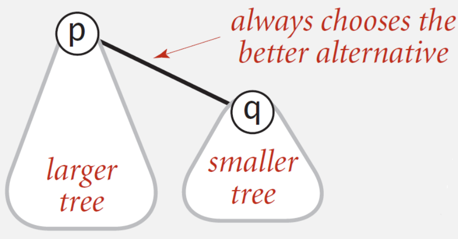
1. Weighted QuickUnion

- Idea:
  - Modify QuickUnion 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.
  - Other alternatives: ?

QuickUnion



Weighted QuickUnion

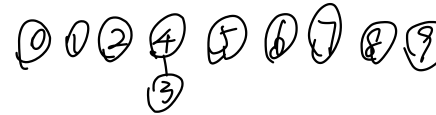


Initial id

0	1	2	3	4	5	6	7	8	9

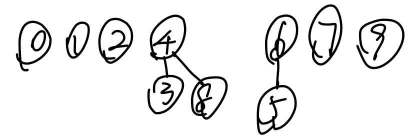
make 3 a child of 4
Union(4, 3)

0	1	2	4	4	5	6	7	8	9



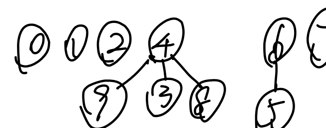
Union(3, 8)
size of 3 union: 2 > 1
Union(6, 5)

0	1	2	4	4	6	6	7	4	9



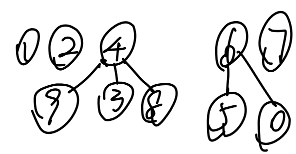
make 9 the lower
Union(9, 4)

0	1	2	4	4	6	6	7	4	4



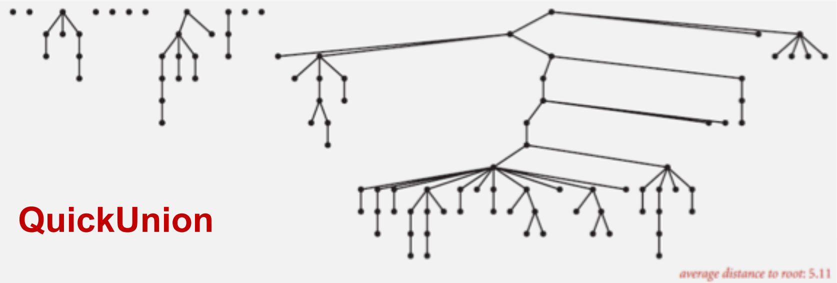
make 0 the lower tree
Union(0, 5).

