



AUBURN
UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Disjoint Sets

Applications

Kruskal's algorithm

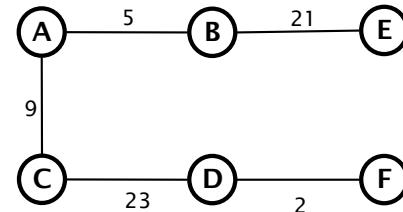
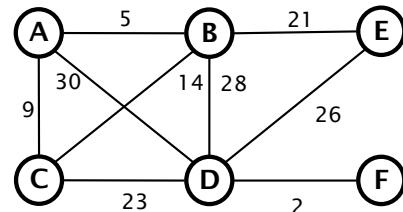
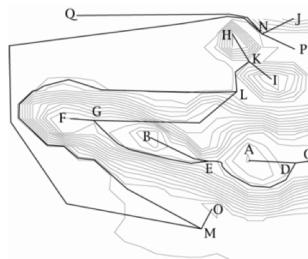
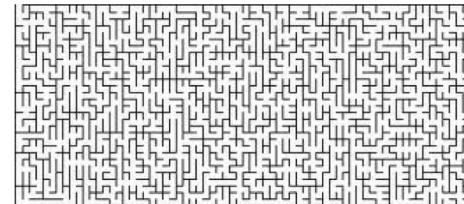


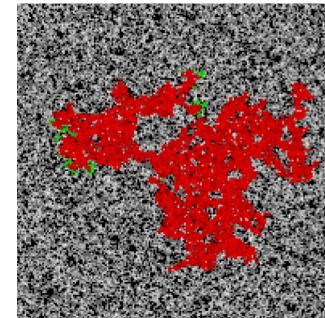
Image processing



Maze layout,
generation



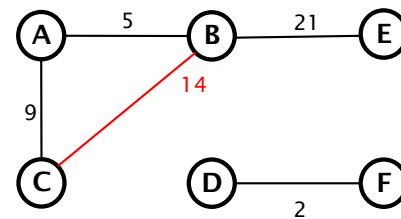
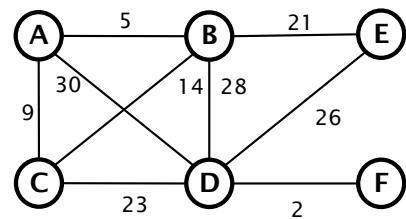
Percolation system modeling



water, oil, etc.
through ground
disease through
populations

Context: Kruskal's MST algorithm

```
Initialize MST with all vertices but no edges  
Add all the edges to a collection E  
while (still more edges in E) && (have not added n-1 edges to the MST) {  
    Remove the edge with minimum cost from E.  
    Add it to the MST if it does not create a cycle.  
}
```



Maintain a set of the connected components in the MST.

```
if (find(u) != find(v)) {  
    // add edge (u,v) to MST  
    union(u, v);  
}
```

```
if (!connected(u, v)) {  
    // add edge (u,v) to MST  
    union(u, v);  
}
```

{A, B, C} {D, F} {E}

{A, B, C, E} {D, F}

Disjoint Sets

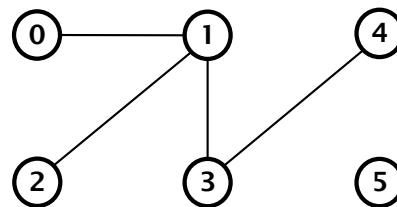
A **disjoint set** is a collection that contains a set of elements that are partitioned into disjoint (non-overlapping) subsets.

```
public interface DisjointSet {  
  
    /**  
     * combine components containing p and q  
     */  
    void union(int p, int q);  
  
    /**  
     * return component id for p  
     */  
    int find(int p);  
  
    /**  
     * are p and q in the same component?  
     */  
    boolean connected(int p, int q);  
  
    /**  
     * return number of connected components  
     */  
    int count();  
}
```

Typical application problems involve N elements that begin as individual disjoint sets, and the problem solution involves a sequence of union and find operations.

Disjoint set operations

count()	6
connected(3, 5)	false
connected(0, 4)	false
union(0, 1)	
union(3, 4)	
count()	4
connected(0, 4)	false
union(1, 3)	
connected(0, 4)	true
union(2, 1)	
count()	2



{0} {1} {2} {3} {4} {5}

{0, 1} {2} {3} {4} {5}

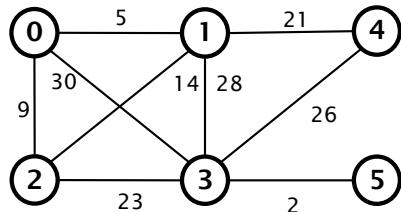
{0, 1} {2} {3, 4} {5}

{0, 1, 3, 4} {2} {5}

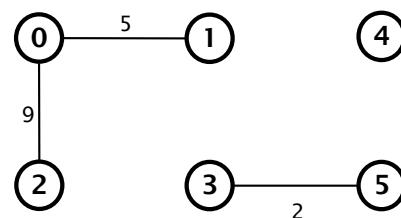
{0, 1, 2, 3, 4} {5}

Kruskal's algorithm with disjoint sets

Original graph



Minimum Spanning Tree



Disjoint Set of MST components

{0} {1} {2} {3} {4} {5}

connected(3, 5) false, ok to add

add (3, 5) to MST

union(3, 5)

