ID1020: Union-Find

Dr. Per Brand
pbrand@kth.se

kap. 1.5

Slides adapted from Algorithms 4th Edition, Sedgewick.

Developing an algorithm

- Steps:
  - Construct a model of the problem
  - Find an algorithm that solves the problem.
  - Is the algorithm fast enough? Memory usage ok?
  - If not determine why not.
  - Find a better algorithm.
  - Itererate until satisfied.

- Thereafter analyze the algorithm. Determine how it scales.

Case study: Dynamic-connectivity problem

- Given N objects, the program should support the following:
  - **Union**: Connect two of the objects.
  - **Find**: Determine if their is path (of connections) between any two objects.
  - If there is a path then the two objects are in the same component

connect 4 and 3

connect 3 and 8

connect 6 and 5

connect 9 and 4

connect 2 and 1

are 0 and 7 connected? X

are 8 and 9 connected? ✓

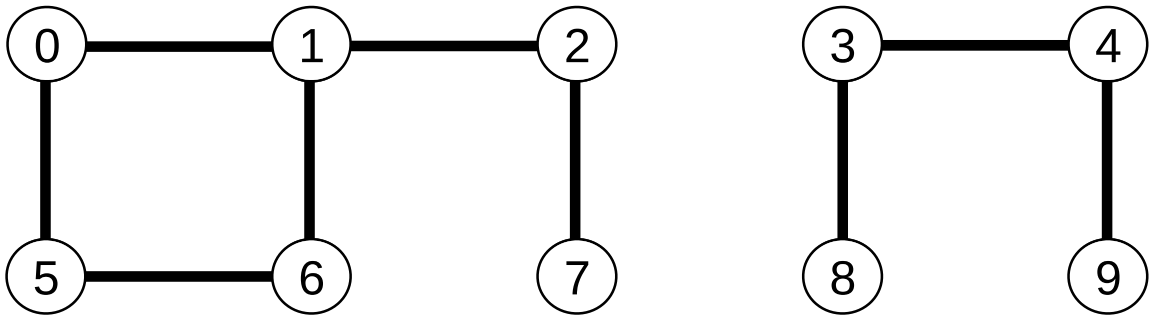
connect 5 and 0

connect 7 and 2

connect 6 and 1

connect 1 and 0

are 0 and 7 connected? ✓



A large connectivity problem

- Is there a path between P andQ ?

Yes

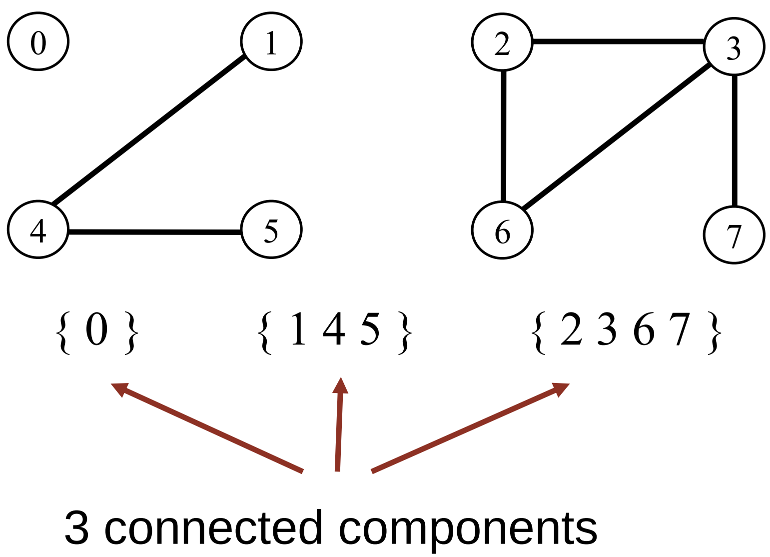
Applications and the model

- Applications
  - Pixels in a computer image.
  - Computers in a network.
  - Friends in a social network.
  - Transistors on a chip.
  - Elements in mathematical sets.
  - Variable names in a Fortran program.

- To simplify programming, we call the objects 0 to N – 1.
  - Use the integer as an array index.
  - Abstract away the details not needed to solve/model union-find.
  - a[i] == a[j] for all objects in the same component

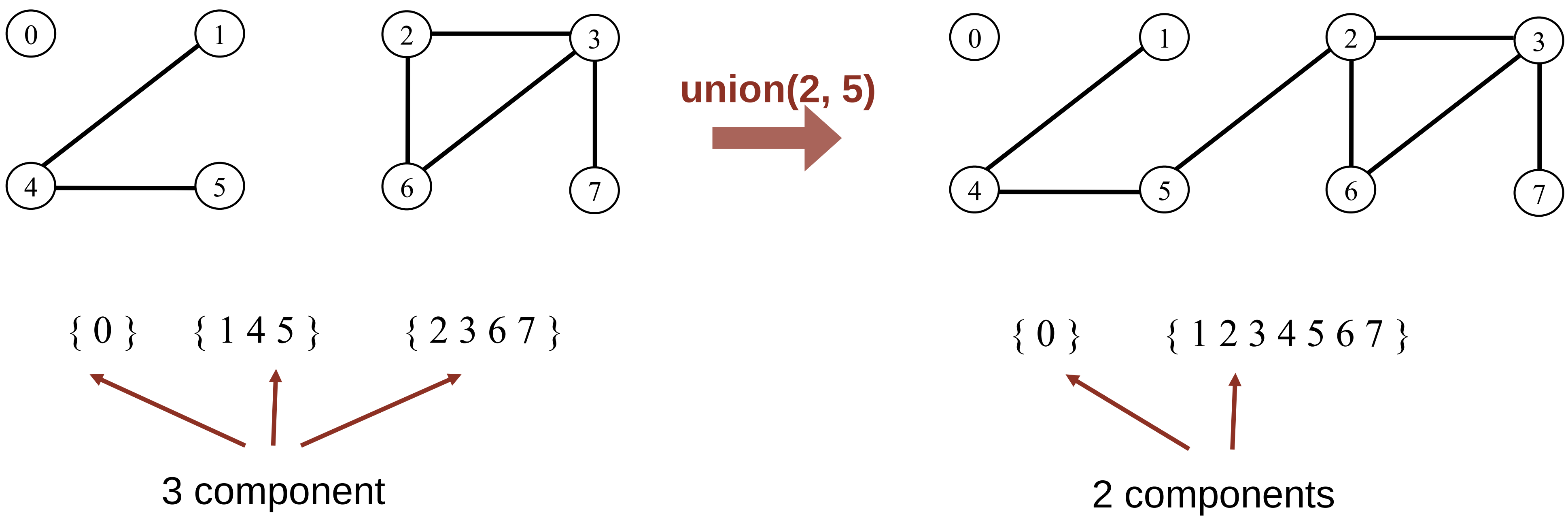
The model

- Connectedness is an equivalence relation:
  - Reflexive: *p* is connected to itself.
  - 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 partition the objects into equivalence classes which in this case we call ”connected components”.



Find and Union operations

- Component Identifier: Unique id for each component
- Find. In what component do we find *p* ?
- Connected. Are *p* and *q* in the same component?
- Union. All the objects in the components that *p* and *q* become part of the same component. (In one component we replace the component identifier with that of the other component)



Union-find datatype (API)

- Goal. Efficient datastructure for union-find.
  - The number of objects N might be very large.
  - The number of operations M might also be very large.
  - Client can interleave find and union operations in any order.

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?*

connected()

- `connected()` is implemented in one line:

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

Example of client

- Read in the numbers objects N from stdin (standard input).
- Repeat while stdin is non-empty
  1. Read a pair of integers from stdin
  2. If not connected, create a connection (using union) and print the pair.

```
public static void main(String[] args) {
    int N = StdIn.readInt();
    UF uf = new UF(N);
    while (!StdIn.isEmpty())   {
        int p = StdIn.readInt();
        int q = StdIn.readInt();
        if (!uf.connected(p, q)) {
            uf.union(p, q);
            StdOut.println(p + " " + q);
        }
    }
}
```

```
% more tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
```

← Number of object N

Already connected

Quick-find

Quick-find [eager method]

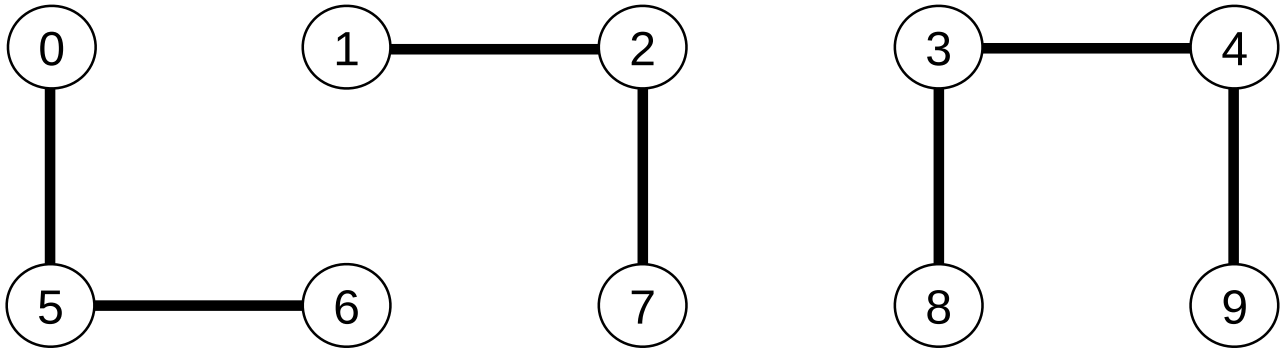
- Datastructure.
  - Integer array `id[]` of size `N`.
  - Representation: `id[p]` is the id of the component containing `p`.
    `p` and `q` are connected iff they have the same id.

	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



Quick-find operations

- Find p
  - Return id[p] the component identifier
- Connected(p,q)
  - Ís id[p]==id[q]]?

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

id[6] = 0; id[1] = 1

6 ad 1 are not connected

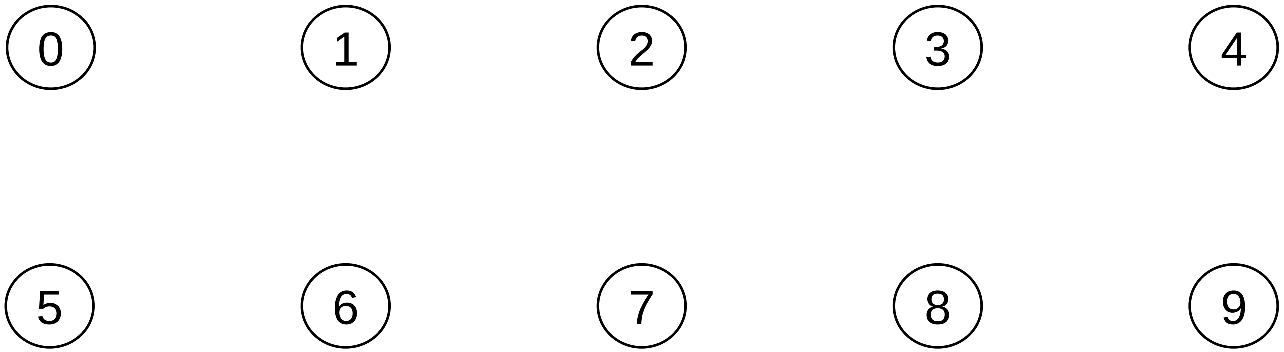
- Union. To unify two components (where `p` and `q`, respectively are memtbers) change all elements = `id[p]` to `id[q]`.