0	1	2	4	4	6	6	7	4	4



QuickUnion vs Weighted QuickUnion

On 100 objects, 88 union operations



QuickUnion

Weighted QuickUnion

Weighted QuickUnion Implementation

- Data structure:
  - Extra array sz[i] to count no. of objects in the tree rooted at i.
- find() and connect()
  - Same as QuickUnion
- union()
  - Make smaller tree lower
  - Update the array sz[ ]

```
int i = find(p);
int j = find(q);
if (i == j) return;
if  (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```

NANYANG TECHNOLOGICAL UNIVERSITY

Weighted QuickUnion - Time Complexity

- Running time depends on the depth of a node.
- **Proposition.** Depth of any node x is at most log2*N*.
- Proof:
  - (only union operation does)
  - What causes the depth of a node x to increase?
  - Increases by 1 when tree T1 containing x is merged into another tree T2
    - smaller tree
    - $|T| \geq 2|T_1|$
    - Size of the tree containing x at least doubles since |T2| ≥ |T1|
    - Size of the tree containing x can double at most log2*N* times. (?)

NANYANG TECHNOLOGICAL UNIVERSITY

Weighted QuickUnion - Time Complexity

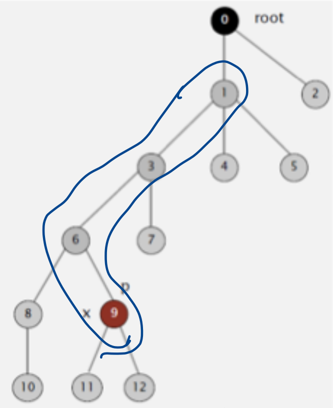
- Running time depends on the depth of a node.
- **Proposition.** Depth of any node x is at most log2*N*.

Algorithm	Initialization	union	find	connected
QuickFind	O(N)	O(N)	O(1)	O(1)
QuickUnion	O(N)	O(N)	O(N)	O(N)
Weighted QuickUnion	O(N)	O(log N)	O(log N)	O(log N)

- Stop here??
  - No, easy to improve further.

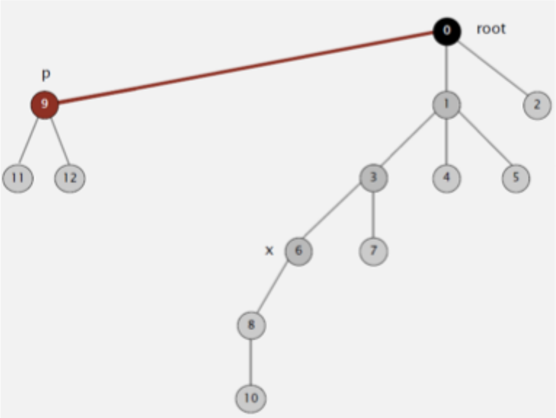
2. Weighted QuickUnion with Path Compression

- Idea:
  - Right after computing the root of *p*, set the id[ ] of each node on the path to that root



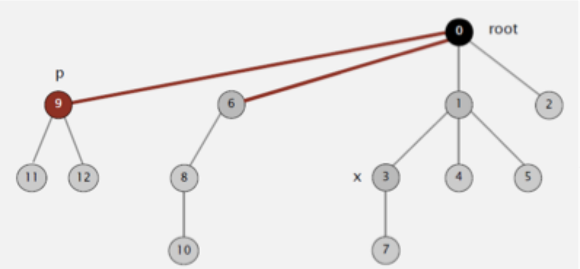
2. Weighted QuickUnion with Path Compression

- Idea:
  - Right after computing the root of *p*, set the id[ ] of each node on the path to that root



2. Weighted QuickUnion with Path Compression

- Idea:
  - Right after computing the root of *p*, set the id[ ] of each node on the path to that root



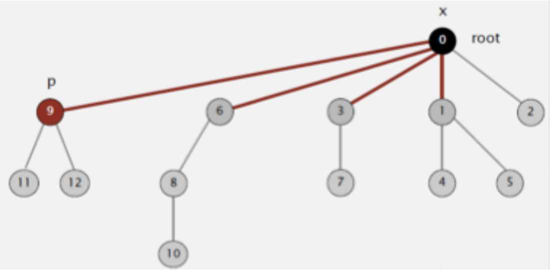
2. Weighted QuickUnion with Path Compression

- Idea:
  - Right after computing the root of *p*, set the id[ ] of each node on the path to that root

NANYANG TECHNOLOGICAL UNIVERSITY

2. Weighted QuickUnion with Path Compression

- Idea:
  - Right after computing the root of *p*, set the id[ ] of each node on the path to that root



WQUPC Implementation (Java)

- Two-pass implementation
