

UNION FIND - dynamic connectivity

Subtext

如何编一个高效的程序：

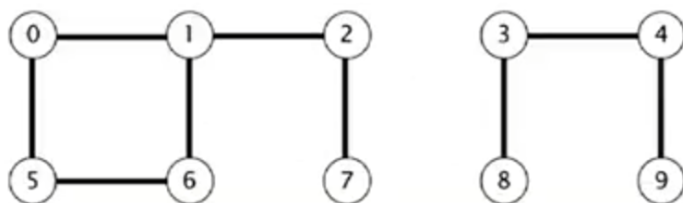
- 为问题建模
- 找到一个合适的算法
- 分析算法运行的时间，所需的空间大小
- 优化

dynamic connectivity 动态连通性

Given a set of N objects.

- **Union command:** connect two objects.
- **Find/connected query:** is there a path connecting the two objects?

```
union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)
connected(0, 7) ✗
connected(8, 9) ✓
union(5, 0)
union(7, 2)
union(6, 1)
union(1, 0)
connected(0, 7) ✓
```



算法的目的：

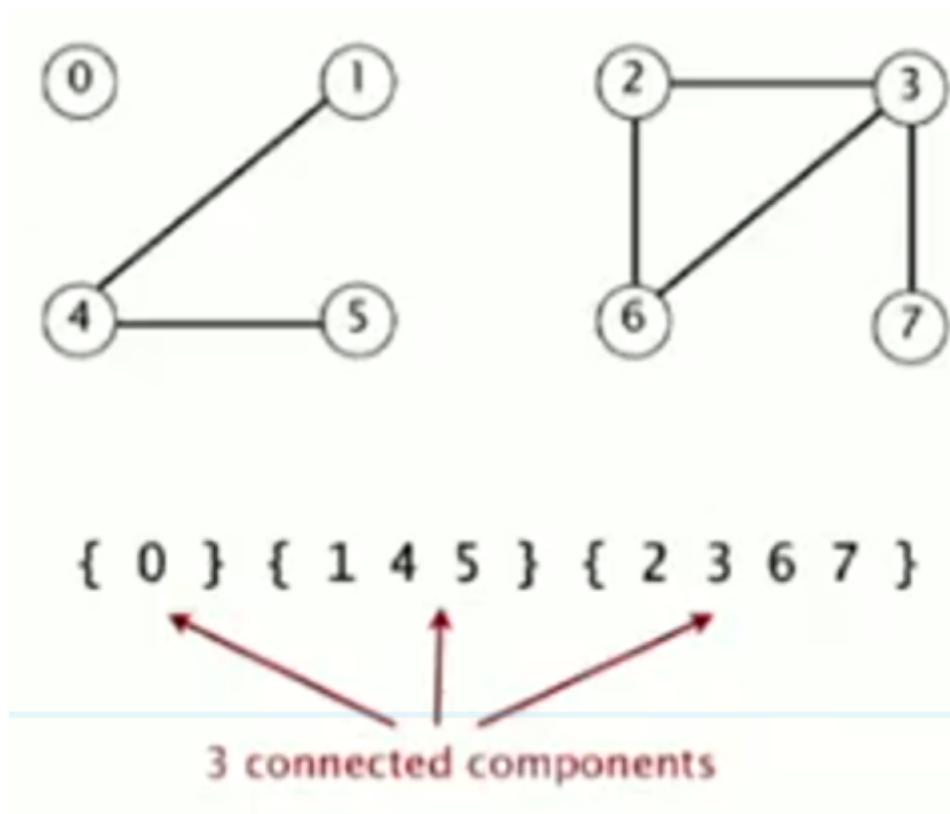
回答问题：两点之间是否存在通路？

Modeling

并查集问题在实际中可能有许多不同的应用场景，但是在编程实现的算法中，我们选取 $0, N - 1$ 作为各个节点的代替（利用整型变量来对问题进行简化描述）。

此外我们还需要定义这个图之中的一些性质：

- 连通性：p, q是存在连接通路的
- 对称性：如果p, q是连通的；则q, p也是连通的
- 传递性：如果p, q是连通的；q, r也是连通的；则p, r是连通的
- 连通分量：互相连接的最小图（集合） **connected components**