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

After union of 6 and 1

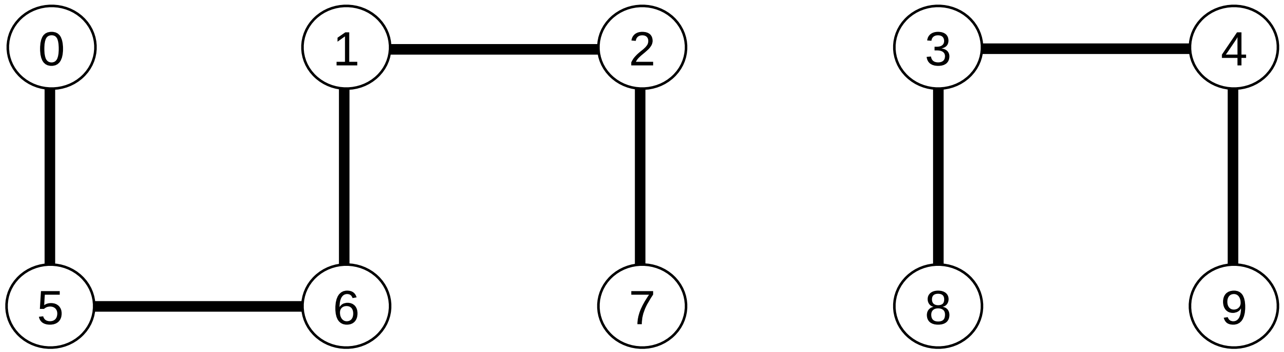
note: whole array needs to be checked

Quick-find demo



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

Quick-find demo



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

Quick-find: Java implementation

```
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 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;
      }
   }
}
```

Initially: each object is identified by its own id (N array accesses)

Return the id of p (1 array access)

Change all elements with id[p] till id[q] (at most 2N + 2 array accesses)

Quick-find is slow

- Cost model. The number of array accesses

algoritm	initialize	union	find	connected
quick-find	N	N	1	1

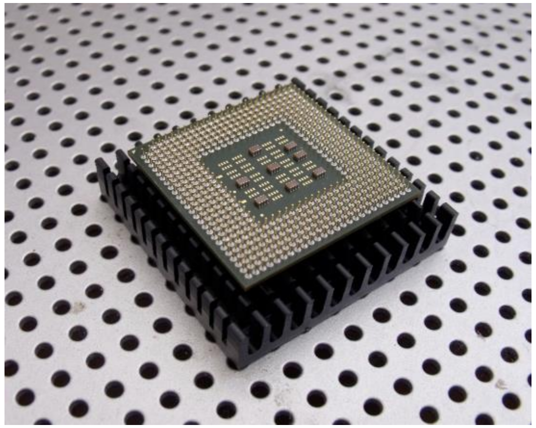
Number of array accesses

- Union is expensive. It will take N^2 array accesses to handle of N union operations on N objekt. Quadratic!!!

Quadratic algorithms don't scale

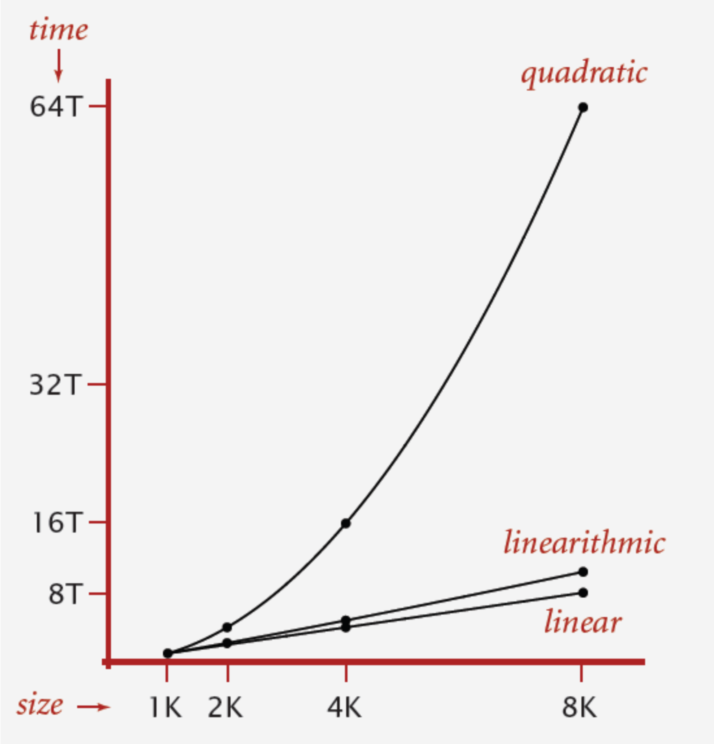
- Rule of thumb
  - 10^9 operations per second
  - 10^9 memory words in main memory.
  - To access entire memory takes about 1 secund.

Been this way since 1950!



- Consequences for quick-find
  - With10^9 union operations on 10^9 object, quick-find will perform 10^{18} operations=> 30+ years of computation!

- Quadratic algorithms don't scale better with technology improvements
  - New computer is 10x faster.
  - But, computer has 10x more memory => So we want to solve a problem that is 10x bigger.
  - With a quadratic algorithm, => 10x slower



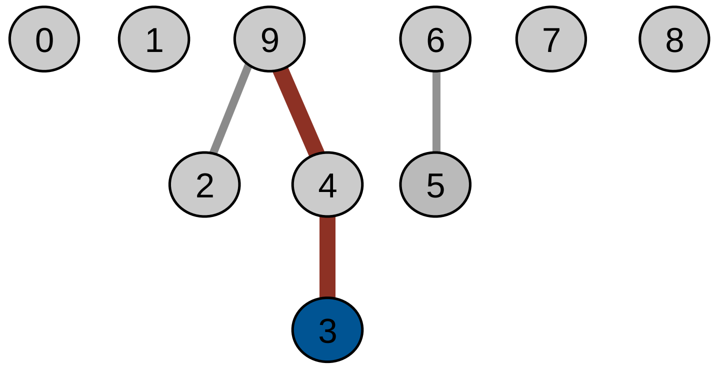
Quick Union

Quick-union [lazy metod]

- Datastructure.
  - Integer array id[] of length N.
  - Representation: id[i] is parent of i
  - Root: parent of itself
  - Root of i is
    `id[id[id[...id[i]...]]].`

Repeat until you reach root

(Note: not cycles)



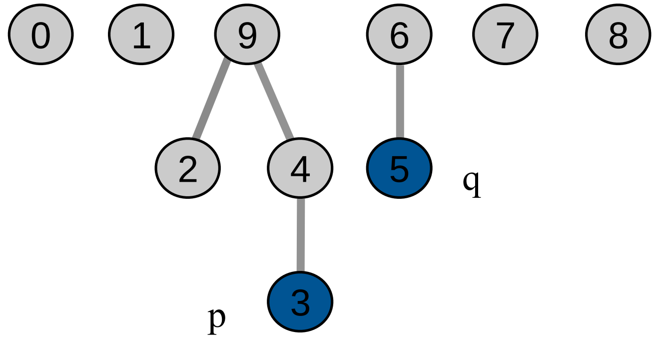
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9

Parent of 3 is 4

Parent of 3 is 9

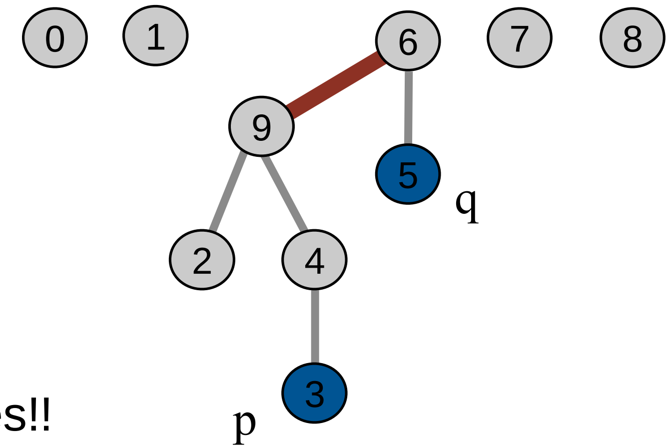
9 is a root

Quick-union



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

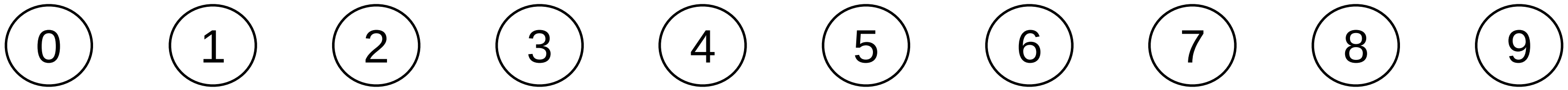
- Find. Determine the root of p?
- Connected. Determine if p and q have the same root?
- Union. To connect the components that contain p and q, assign the id of p's root to the id of q's root.



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

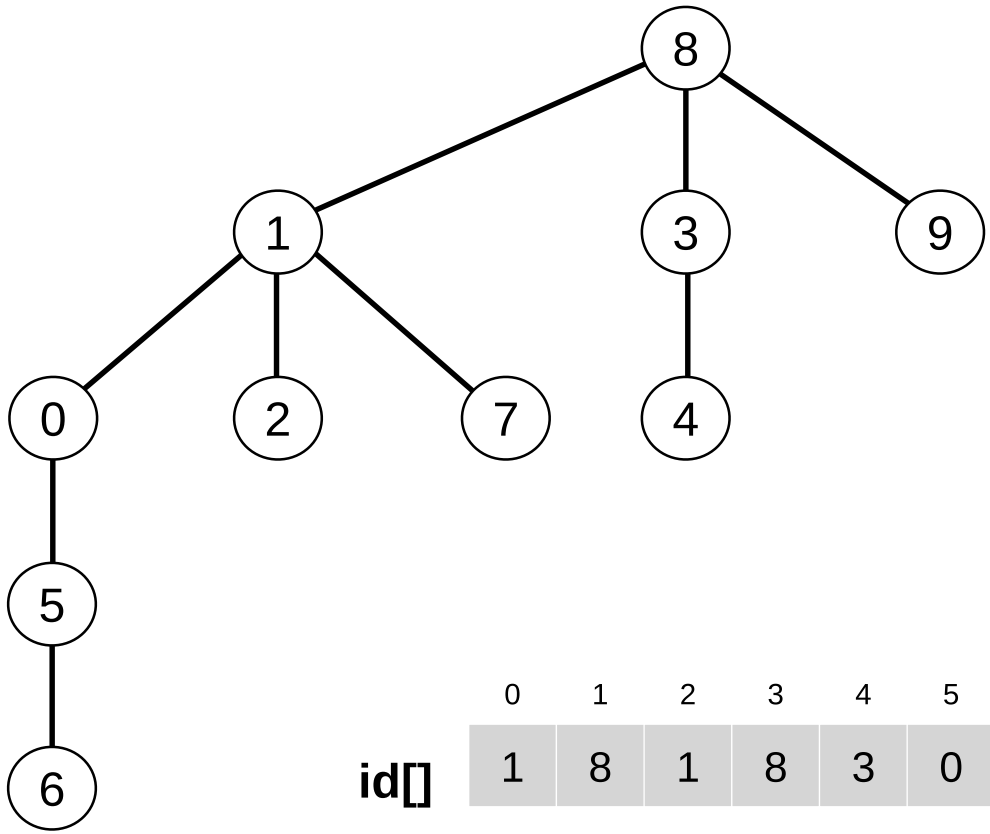
Only one value changes!!