  - Add second loop to find() to set the id[ ] of each examined node to the root. (As in previous example)
- Simpler one-pass variant (path halving)
  - Make every other node in path point to its grandparent

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

Only one extra line of code !

Jump to grandparent

- In practice: keeps tree almost completely flat

WQUPC: Time Complexity

iterative logarithm function

Proposition. [Hopcroft-Ulman, Tarjan] Starting from an empty data structure, any sequence of M union–find ops on N objects makes $\le c\,(N + M \lg^* N)$ array accesses.

- Analysis can be improved to $N + M\,\alpha(M, N)$.
- Simple algorithm with fascinating mathematics.

# of N to make lg N to be 0

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

iterated lg function

$\lg(\lg(\lg(\lg(\lg 2^{65536})))) = 5$

- [Fredman-Saks] No linear-time algorithm exists.
- WQUPC:
  - In theory: not quite linear
  - In practice: linear

Summary: Union-Find

- Weighted QuickUnion (w/wo path compression) makes it possible to solve problems that could not otherwise be addressed.
- Complexity for M union-find operations on N objects

Algorithms	Worst-Case Time
QuickFind	M N
QuickUnion	M N
Weighted QuickUnion	N + M logN
Weighted QuickUnion with Path Compression	N + M log*N

- [1011 unions with 1011 objects]
  - WQUPC reduces time from 3000 years to 6 seconds
  - Supercomputer won’t help much; good algorithm makes it feasible!

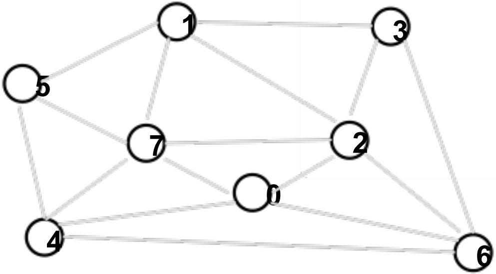
Kruskal’s Algorithm

NANYANG TECHNOLOGICAL UNIVERSITY

Kruskal's Algorithm

- To compute the minimum spanning tree (MST)
- Uses the **greedy** strategy
- Main Idea:
  - Consider edges in increasing order of weight
  - Add next edge to tree *T* unless it would create a cycle

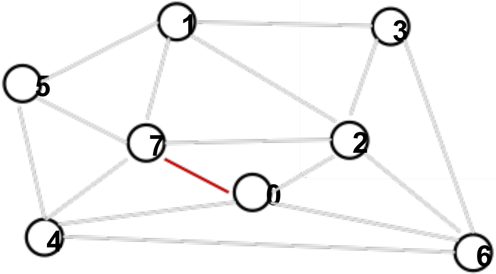
Kruskal's Algorithm – Example



Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Kruskal's Algorithm - Example

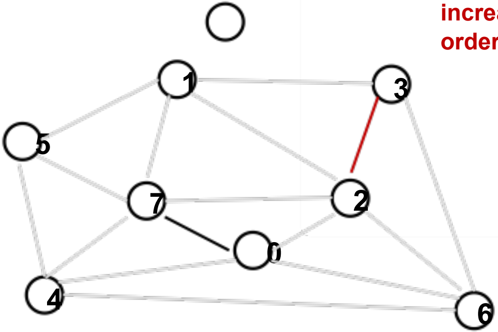


Graph edges in weight increasing order

0-7	**0.16**
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

NANYANG TECHNOLOGICAL UNIVERSITY

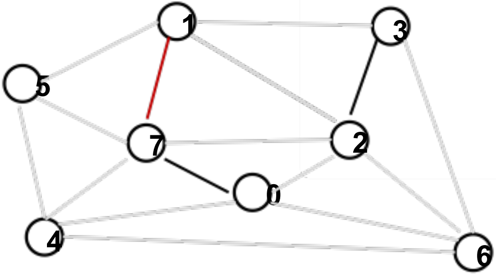
Kruskal's Algorithm - Example



Graph edges in weight increasing order

0-7	0.16
2-3	**0.17**
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

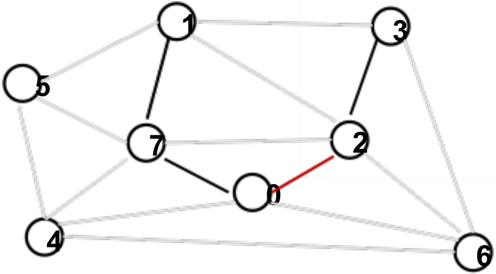
Kruskal's Algorithm - Example



Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	**0.19**