{0} {1} {2} {3, 5} {4}

connected(0, 1) false, ok to add

add (0, 1) to MST

union(0, 1)

{0, 1} {2} {3, 5} {4}

connected(0, 2) false, ok to add

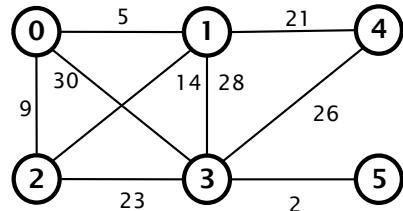
add (0, 2) to MST

union(0, 2)

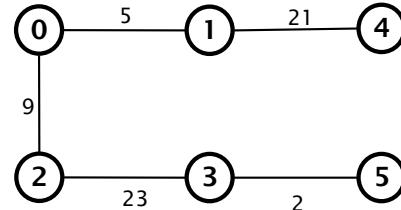
{0, 1, 2} {3, 5} {4}

Kruskal's algorithm with disjoint sets

Original graph



Minimum Spanning Tree



Disjoint Set of MST components

{0, 1, 2} {3, 5} {4}

connected(1, 2) *true, adding would create cycle*

connected(1, 4) *false, ok to add*

add (1, 4) to MST

union(1, 4)

{0, 1, 2, 4} {3, 5}

connected(2, 3) *false, ok to add*

add (2, 3) to MST

union(2, 3)

{0, 1, 2, 3, 4, 5}

Disjoint Set collection

A **disjoint set** is a collection that contains a set of elements that are partitioned into disjoint (non-overlapping) subsets.

```
public interface DisjointSet {  
  
    /**  
     * combine components containing p and q  
     */  
    void union(int p, int q);  
  
    /**  
     * return component id for p  
     */  
    int find(int p);  
  
    /**  
     * are p and q in the same component?  
     */  
    boolean connected(int p, int q);  
  
    /**  
     * return number of connected components  
     */  
    int count();  
}
```

Typical application problems involve N elements that begin as individual disjoint sets, and the problem solution involves a sequence of union and find operations.

Implementation strategies must consider the cost of a sequence of operations, not just the individual operations themselves.

Disjoint Set collection

A **disjoint set** is a collection that contains a set of elements that are partitioned into disjoint (non-overlapping) subsets.

```
public class ArrayDisjointSet implements DisjointSet {  
  
    private int[] id; // component ids  
    private int count; // number of components  
  
    public ArrayDisjointSet(int N) { . . . }  
  
    public void union(int p, int q);  
  
    public int find(int p);  
  
    public boolean connected(int p, int q);  
  
    public int count();  
}
```

Typical application problems involve N elements that begin as individual disjoint sets, and the problem solution involves a sequence of union and find operations.

Implementation strategies must consider the cost of a sequence of operations, not just the individual operations themselves.

Fast find

Fast find

Fast find strategy: Let each component be identified by the label of one vertex in that component. Store these “component ids” in an int array such that $\text{id}[i] == \text{component id of element } i$.

Initially:

{0}	{1}	{2}	{3}	{4}	{5}
0	1	2	3	4	5

$\text{find}(0) = 0$ $\text{find}(2) = 2$ $\text{find}(3) = 3$

After a sequence of unions:

{0, 1, 2}	{3, 5}	{4}
2	2	2

$\text{find}(0) = 2$ $\text{find}(2) = 2$ $\text{find}(3) = 5$

Fast find

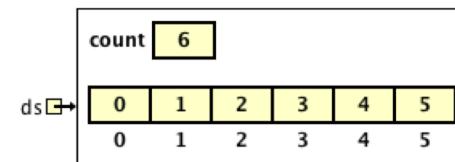
Fast find strategy: Let each component be identified by the label of one vertex in that component. Store these “component ids” in an int array such that $\text{id}[i] == \text{component id of element } i$.