Quick-union demo



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

Quick-union demo



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

Quick-union: Java 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;
   }
 }
 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;
 }
}
```

each object is initialized to be the root of a component with one member- itself (N array accesses)

follow parent links till root is reached (number of array accesses=depth)

make the root of p link to the root of q (number of array acceses = depth of q and p)

Quick-union is also slow

Cost model. Number of array accesses.

algoritm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N †	N	N

worst case

† includes cost of finding root

Quick-find defect.

- Union is too expensive. (N array accesses).
- Trees are ”flat", but it costs too much to keep them *flat*.

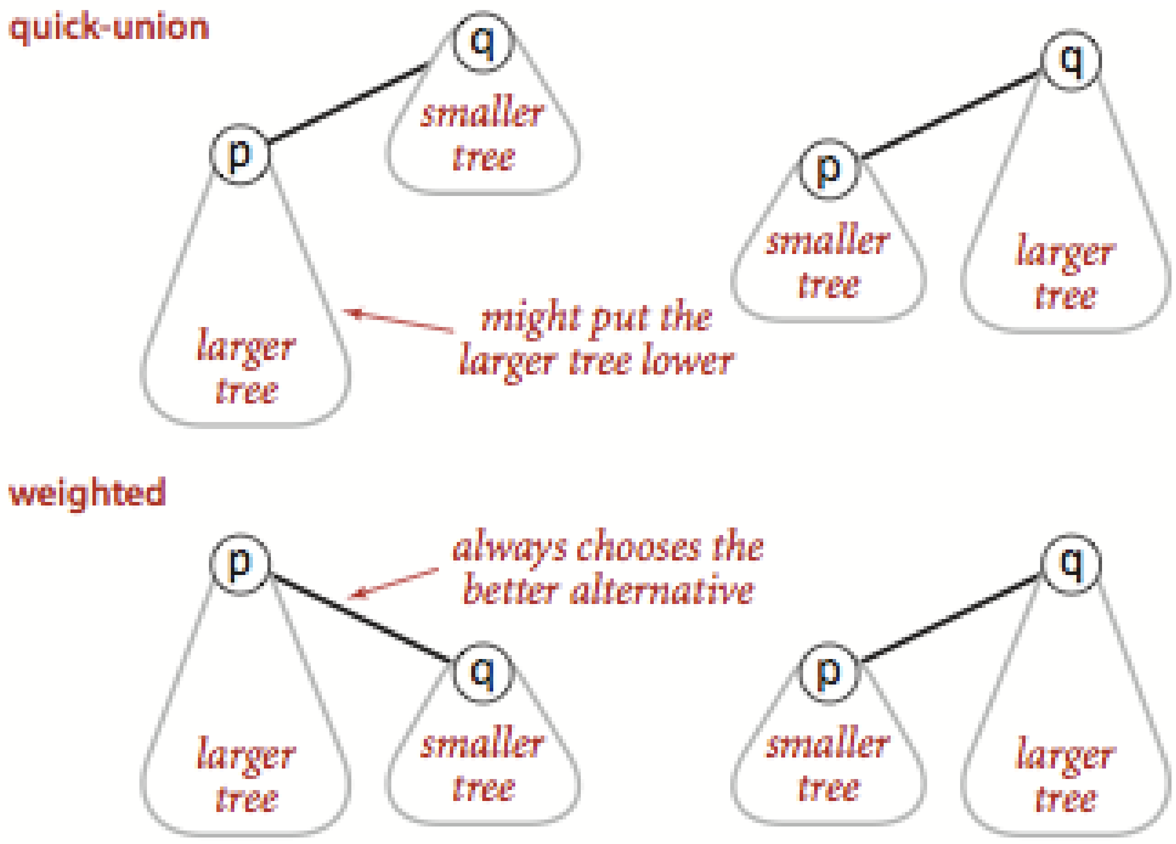
Quick-union defect.

- Trees can become deep.
- Find/connected/union to expensive (may take N array accesses).

Quick-union improvements

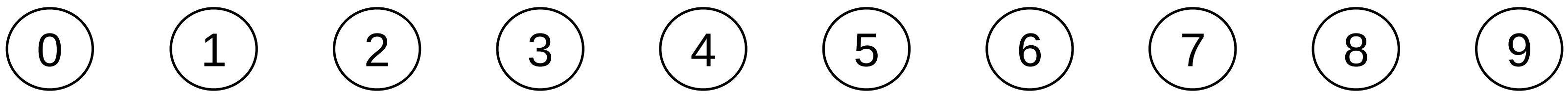
Improvement 1: Weights

- Weighted quick-union.
  - Change quick-union to avoid deep trees.
  - Store size of tree (number of objects).
  - Balance tree with linking the root of the smaller tree to the larger tree.



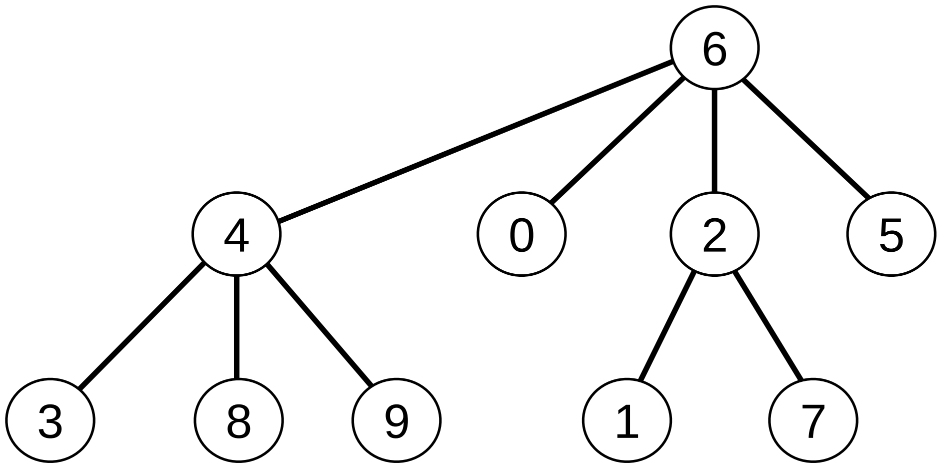
Weighted quick-union

Viktad quick-union demo



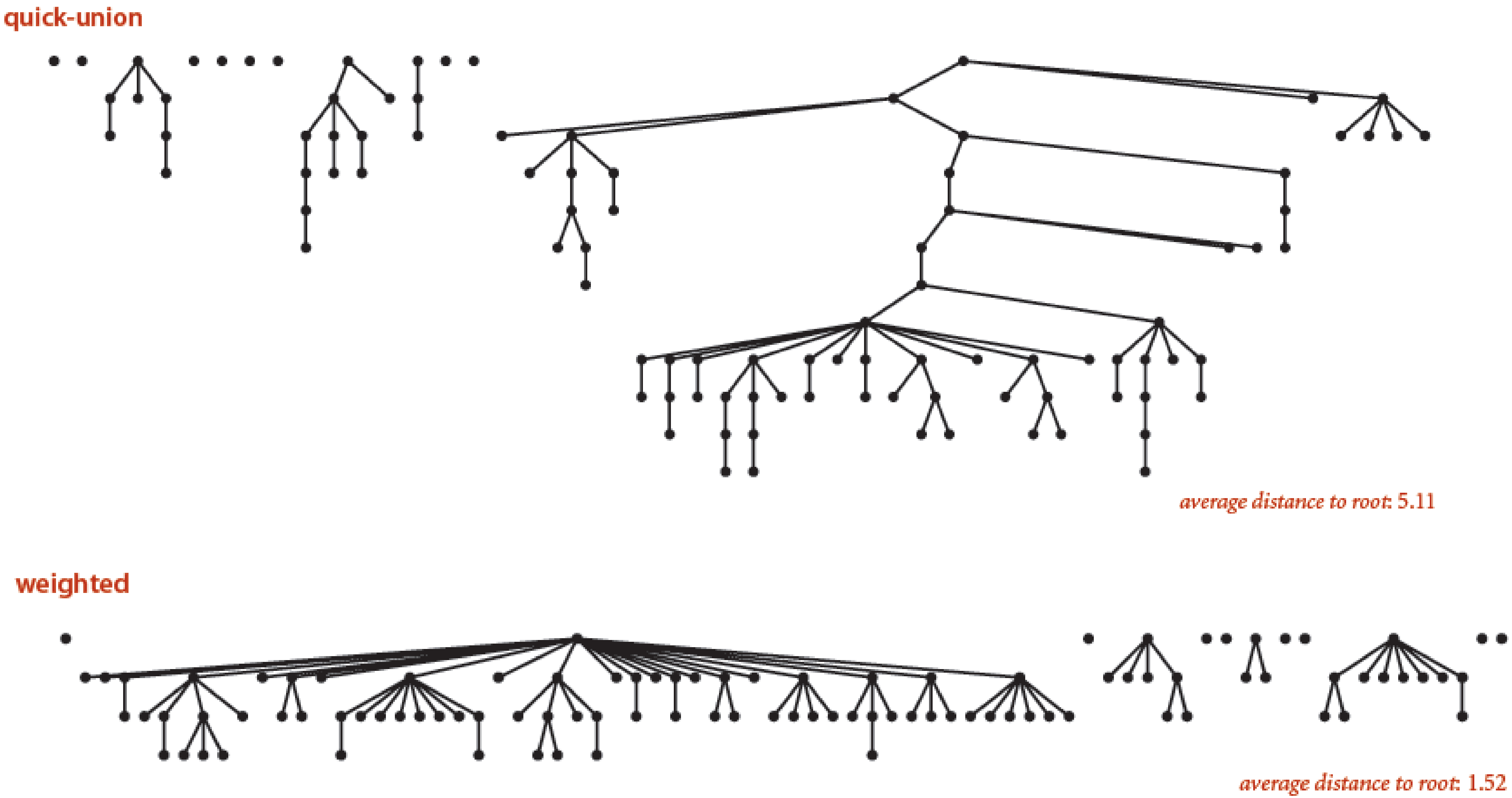
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	3	4	5	6	7	8	9

Viktad quick-union demo