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

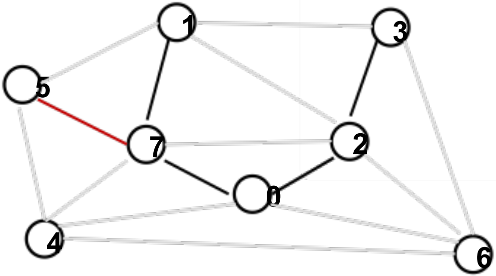
Kruskal's Algorithm - Example



Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	**0.26**
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

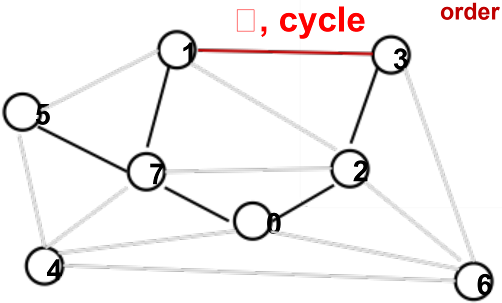
Kruskal's Algorithm - Example



Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	**0.28**
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

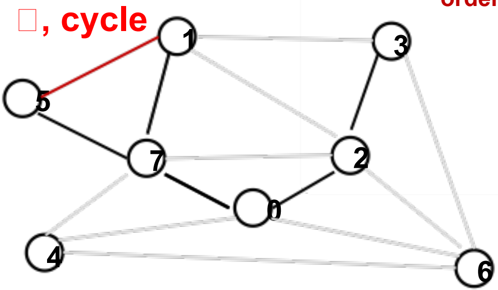
Kruskal's Algorithm - Example



Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	**0.29**
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

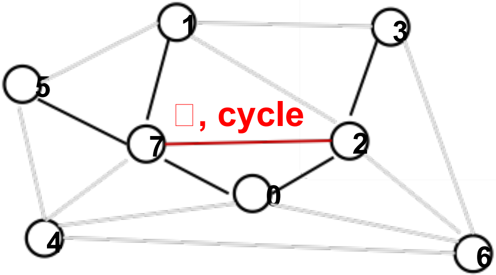
Kruskal's Algorithm - Example



Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	**0.32**
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Kruskal's Algorithm - Example

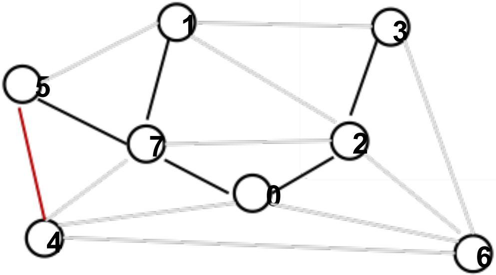


Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	**0.34**
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

NANYANG TECHNOLOGICAL UNIVERSITY

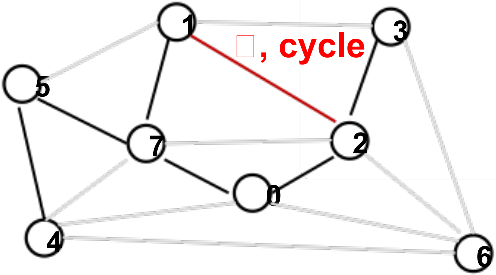
Kruskal's Algorithm - Example



Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	**0.35**
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Kruskal's Algorithm - Example

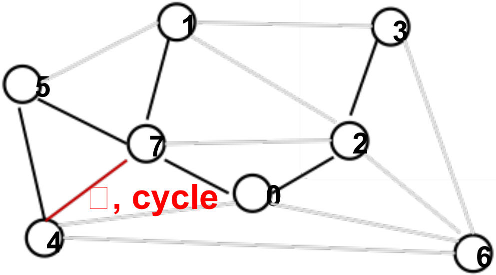


Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	**0.36**
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

NANYANG TECHNOLOGICAL UNIVERSITY

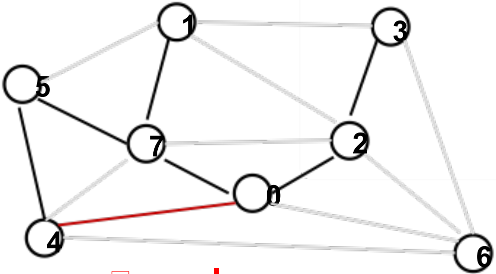
Kruskal's Algorithm - Example



Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	**0.37**
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Kruskal's Algorithm - Example

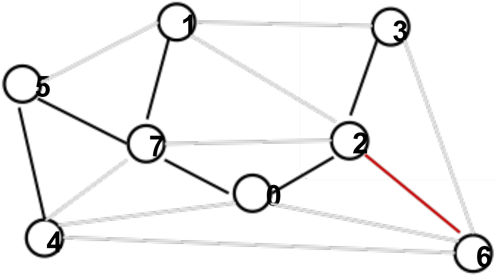


□, cycle

Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	**0.38**
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Kruskal's Algorithm - Example



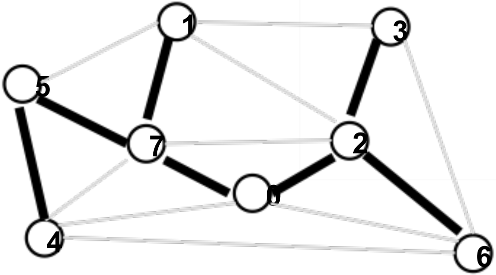
Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	**0.40**
3-6	0.52
6-0	0.58
6-4	0.93

Kruskal's Algorithm – Example

Graph edges in weight increasing order

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	**0.40**
3-6	0.52
6-0	0.58
6-4	0.93

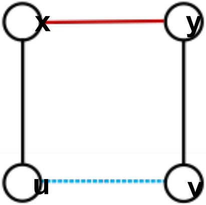


N-1 edges Done!

NANYANG TECHNOLOGICAL UNIVERSITY

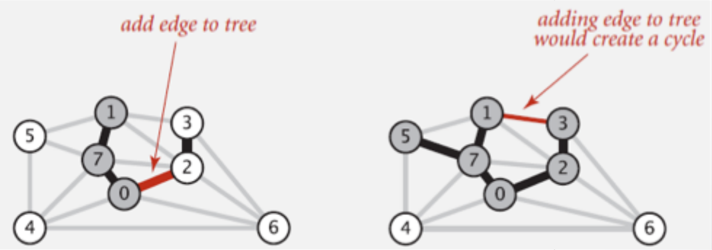
Kruskal's Algorithm: Correctness

- **Proposition.** [Kruskal 1956] Kruskal's algorithm correctly computes the MST.