Initially:

{0}	{1}	{2}	{3}	{4}	{5}
0	1	2	3	4	5

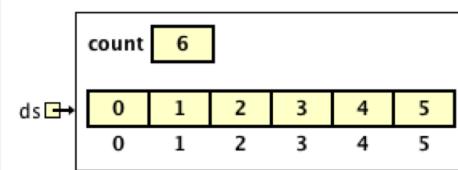
```
public class FastFindDS implements DisjointSet {  
  
    private int[] id; // component ids  
    private int count; // number of components  
  
    public FastFindDS(int N) {  
        count = N;  
        id = new int[N];  
        for (int i = 0; i < N; i++)  
            id[i] = i;  
    }  
}
```

```
FastFindDS ds;  
  
ds = new FastFindDS(6);
```



Fast find

```
public class FastFindDS implements DisjointSet {  
  
    private int[] id; // component ids  
    private int count; // number of components  
  
    /**  
     * return component for p  
     */  
    public int find(int p) {  
        return id[p];  
    }  
  
    /**  
     * are p and q in the same component?  
     */  
    public boolean connected(int p, int q) {  
        return find(p) == find(q);  
    }  
    /**  
     * return number of connected components  
     */  
    public int count() {  
        return count;  
    }  
}
```



OR: return id[p] == id[q];

Fast find

Union: The call `union(x, y)` combines the *component* that `x` belongs to with the *component* that `y` belongs to.

	{0} {1} {2} {3} {4} {5}	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	0	1	2	3	4	5	0	1	2	3	4	5
0	1	2	3	4	5									
0	1	2	3	4	5									
<code>union(3, 5)</code>	{0} {1} {2} {3, 5} {4}	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>5</td><td>4</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	0	1	2	5	4	5	0	1	2	3	4	5
0	1	2	5	4	5									
0	1	2	3	4	5									
<code>union(0, 2)</code>	{0, 2} {1} {3, 5} {4}	<table border="1"><tr><td>2</td><td>1</td><td>2</td><td>5</td><td>4</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2	1	2	5	4	5	0	1	2	3	4	5
2	1	2	5	4	5									
0	1	2	3	4	5									
<code>union(1, 0)</code>	{0, 1, 2} {3, 5} {4}	<table border="1"><tr><td>2</td><td>2</td><td>2</td><td>5</td><td>4</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2	2	2	5	4	5	0	1	2	3	4	5
2	2	2	5	4	5									
0	1	2	3	4	5									

Fast find

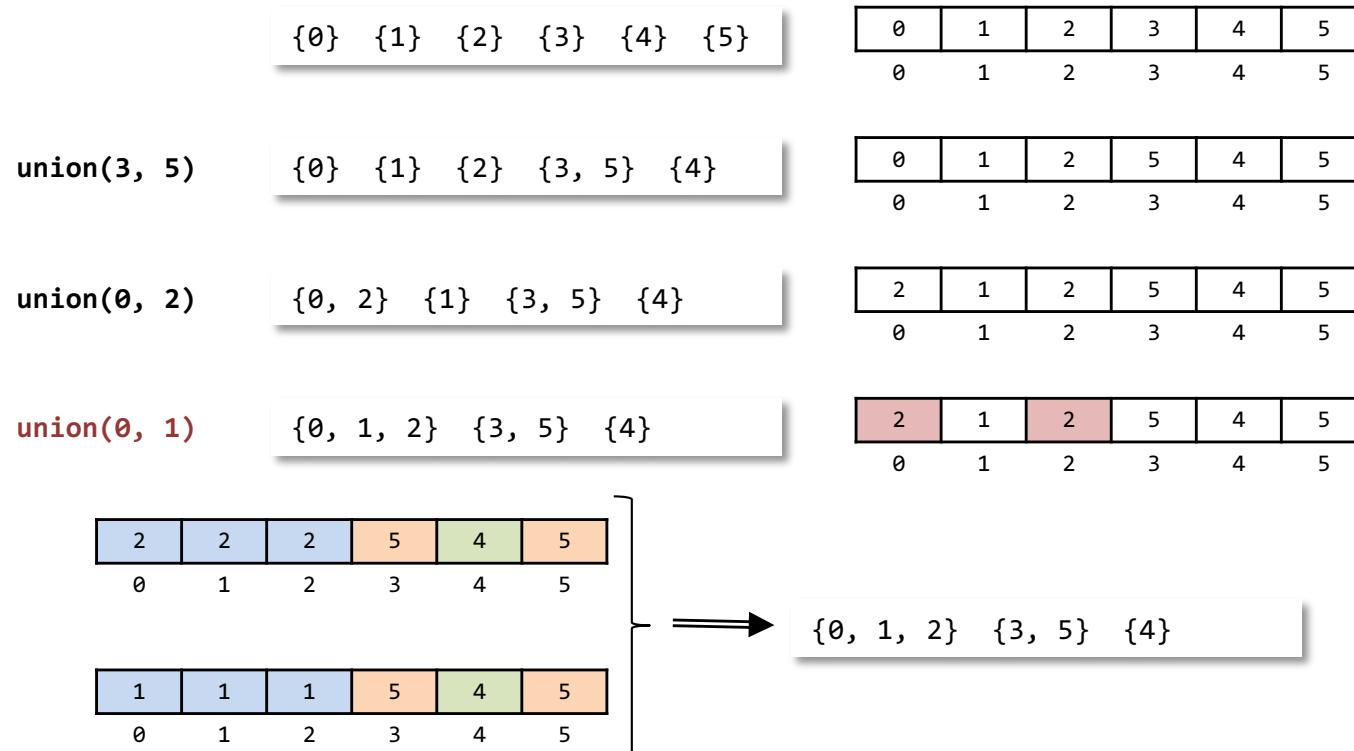
Union: The call `union(x, y)` combines the *component* that `x` belongs to with the *component* that `y` belongs to.