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

Quick-union and weighted quick-union examples



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

Weighted quick-union: Java implementation

- Datastructure. Like quick-union, but we use an extra array `sz[i]` to keep track of number of object in tree

- Find/connected. The same as for quick-union.

- Union. Changes:
  - Link the root of the smaller tree to the larger.
  - Uppdate `sz[]` array.

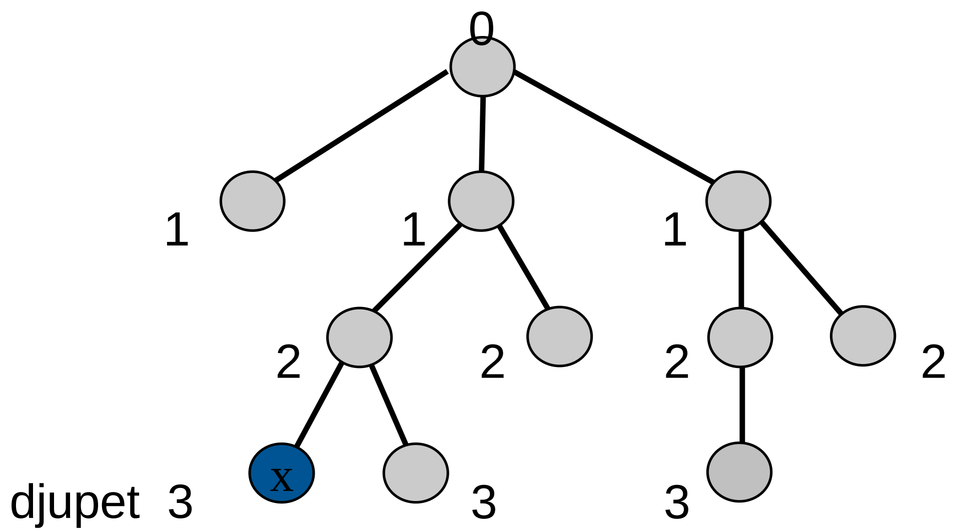
```
int i = find(p);
int j = find(q);
if (i == j) return;
if  (sz[i] < sz[j]) {
        id[i] = j; sz[j] += sz[i];
} else {
        id[j] = i; sz[i] += sz[j];
}
```

Weighted quick-union analysis

- Running time.
  - Find: time proportional to the depth of p.
  - Union: takes constrant time, given the roots.

lg = base-2 logarithm

- Theorem. Depth of object is at most $lg\ N$.

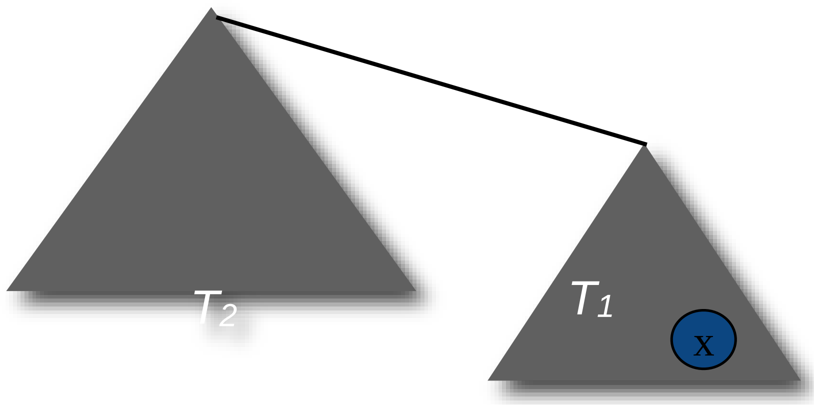


N = 11

djupet(x) = 3 ≤ lg N

Weighted quick-union analysis

- Theorem. Depth of object is at most $lg\ N$.
- Proof. What can increase the depth of node x ?
  - Increase by 1 when the tree `T1` contain X is merged with another tree `T2`.
  - Size of tree containing x is at least doubled as $|T_2| \geq |T_1|$.
  - Size of tree containing x can be doubled at most $lg\ N$ times. Why?



1

2

4

8

16

⋮

N

lg N

Weighted quick-union analysis

- Running time.
  - Find: time proportional to the depth of p i.e at most lgN.
  - Union: takes constrant time, given the roots. ~lgN

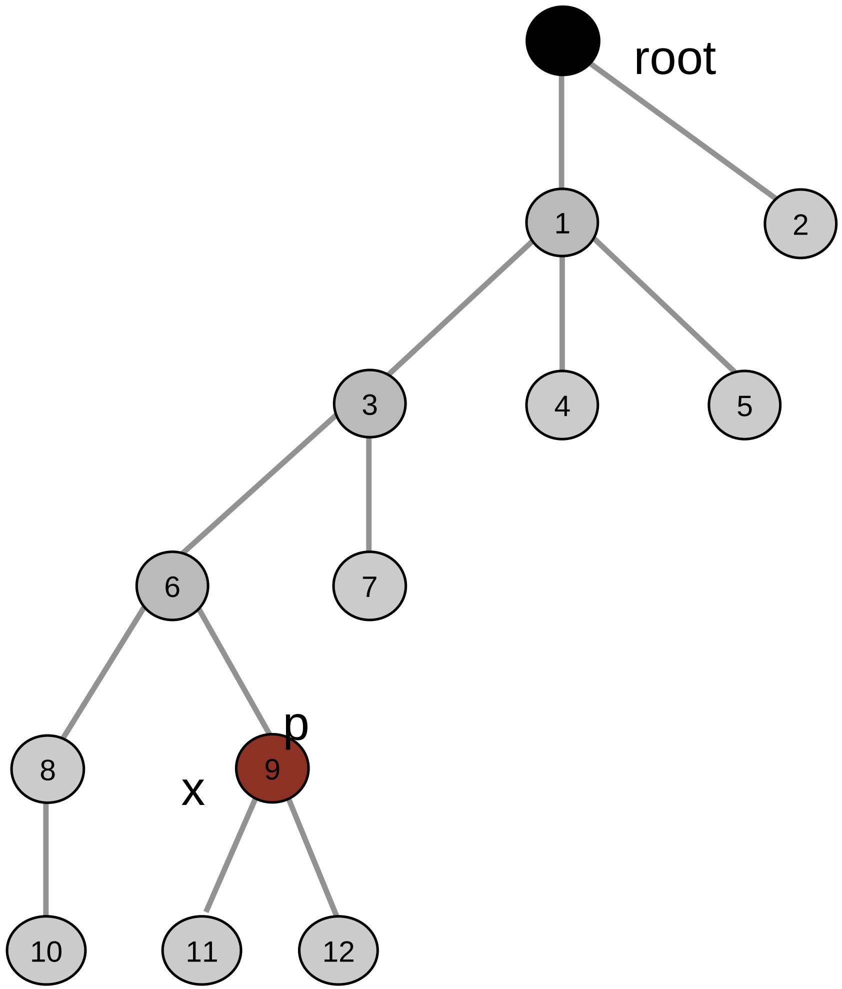