- Proof by contradiction:
  - Suppose the tree *T* produced by Kruskal's is not an MST, i.e., it doesn't satisfy the MST property.
  - There is some edge *u-v* not in *T* such that adding *u-v* creates a cycle, in which some other edge *x-y* has weight W(*x-y*) > W(*u-v*).
  - As W(*x-y*) > W(*u-v*), edge *x-y* must be processed after edge *u-v* in Kruskal's.
  - At the time when *u-v* is processed, it must be added into *T* because it doesn't form a cycle.
  - This contradicts that *u-v* is not in *T*.



Kruskal's Algorithm: Implementation

- Challenge:
  - How to check if adding an edge *v*-*w* to tree *T* will create a cycle?
- Possible solutions:
  - DFS: run DFS from *v* to check if it could reach *w*. **O(|V|)**
  - Union-find: connected(*v*, *w*) = ? **O(log* |V|)**



Kruskal's Algorithm: Implementation

- Use the union-find data structure:
  - Use the id array to keep track of connected components in *T*.
  - If connected(*v*, *w*) = TRUE, then adding *v*-*w* will create a cycle.
  - Else, add *v*-*w* to *T* by calling union(*v*, *w*).

Case 1: adding v–w creates a cycle Case 2: add v–w to T and merge sets containing v and w

NANYANG TECHNOLOGICAL UNIVERSITY

Kruskal's Algorithm: Implementation (Java)

```
public class KruskalMST
{
   private Queue<Edge> mst = new Queue<Edge>();

   public KruskalMST(EdgeWeightedGraph G)
   {
      MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());        O(|E|)

      UF uf = new UF(G.V());                               O(|V|)
      while (!pq.isEmpty() && mst.size() < G.V()-1)
      {
         Edge e = pq.delMin();                       O(|E| log|E|)
         int v = e.either(), w = e.other(v);
         if (!uf.connected(v, w))                    O(|E| log*|V|)
         {
            uf.union(v, w);                          O(|V| log*|V|)
            mst.enqueue(e);                          O(|V|)
         }
      }
   }

   public Iterable<Edge> edges()
   {  return mst;  }
}
```

- build priority queue (or sort)
- greedily add edges to MST
- edge v–w does not create cycle
- merge sets
- add edge to MST

Overall: O(|E| log|E|)

NANYANG TECHNOLOGICAL UNIVERSITY

Kruskal's Algorithm: Summary

- Kruskal's algorithm finds the minimum spanning trees in weighted graphs
- It is a greedy algorithm.
- It uses the union-find data structure for efficient implementation.
- Its time complexity is O(|E| log|E|).