	{0} {1} {2} {3} {4} {5}	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	0	1	2	3	4	5	0	1	2	3	4	5
0	1	2	3	4	5									
0	1	2	3	4	5									
<code>union(3, 5)</code>	{0} {1} {2} {3, 5} {4}	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>5</td><td>4</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	0	1	2	5	4	5	0	1	2	3	4	5
0	1	2	5	4	5									
0	1	2	3	4	5									
<code>union(0, 2)</code>	{0, 2} {1} {3, 5} {4}	<table border="1"><tr><td>2</td><td>1</td><td>2</td><td>5</td><td>4</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2	1	2	5	4	5	0	1	2	3	4	5
2	1	2	5	4	5									
0	1	2	3	4	5									
<code>union(1, 0)</code>	{0, 1, 2} {3, 5} {4}	<table border="1"><tr><td>2</td><td>2</td><td>2</td><td>5</td><td>4</td><td>5</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2	2	2	5	4	5	0	1	2	3	4	5
2	2	2	5	4	5									
0	1	2	3	4	5									

Parameter order can't matter, so union isn't as efficient as the last example might have implied.

Fast find

Union: The call `union(x, y)` combines the *component* that `x` belongs to with the *component* that `y` belongs to.



Fast find

```
public class FastFindDS implements DisjointSet {  
  
    public void union(int p, int q) {  
        int pid = find(p);  
        int qid = find(q);  
        if (pid == qid) return; } }  
  
    for (int i = 0; i < id.length; i++) {  
        if (id[i] == pid) id[i] = qid; } }  
    count--;  
}
```

Could be shortened with connected(p, q).
Expressed this way for consistency among
different implementations.

Iterates over the entire array and
changes every pid to qid.

```
FastFindDS ds = new FastFindDS(6);  
  
ds.union(0, 1); id[0] == 0, id[1] == 1  
  
ds.union(3, 4); id[3] == 3, id[4] == 4  
  
ds.union(1, 3); id[1] == 1, id[3] == 4  
  
ds.union(2, 1); id[2] == 2, id[1] == 4
```

0	1	2	3	4	5
0	1	2	3	4	5

0	1	2	3	4	5
0	1	2	3	4	5

1	1	2	3	4	5
0	1	2	3	4	5

1	1	2	4	4	5
0	1	2	3	4	5

4	4	2	4	4	5
0	1	2	3	4	5

1	1	2	3	4	5
0	1	2	3	4	5

1	1	2	4	4	5
0	1	2	3	4	5

4	4	2	4	4	5
0	1	2	3	4	5

Fast find

Before

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

union(5, 6)

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

union(2, 9)

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

union(4, 9)

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

union(3, 4)

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

union(9, 6)

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

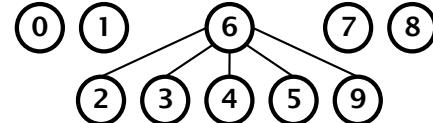
After

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

Final

0	1	6	6	6	6	7	8	6
0	1	6	6	6	6	6	7	8

{0} {1} {2, 3, 4, 5, 6, 9} {7} {8}



Fast find

Fast find strategy: Let each component be identified by the label of one vertex in that component. Store these “component ids” in an int array such that $\text{id}[i] == \text{component id of element } i$.

Advantage: find, connected, and count are fast (and trivial to write)

Disadvantage: union will access every array element each time it's called.

Cost of find

$O(1)$ per find

$O(N)$ for a sequence of N finds

Cost of union

$O(N)$ per union

$O(N^2)$ for a sequence of N unions

Fast union

Fast union

Fast union strategy: Let each component be identified by the label of one vertex in that component, its “root”. Store these “component ids” in an int array such that $\text{id}[i] ==$ the “parent” of element i . The i for which $\text{id}[i] == i$ is the component root.

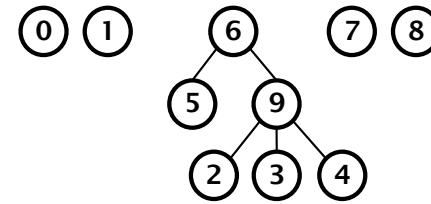
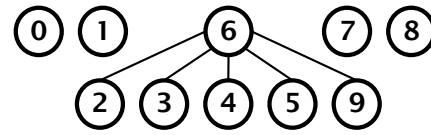
{0} {1} {2, 3, 4, 5, 6, 9} {7} {8}

Fast find

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

Fast union

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



Fast union

Fast union strategy: Let each component be identified by the label of one vertex in that component, its “root”. Store these “component ids” in an int array such that $\text{id}[i] ==$ the “parent” of element i . The i for which $\text{id}[i] == i$ is the component root.