algoritm	**initialize**	**union**	**find**	**connected**
quick-find	N	N	1	1
quick-union	N	N †	N	N
viktad QU	N	lg N †	lg N	lg N

† including cost of finding root

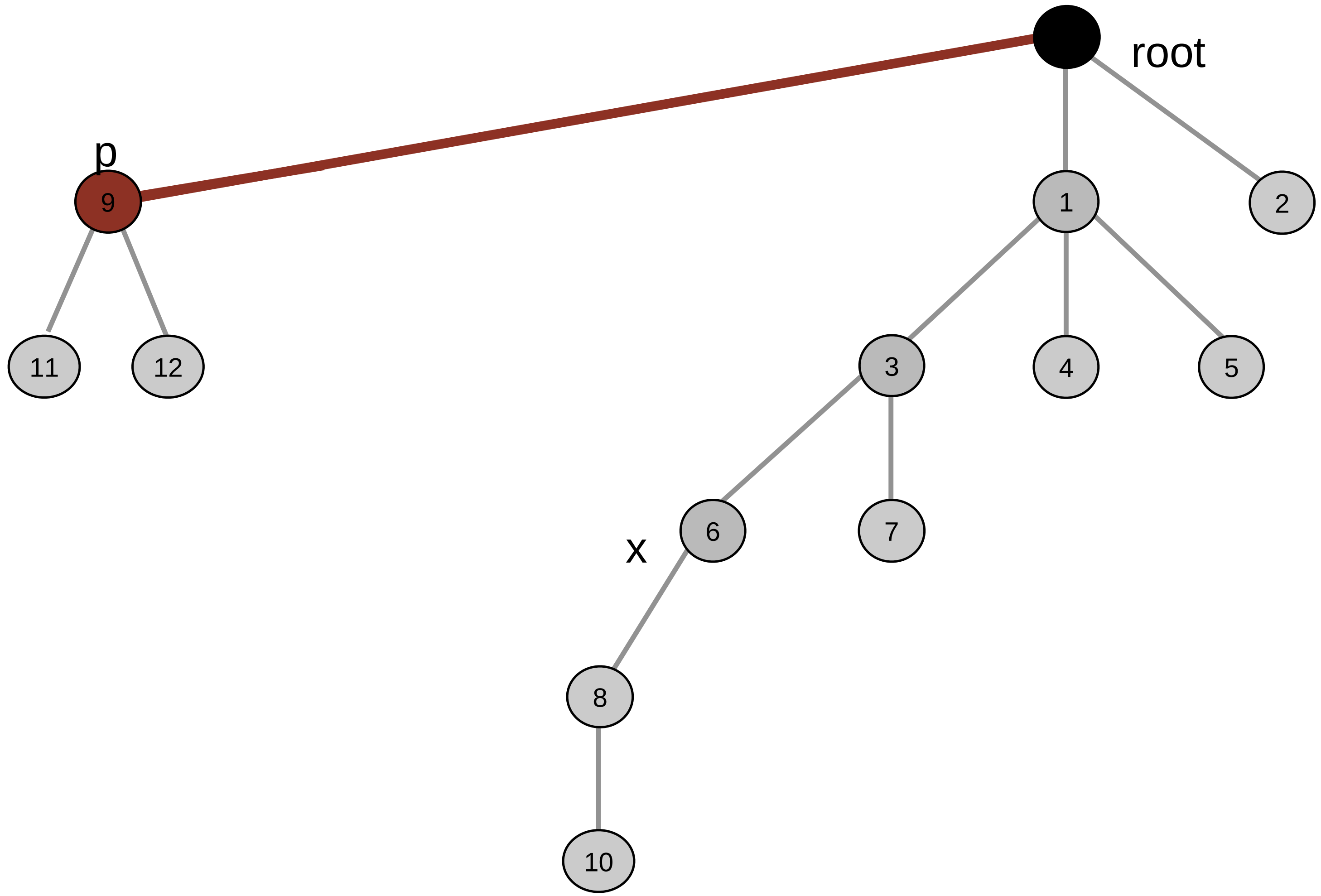
Can the algorithm be further improved. Yes, a bit.

Improvement 2: path compression

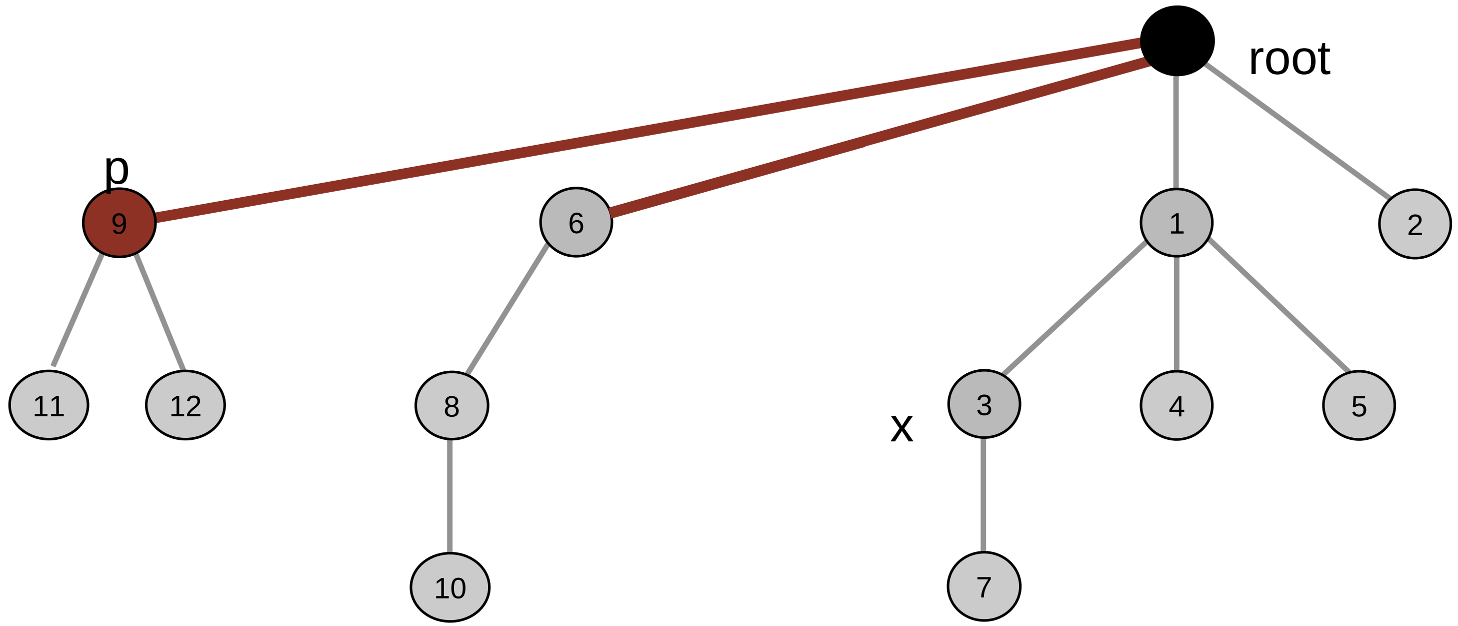
- Quick union with *path compression*. Flatten the tree when moving to find root.



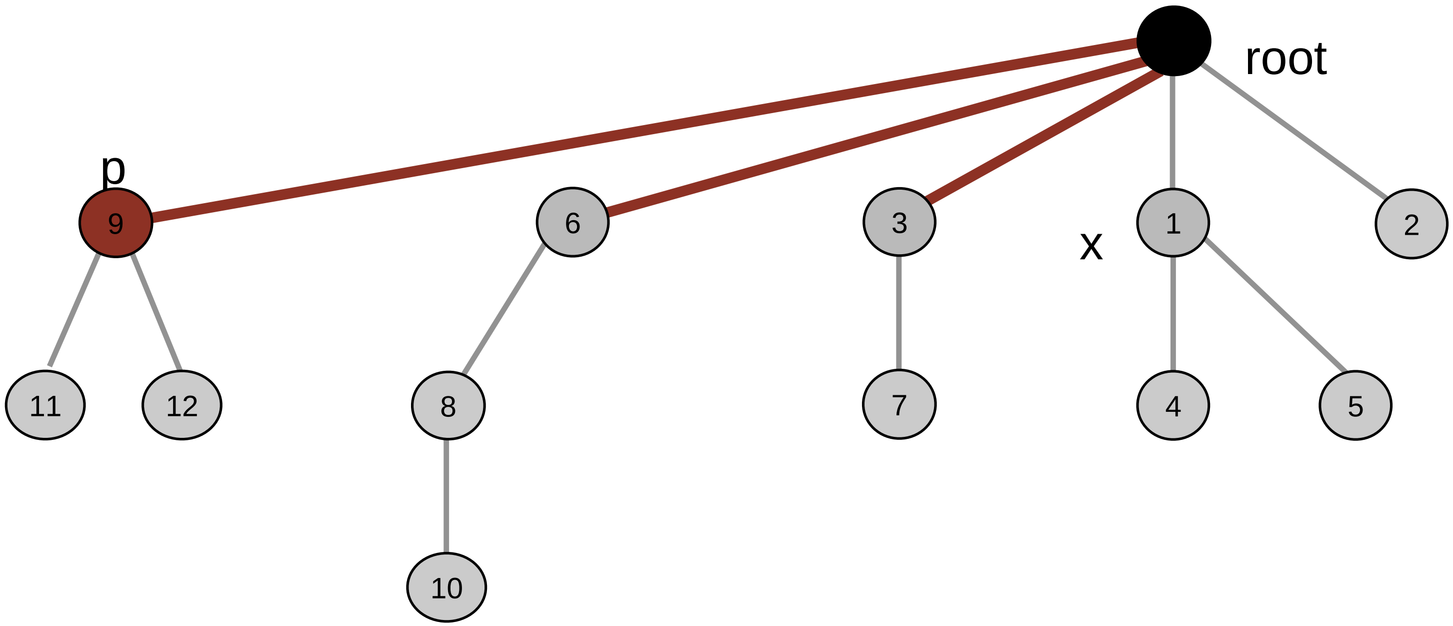
Improvement 2: path compression



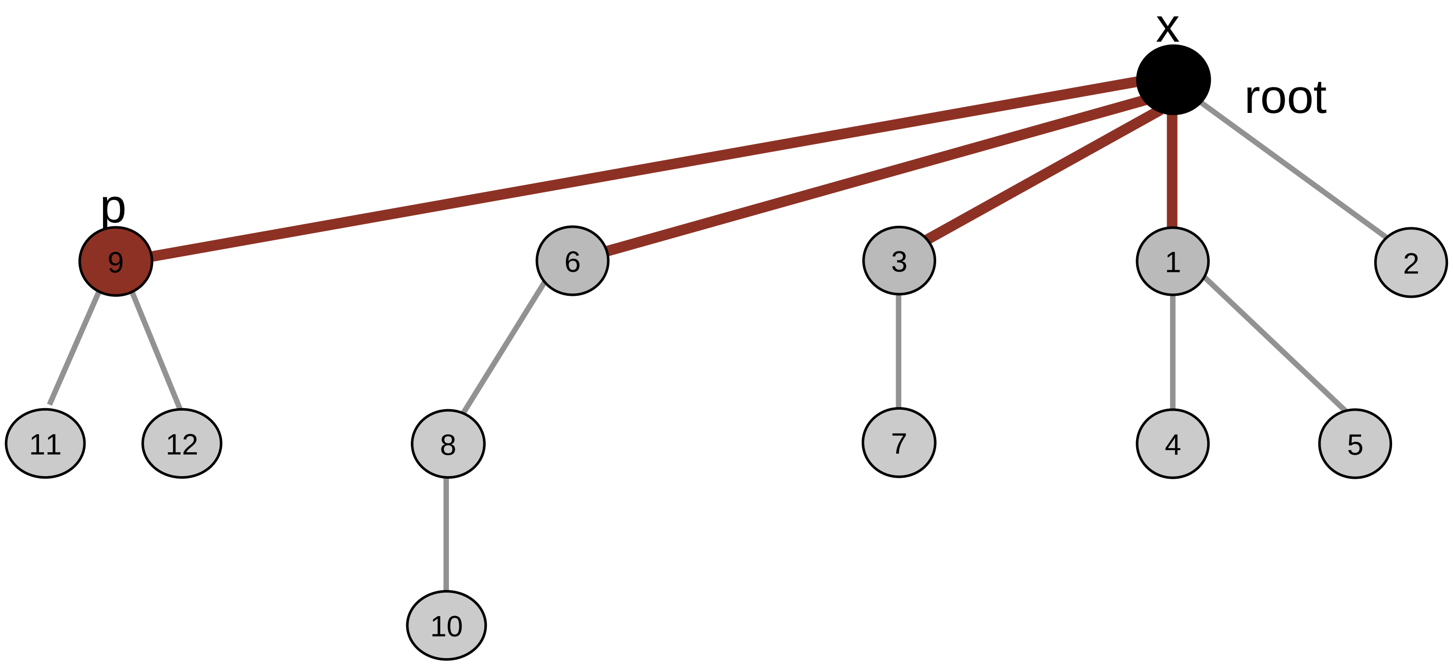
Improvement 2: path compression



Improvement 2: path compression



Improvement 2: path compression



Path compression: Java implementation

- Two step implementation: add an extra loopto`find()` to assign new root `id[]` to every examined root.

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

One extra line of code

Note. Path is compressed as the cost of increased constant overhead in certain operations.

Path compression even better?

No linear algoritm for M union-find operationer on object!!

- In theory WQUPC (weighted quick-union with path compression) is not linear
- But in practice it is WQUPC linjär.

Conclusion

- Weighted quick union (with or without path compression) enables problems to be solved that otherwise could not be

algoritm	worst-case tid
quick-find	M N
quick-union	M N
viktad QU	N + M log N
QU + stig komprimering	N + M log N
viktad QU + stig komprimering	Almost but not quite N + M

Runtime for M union-find operationer on N objects

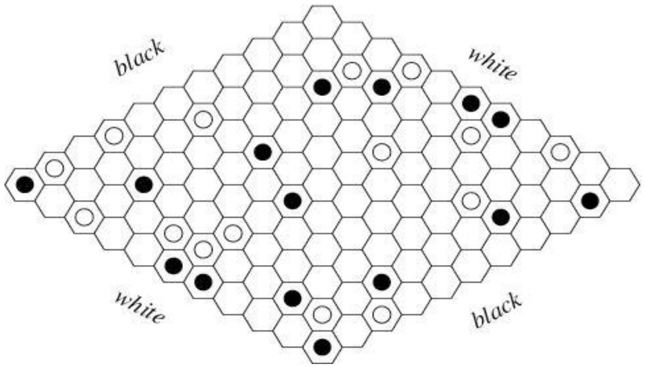
Ex. [10^9 unions and finds with 10^9 objects]

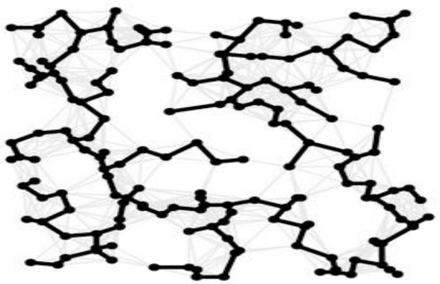
- WQUPC dimiishes running time from 30 years to 6 seconds.
- Faster computers don’t help – good algorithms do

Union-Find applications

Union-find applications

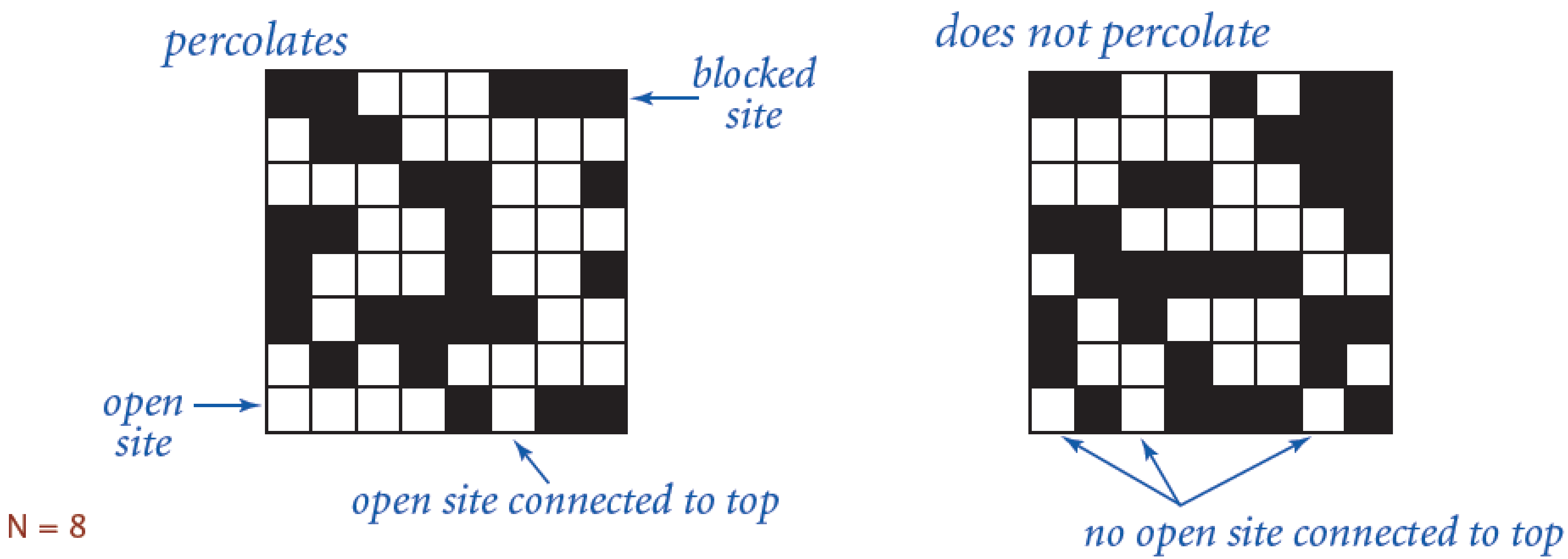
- Perkolation.
- Spel (Go, Hex).
- ✓ Dynamisk konnektivitet.
- Least common ancestor.
- Ekvivalens av ändliga tillståndsautomater.
- Hinley-Milner polymorphic type inference.
- Kruskal's spanning tree algorithm.
- Matlab's bwlabel() function i image processing.





Percolation

- An abstract model of many physical systems
  - N x N grid of *sites*.
  - A site is open with probablity p (and blocked with probability $1 - p$).
  - The system will percolate iff top och bottom are connected via genom open sites.



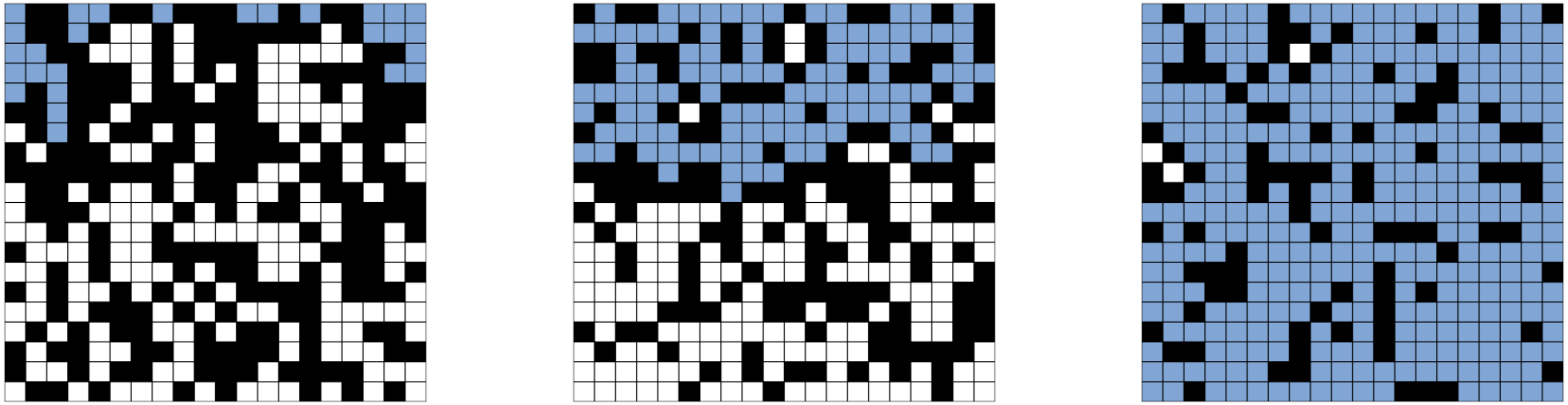
Percolation

- An abstract model of many physical systems
  - N x N grid of *sites*.
  - A site is open with probablity p
    (and blocked with probability $1 - p$).
  - The system will percolate iff top och bottom are connected via genom open sites

model	system	vacant site	occupied site	percolates
electricity	material	conductor	insulated	conducts
fluid flow	material	empty	blocked	porous
social interaction	population	person	empty	communicates

Percolation

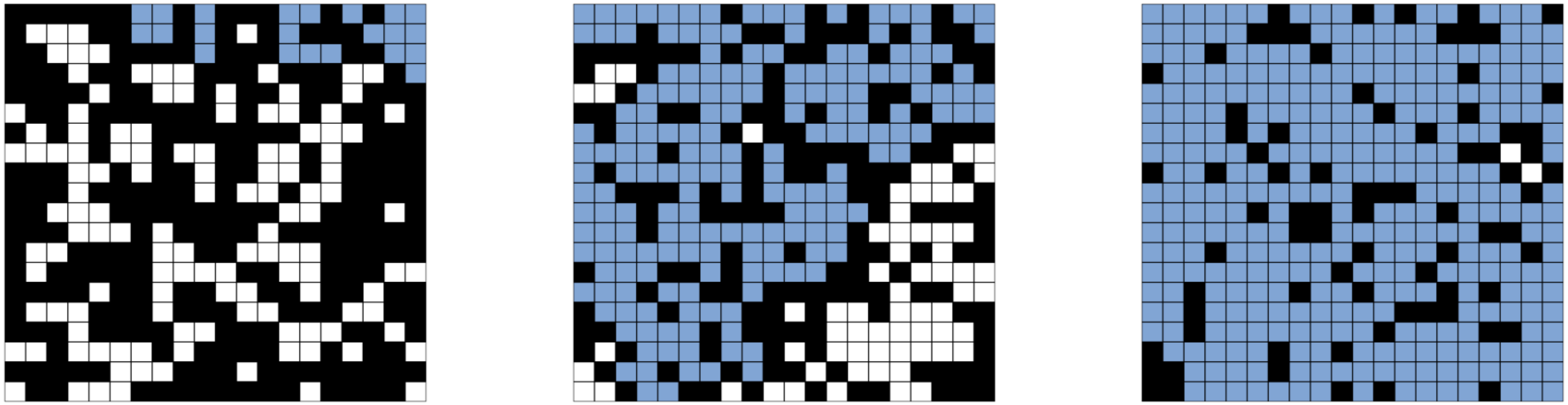
- Depending on grid size N and *site vacancy* probability p.



p low (0.4)
does not percolate

p medium (0.6)
percolates?

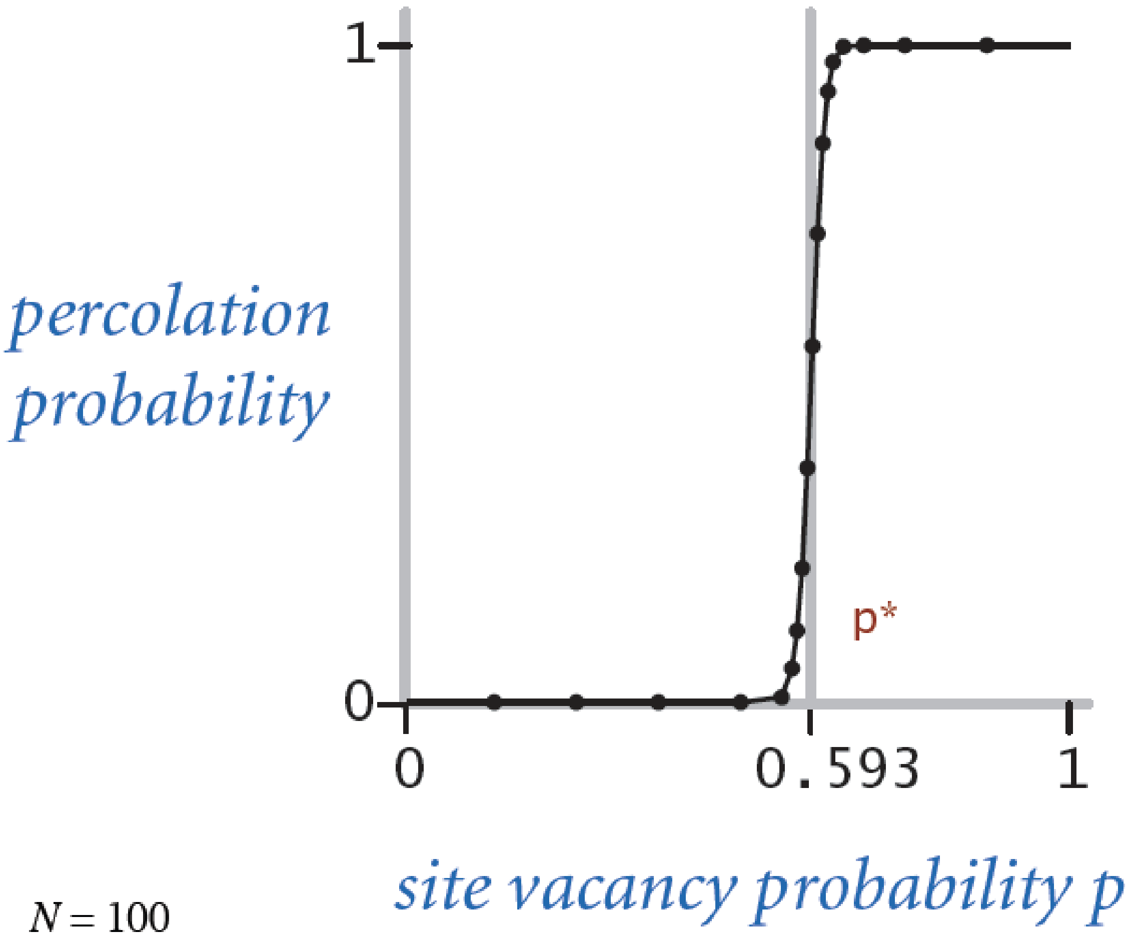
p high (0.8)
percolates



Percolation phase transition