Initially:

{0} {1} {2} {3} {4} {5}

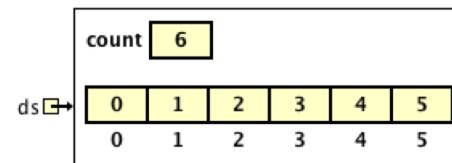
0	1	2	3	4	5
0	1	2	3	4	5

```
public class FastUnionDS implements DisjointSet {

    private int[] id; // component ids
    private int count; // number of components

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

```
FastUnionDS ds;
ds = new FastUnionDS(6);
```

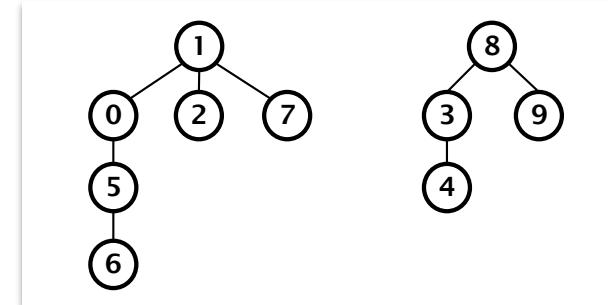


Fast union

```
public class FastUnionDS implements DisjointSet {  
  
    private int[] id; // component ids  
    private int count; // number of components  
  
    /**  
     * return component for p  
     */  
    public int find(int p) {  
        while (p != id[p])  
            p = id[p];  
        return p;  
    }  
}
```

Must traverse parents until the root is found

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



find(8) == 8

find(9) == 8

find(4) == 8

find(6) == 1

Fast union

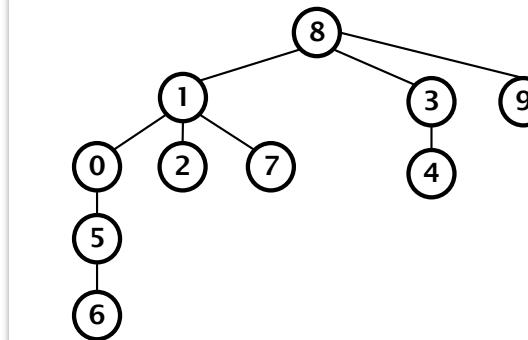
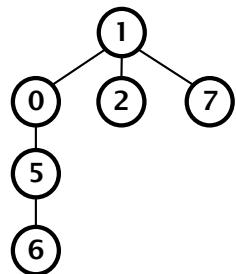
Fast union strategy: Let each component be identified by the label of one vertex in that component, its “root”. Store these “component ids” in an int array such that $\text{id}[i] == \text{the “parent” of element } i$. The i for which $\text{id}[i] == i$ is the component root.

For each $\text{union}(x, y)$, set $\text{id}[\text{root}(x)]$ to $\text{root}(y)$.

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

union(2, 4)

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



Fast union

```
public class FastUnionDS implements DisjointSet {  
  
    public void union(int p, int q) {  
        int pr = find(p);  
        int qr = find(q);  
        if (pr == qr) return;  
  
        id[pr] = qr; }  
        count--;  
    }  
}
```

Exactly the same as in the fast find version,
except that find() returns the component
root.

Updates only one id per call. The
root of p is set to the root of q

```
FastUnionDS ds = new FastUnionDS(6);
```

0	1	2	3	4	5
0	1	2	3	4	5

```
ds.union(0, 1);      r(0) == 0, r(1) == 1
```

0	1	2	3	4	5
0	1	2	3	4	5

1	1	2	3	4	5
0	1	2	3	4	5

```
ds.union(3, 4);      r(3) == 3, r(4) == 4
```

1	1	2	3	4	5
0	1	2	3	4	5

1	1	2	4	4	5
0	1	2	3	4	5

```
ds.union(1, 3);      r(1) == 1, r(3) == 4
```

1	1	2	4	4	5
0	1	2	3	4	5

1	4	2	4	4	5
0	1	2	3	4	5

```
ds.union(2, 1);      r(2) == 2, r(1) == 4
```

1	4	2	4	4	5
0	1	2	3	4	5

1	4	4	4	4	5
0	1	2	3	4	5

Fast union

```
FastUnionDS ds = new FastUnionDS(10);
```

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

```
ds.union(4, 3)
```

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

```
ds.union(3, 8)
```

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

```
ds.union(6, 5)
```

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

```
ds.union(9, 4)
```

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

```
ds.union(2, 1)
```

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

```
ds.union(5, 0)
```

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

```
ds.union(7, 2)
```

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

```
ds.union(6, 1)
```

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

```
ds.union(7, 3)
```

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

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

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

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

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

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

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

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

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

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

Fast union

```
public class FastUnionDS implements DisjointSet {  
  
    private int[] id; // component ids  
    private int count; // number of components  
  
    public int find(int p) {  
        while (p != id[p])  
            p = id[p];  
        return p;  
    }  
  
    public boolean connected(int p, int q) {  
        return find(p) == find(q);  
    }  
  
    public int count() {  
        return count;  
    }  
}
```

Must traverse parents until the root is found

No change – exactly the same as before

Fast union

Fast union strategy: Let each component be identified by the label of one vertex in that component, its “root”. Store these “component ids” in an int array such that $\text{id}[i] ==$ the “parent” of element i . The i for which $\text{id}[i] == i$ is the component root.

Advantage: Union updates less on average

Disadvantage: Find and connected no longer constant; union has linear worst case

Cost of find

$O(N)$ per find

$O(N^2)$ for a sequence of N finds

Cost of union

$O(N)$ per union

$O(N^2)$ for a sequence of N unions

Path compression

Comparison of unions

FastFindDS

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

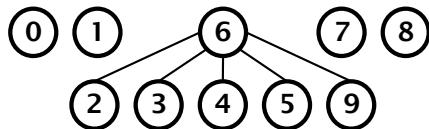
0	1	2	3	4	6	6	7	8	9
0	1	2	3	4	5	6	7	8	9

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

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

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

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



FastUnionDS

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

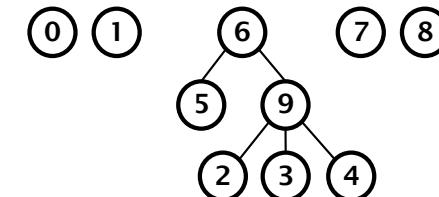
0	1	2	3	4	6	6	7	8	9
0	1	2	3	4	5	6	7	8	9

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

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

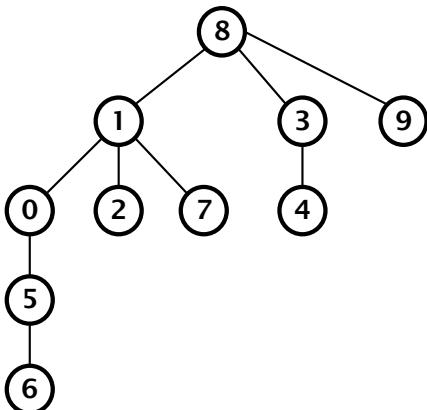
0	1	9	9	9	6	6	7	8	9
0	1	2	3	4	5	6	7	8	9

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



Fast union, tall trees

```
FastUnionDS ds = new FastUnionDS(10);
```



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