- When N is large, teory guarantees a sharp threshold p^*.
  - $p > p^*$: percolates with high probability.
  - $p < p^*$: does not percolate with high probability.

- What is the value of p^*

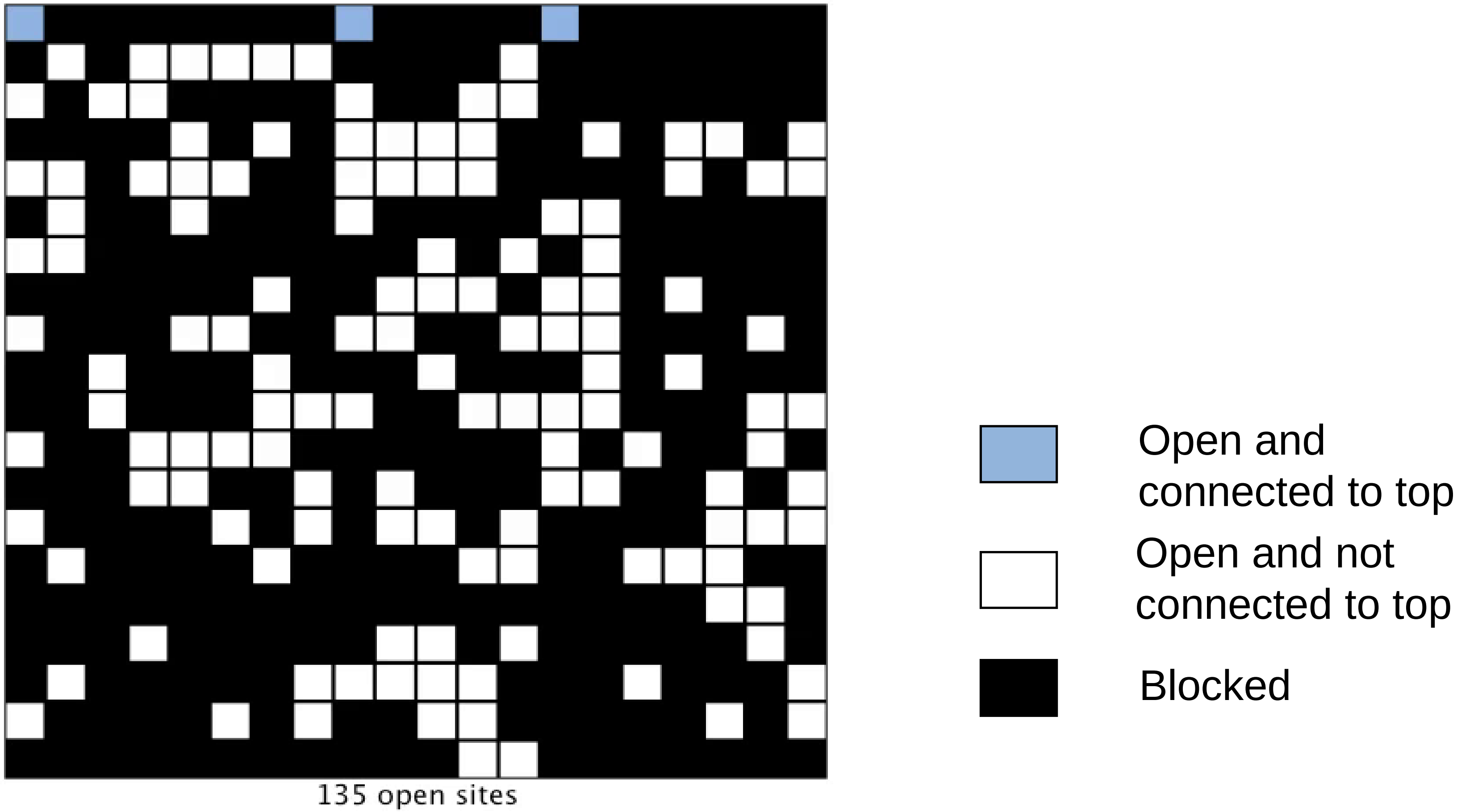


$N = 100$

Monte Carlo simulering

- Initialize all sites in a N-x-N grid to be blocked
- Make randomly chosen sites oepn until their is path between top and bottom.
- Determine *vacancy percentage*, determine p^*.

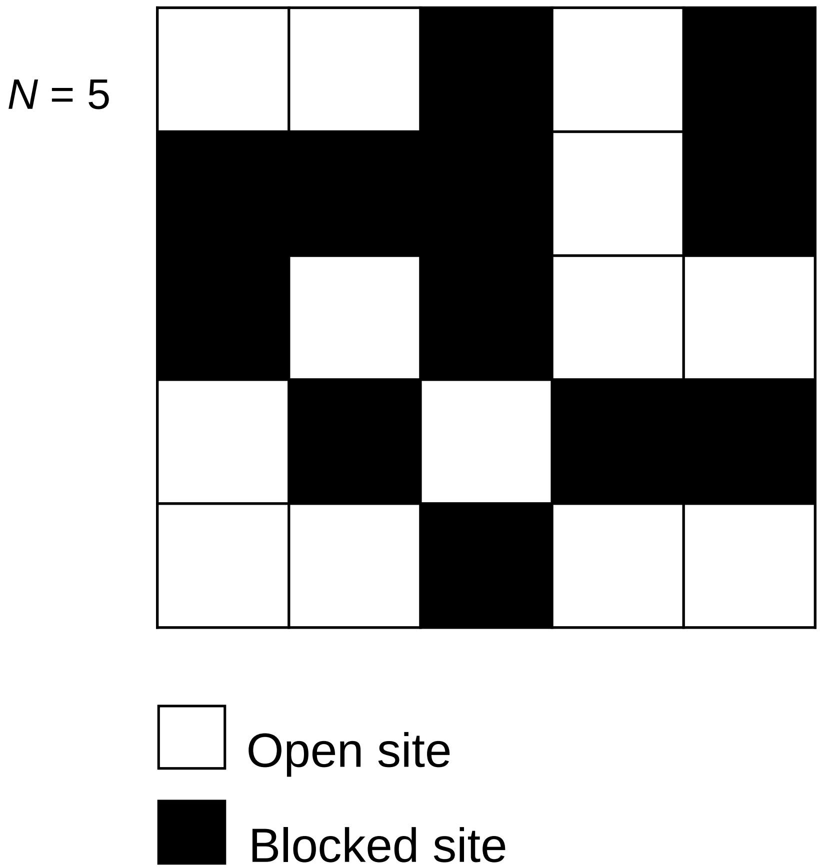
$N = 20$



Model as a connectivity problem

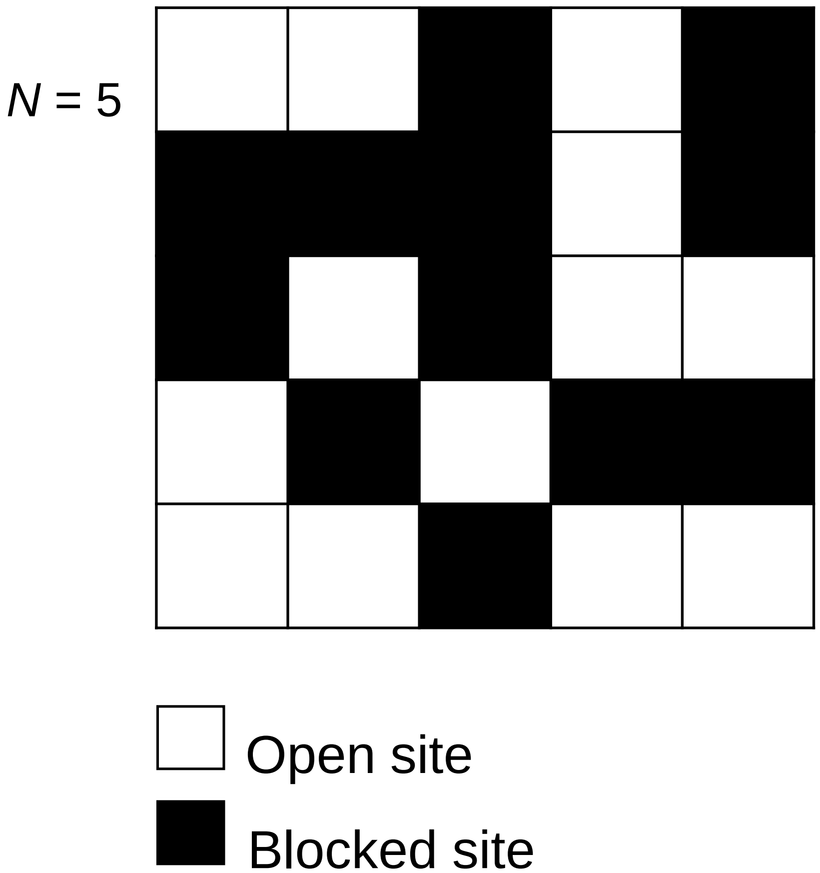
- How to check if a N-x-N system percolates?

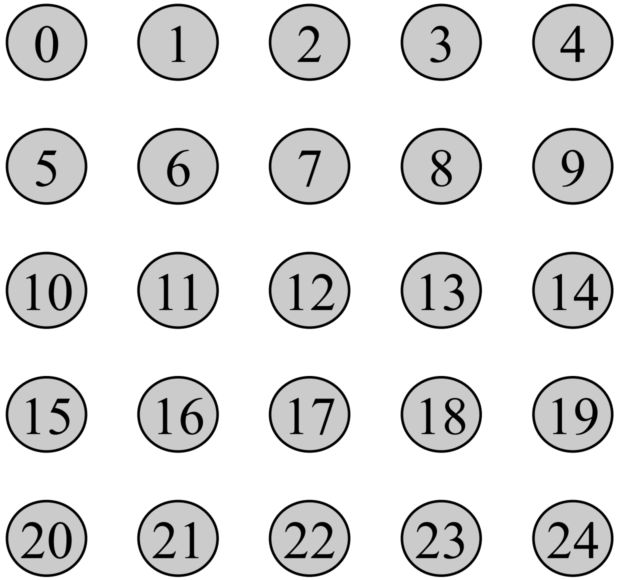
  Modellera as a dynamic connectivity problem and use union-find.



Model as a connectivity problem

✓Create and object for each site and name them from 0 to $N^2 - 1$.





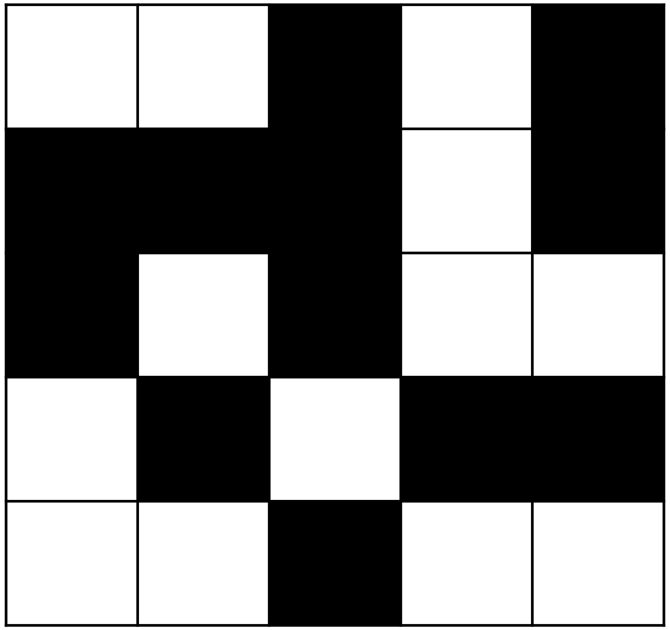
Model as a connectivity problem

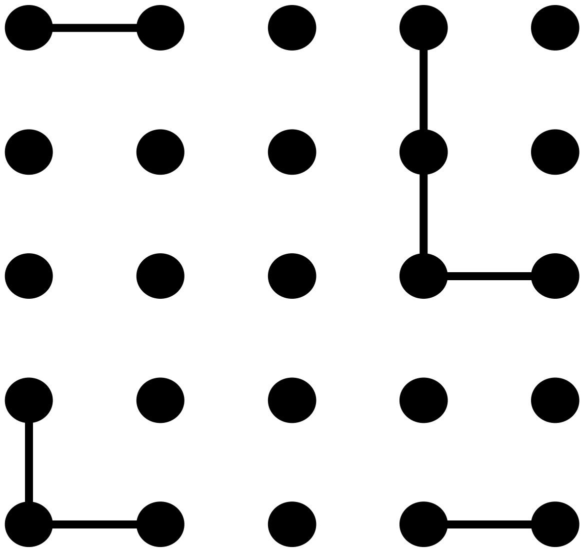
- How to check if a N-x-N system percolates?

✓Create and object for each site and name them from 0 to $N^2 - 1$.

✓Sites are in the same component iff they are connected by open sites.

N = 5





Open site

Blocked site

Model as a connectivity problem

- How to check if a N-x-N system percolates?

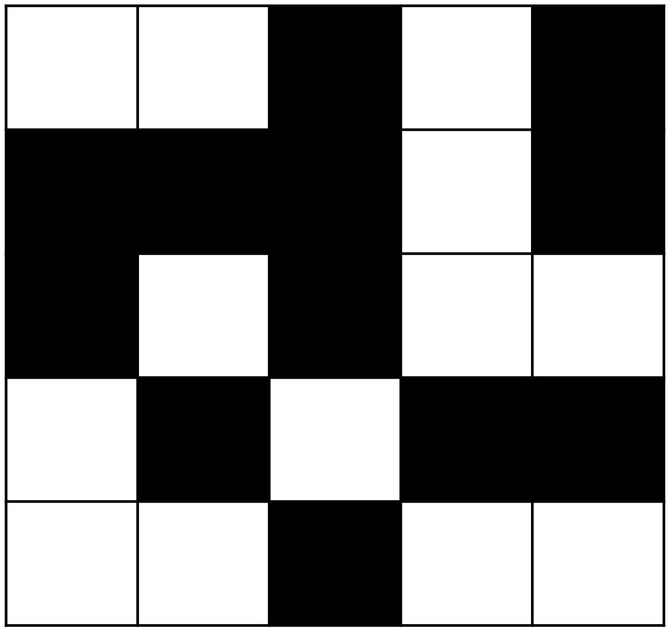
✓ Create and object for each site and name them from 0 to $N^2 - 1$.

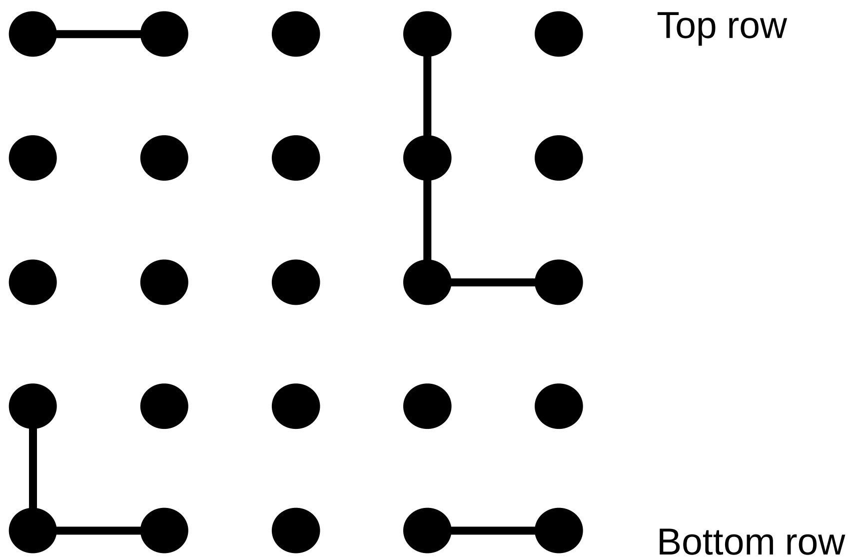
✓ Sites are in the same component iff they are connected by open sites.

✓ System percolates iff any site on the bottom row is to some site on the top row