```
ds.union(4, 3)
```

```
ds.union(3, 8)
```

```
ds.union(6, 5)
```

```
ds.union(9, 4)
```

```
ds.union(2, 1)
```

```
ds.union(5, 0)
```

```
ds.union(7, 2)
```

```
ds.union(6, 1)
```

```
ds.union(7, 3)
```

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

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

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

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

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

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

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

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

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

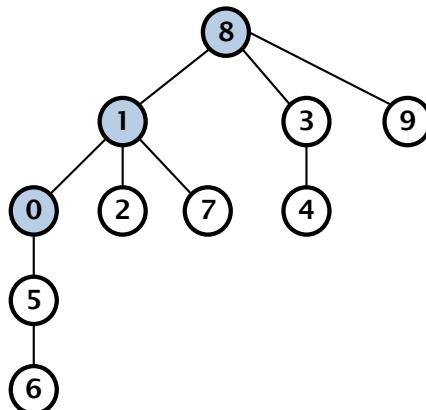
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

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

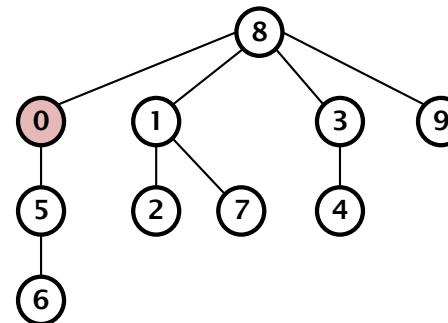
Path compression

Each time we find the root of an element, replace the parent value of each examined element with the root value.

`find(0) == 8`



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

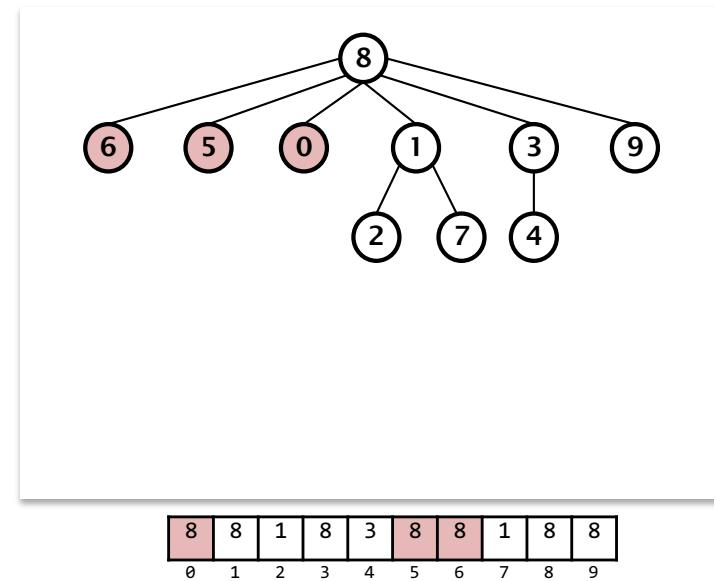
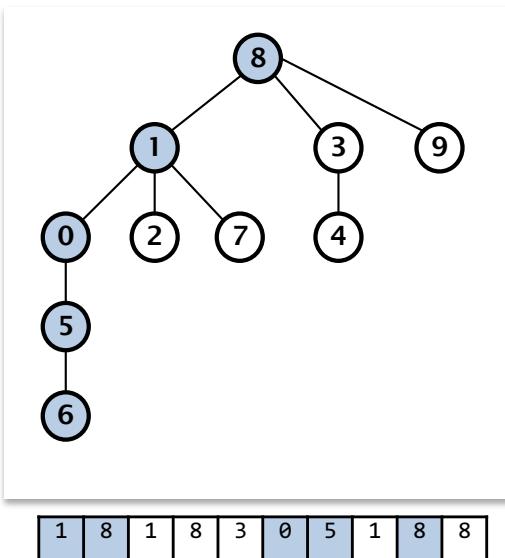


8	8	1	8	3	0	5	1	8	8
0	1	2	3	4	5	6	7	8	9

Path compression

Each time we find the root of an element, replace the parent value of each examined element with the root value.

`find(6) == 8`



Path compression

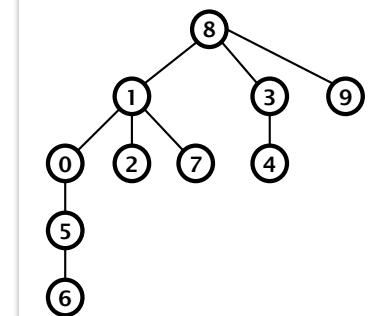
```
public class FastUnionDS implements DisjointSet {  
  
    public int find(int p) {  
        int root = p;  
        while (root != id[root])  
            root = id[root];  
  
        while (p != root) {  
            int newp = id[p];  
            id[p] = root;  
            p = newp;  
        }  
        return root;  
    }  
}
```

Find the root of p.

Compress the path from p to root.

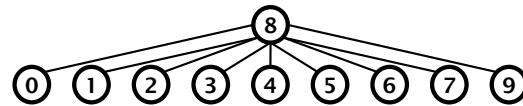
Without path compression:

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



With path compression:

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



Fast union with path compression

Second implementation strategy: Let each component be identified by the label of one vertex in that component, its “root”. Store these “component ids” in an int array such that $\text{id}[i] == \text{the “parent” of element } i$. The i for which $\text{id}[i] == i$ is the component root.

Each time we find the root of an element, replace the parent value of each examined element with the root value.

Cost of find

$O(\log N)$ per find

$O(N \log N)$ for a sequence of N finds

Cost of union

$O(\log N)$ per union

$O(N \log N)$ for a sequence of N unions

Example

Example union sequence

Fast Find:

union(4, 3)	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	{0} {1} {2} {3} {4} {5} {6} {7} {8} {9}
union(3, 8)	[0, 1, 2, 3, 3, 5, 6, 7, 8, 9]	{0} {1} {2} {3, 4} {5} {6} {7} {8} {9}
union(6, 5)	[0, 1, 2, 8, 8, 5, 6, 7, 8, 9]	{0} {1} {2} {5} {6} {7} {3, 4, 8} {9}
union(9, 4)	[0, 1, 2, 8, 8, 5, 5, 7, 8, 9]	{0} {1} {2} {5, 6} {7} {3, 4, 8} {9}
union(2, 1)	[0, 1, 1, 8, 8, 5, 5, 7, 8, 8]	{0} {1, 2} {5, 6} {7} {3, 4, 8, 9}
union(5, 0)	[0, 1, 1, 8, 8, 0, 0, 7, 8, 8]	{0, 5, 6} {1, 2} {7} {3, 4, 8, 9}
union(7, 2)	[0, 1, 1, 8, 8, 0, 0, 1, 8, 8]	{0, 5, 6} {1, 2, 7} {3, 4, 8, 9}
union(6, 1)	[1, 1, 1, 8, 8, 1, 1, 1, 8, 8]	{0, 1, 2, 5, 6, 7} {3, 4, 8, 9}
union(7, 3)	[8, 8, 8, 8, 8, 8, 8, 8, 8]	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Example union sequence

Fast Union:

union(4, 3)	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	{0} {1} {2} {3} {4} {5} {6} {7} {8} {9}
union(3, 8)	[0, 1, 2, 3, 3, 5, 6, 7, 8, 9]	{0} {1} {2} {3, 4} {5} {6} {7} {8} {9}
union(6, 5)	[0, 1, 2, 8, 3, 5, 6, 7, 8, 9]	{0} {1} {2} {5} {6} {7} {3, 4, 8} {9}
union(9, 4)	[0, 1, 2, 8, 3, 5, 5, 7, 8, 9]	{0} {1} {2} {5, 6} {7} {3, 4, 8} {9}
union(2, 1)	[0, 1, 1, 8, 3, 5, 5, 7, 8, 8]	{0} {1, 2} {5, 6} {7} {3, 4, 8, 9}
union(5, 0)	[0, 1, 1, 8, 3, 0, 5, 7, 8, 8]	{0, 5, 6} {1, 2} {7} {3, 4, 8, 9}
union(7, 2)	[0, 1, 1, 8, 3, 0, 5, 1, 8, 8]	{0, 5, 6} {1, 2, 7} {3, 4, 8, 9}
union(6, 1)	[1, 1, 1, 8, 3, 0, 5, 1, 8, 8]	{0, 1, 2, 5, 6, 7} {3, 4, 8, 9}
union(7, 3)	[1, 8, 1, 8, 3, 0, 5, 1, 8, 8]	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Example union sequence

Path Compression:

union(4, 3)	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	{0} {1} {2} {3} {4} {5} {6} {7} {8} {9}
union(3, 8)	[0, 1, 2, 3, 3, 5, 6, 7, 8, 9]	{0} {1} {2} {3, 4} {5} {6} {7} {8} {9}
union(6, 5)	[0, 1, 2, 8, 3, 5, 6, 7, 8, 9]	{0} {1} {2} {5} {6} {7} {3, 4, 8} {9}
union(9, 4)	[0, 1, 2, 8, 8, 5, 5, 7, 8, 8]	{0} {1} {2} {5, 6} {7} {3, 4, 8} {9}
union(2, 1)	[0, 1, 1, 8, 8, 5, 5, 7, 8, 8]	{0} {1, 2} {5, 6} {7} {3, 4, 8, 9}
union(5, 0)	[0, 1, 1, 8, 8, 0, 5, 7, 8, 8]	{0, 5, 6} {1, 2} {7} {3, 4, 8, 9}
union(7, 2)	[0, 1, 1, 8, 8, 0, 5, 1, 8, 8]	{0, 5, 6} {1, 2, 7} {3, 4, 8, 9}
union(6, 1)	[1, 1, 1, 8, 8, 0, 0, 1, 8, 8]	{0, 1, 2, 5, 6, 7} {3, 4, 8, 9}
union(7, 3)	[1, 8, 1, 8, 8, 0, 0, 1, 8, 8]	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Applications

Kruskal's algorithm

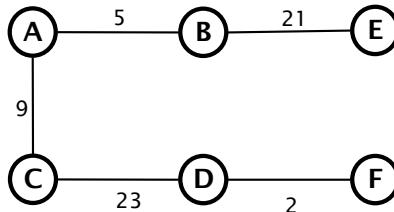
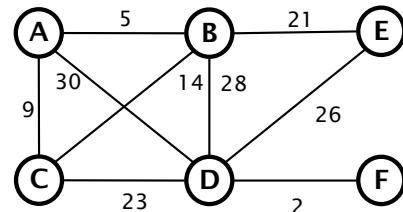
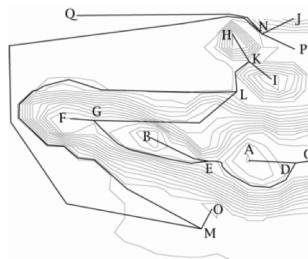
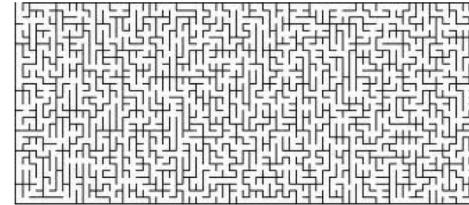


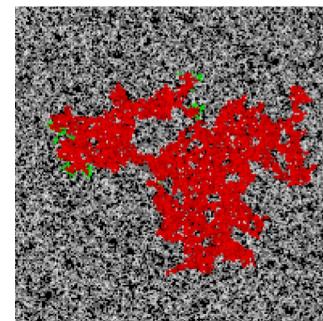
Image processing



Maze layout, generation



Percolation system modeling



water, oil, etc.
through ground

disease through
populations