brute-force algorithm: N^2 calls to connected()

N = 5





Open site

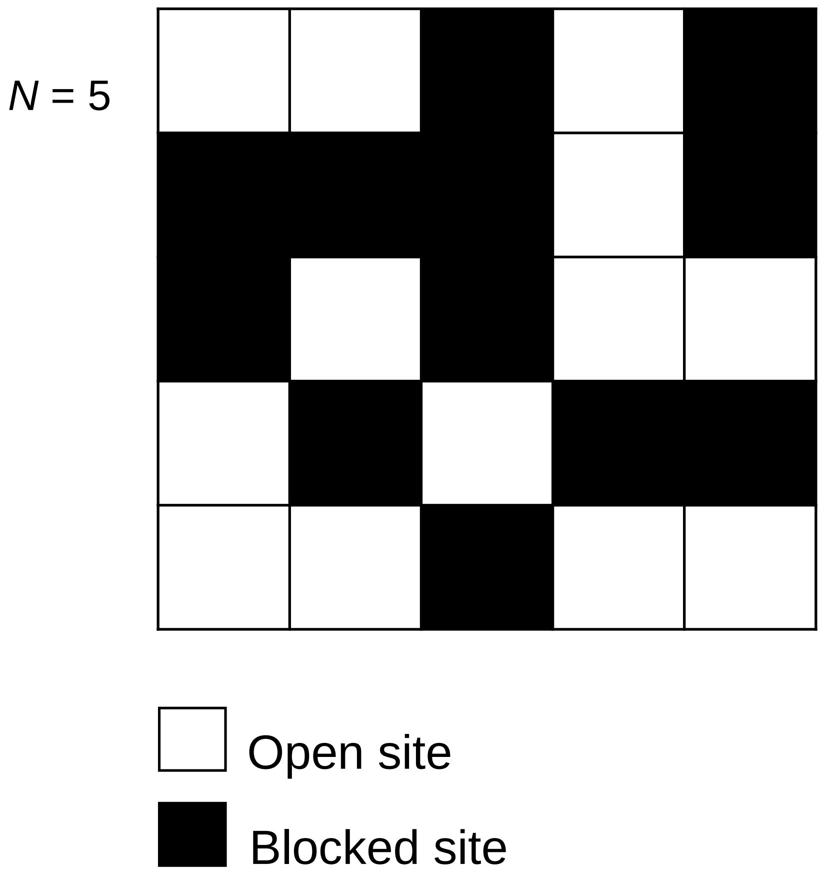
Blocked site

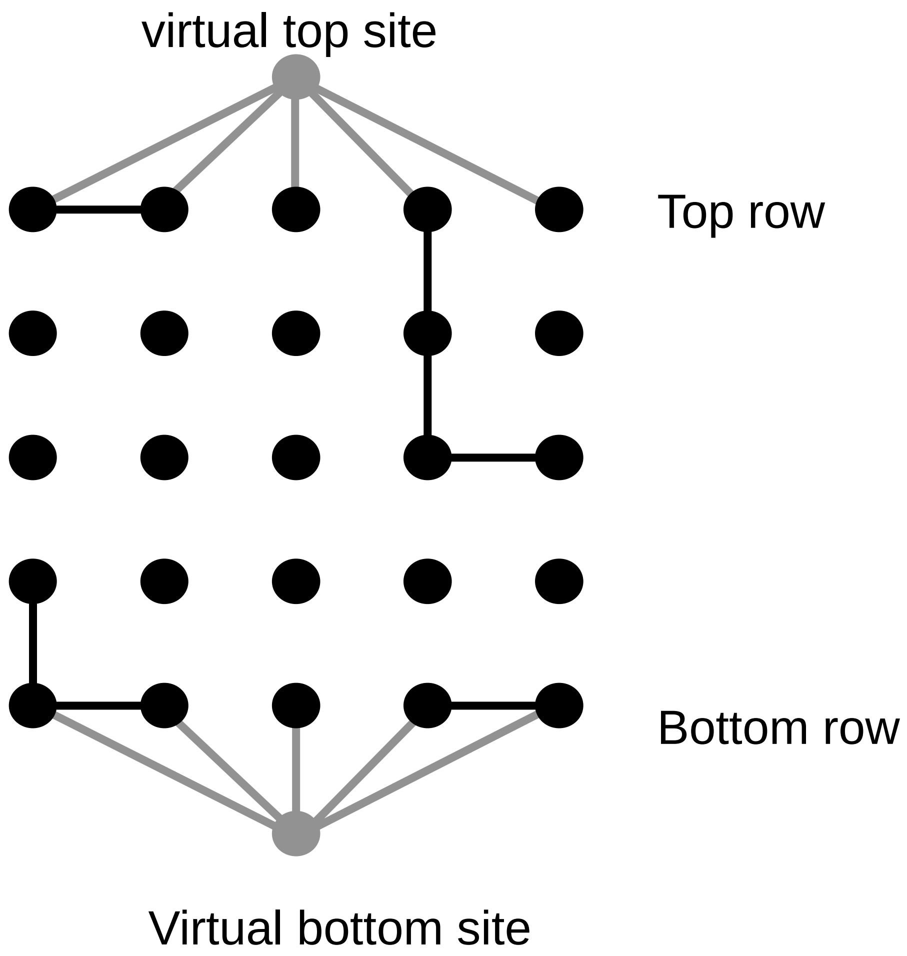
Improvement

- Trick. Add 2 virtual sites that have connections to all sites in top and botto rows, respectively.

✓System percolates iff the virtual top site is connected to the virtual bottom site

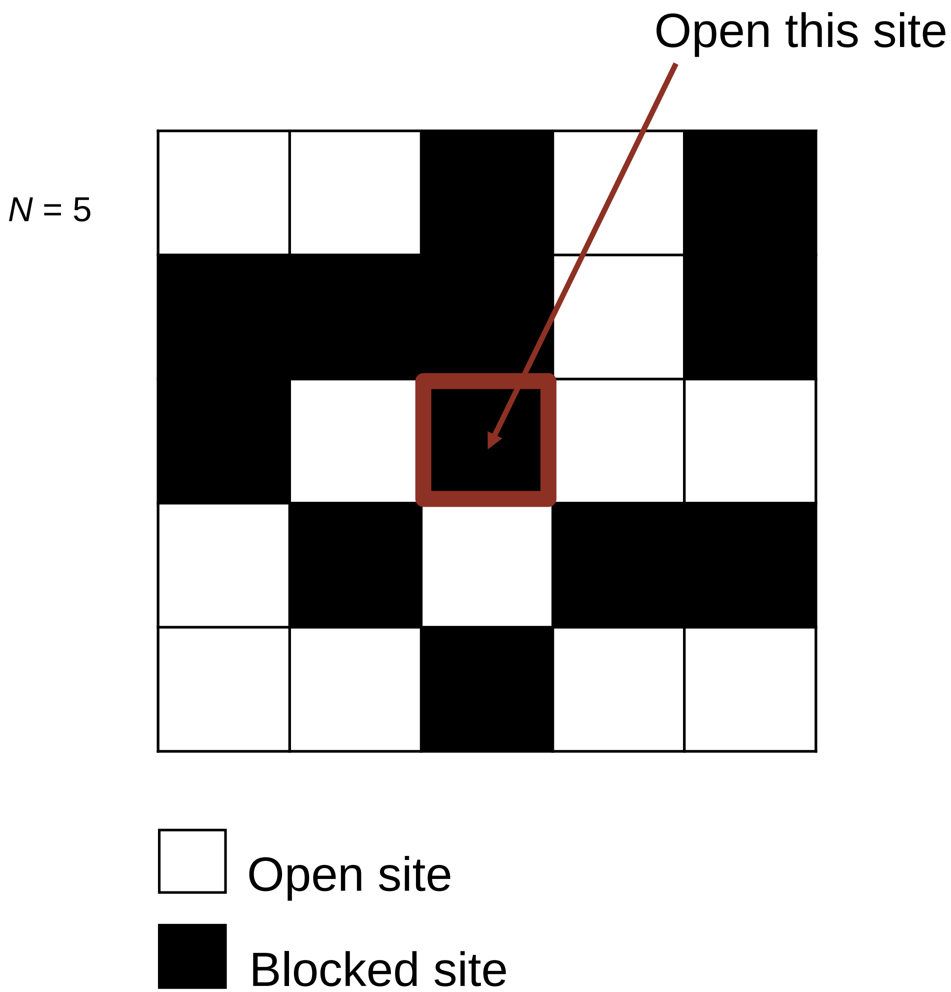
mer effektivt algoritm: bara 1 anrop till connected()

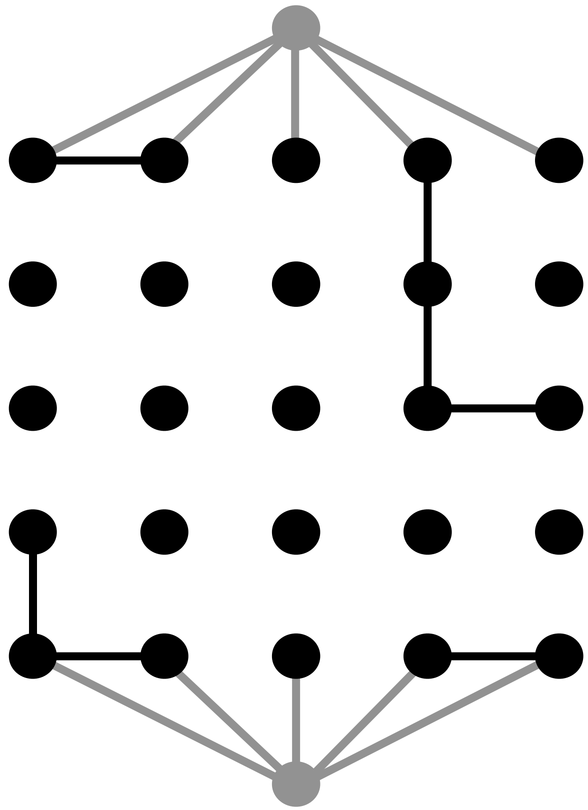




Blocked to open site

- How do we change a site from blocked to open?

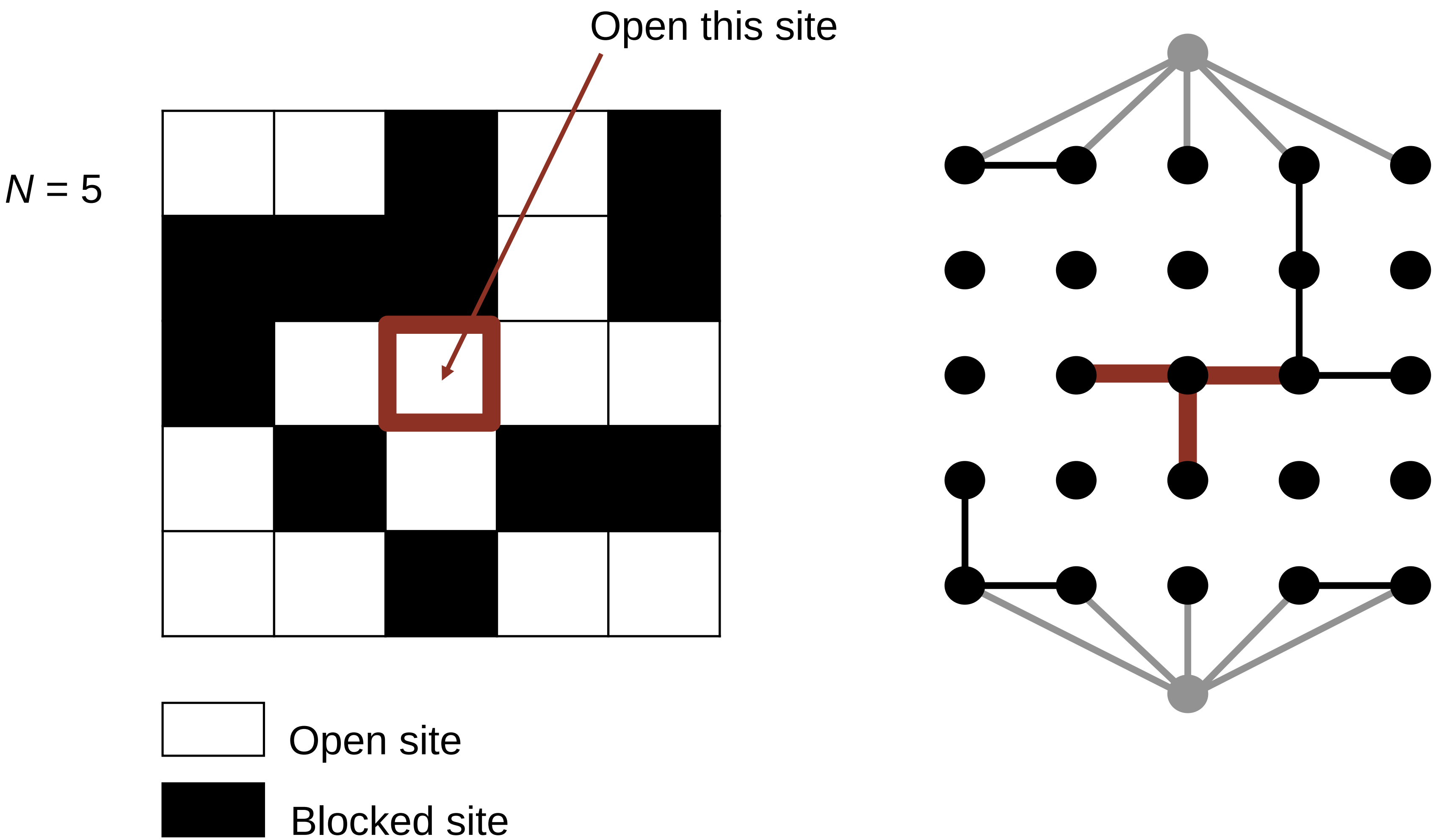




Blocked to open site

- How do we change a site from blocked to open?
- Connect the site to its 4 neighbors .

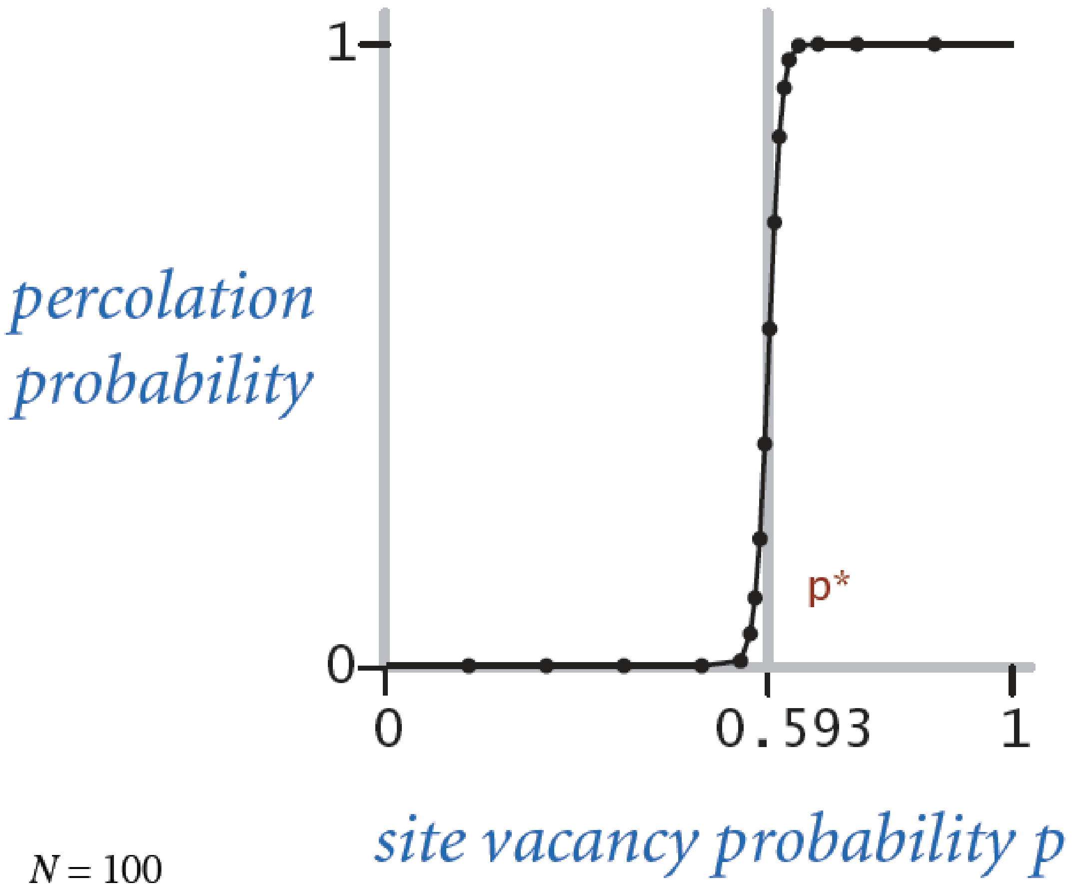
4 calls to union()



Percolation Threshold

- What is the percolation threshold $p*$?
- Approx. 0.592746 for *large square lattices*.

constant only known through simulations



- A fast algorithm enables an accurate answer to a scientific question.

Summary

- Union-Find is an interesting class of algorithm with practical applications
  - Quick-Find, Quick-Union, Weighted Quick-Union, Weighted Quick-Union with Path Compression.
- Using union find as a case study, we have covered the stages of algorithm development:
  - Build a model of the problem
  - Find an algorithm to solve the problem
  - Evaluate. Is it fast enough? Will it fit in memory?
  - If not, determine why?.
  - Find a better algorithm.
  - Itererate until satisfied.