连通分量性质：

- 连通分量中的所有节点都是互相关连通的
- 连通分量中的节点不与连通分量之外的节点连通

算法需要进行的操作（函数）：

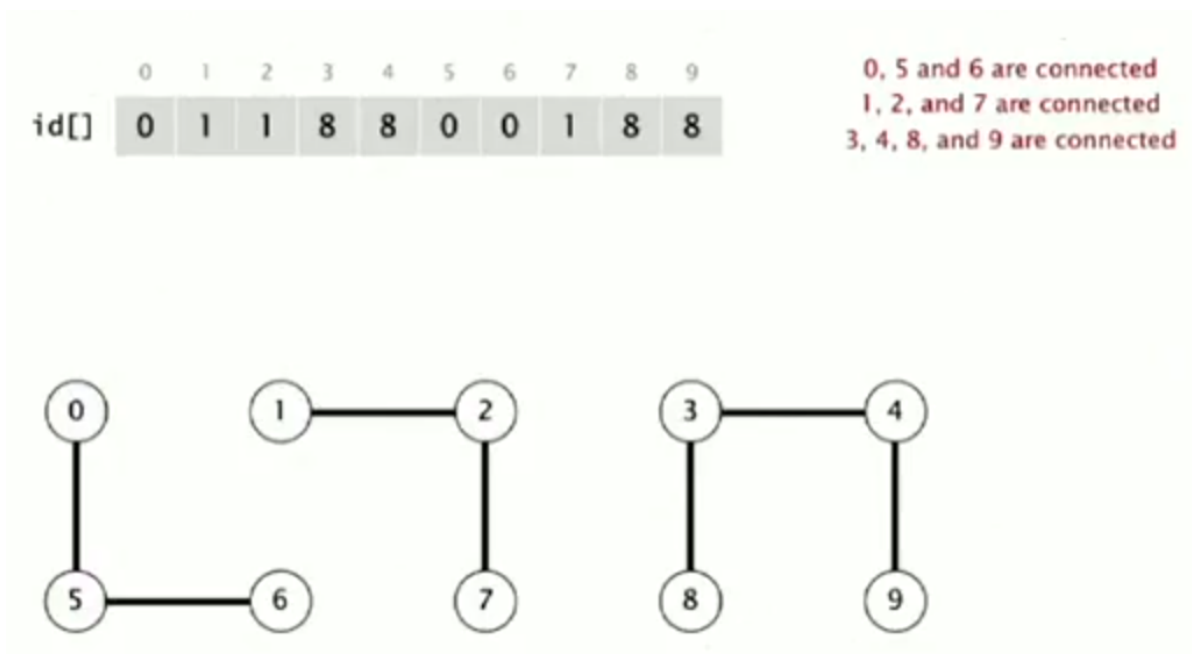
- Find query: 检查两个节点之间是否是连通的
- Union: 将两个连通分量合并

```
public class UF:
    UF(int N)
    void union (int p, int q)
    boolean connected(int p, int q)
```

Solution 1: Quick -find [eager approach]

数据结构：

- 整型数组 $id[]$ （大小为N）
- p, q 是连通的当且仅当他们有着相同的id值



功能函数:

- Find: 判断p, q是否连通; 直接比较两者的id值即可
- Union: 将id值都变为约定好的值 (实现简单, 但对于较大数目的节点有待改进)

```
#流程
#初始化
id[10] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
union(4, 3)
#算法统一将id值定为union的第二个参数
id[10] = [0, 1, 2, 3, 3, 5, 6, 7, 8, 9]
#更新id值的时候要注意, 需要更新所有连通分量中的id值
... ..
```

Java实现:

```
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 connected(int p, int q)
    {
        return id[p] == id[q];
    }

    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;
    }
}
```

python 实现;

```
class QuickFindUF:
    def __init__(self, N):
        self.id = list(range(N))

    def connected(self, p, q):
        return self.id[p] == self.id[q]

    def union(self, p, q):
        pid = self.id[p]
        qid = self.id[q]
        for i in range(len(self.id)):
            if self.id[i] == pid:
                self.id[i] = qid
```

algorithm	initialize	union	find
QuickFind	N	N	1

总结:

快速查找算法不利于解决大问题 huge problems

这种算法在时间量级上是不合理的，虽然简单但是在现实问题中的时间复杂度是不可接受的。例如对N个节点进行union操作就要花费 N^2 的时间，对于构建高效的算法来说，这是过于复杂的。

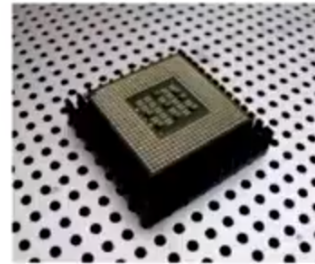
quadratic time is unacceptable

Quadratic algorithms do not scale

Rough standard (for now).

- 10^9 operations per second.
- 10^9 words of main memory.
- Touch all words in approximately 1 second.

a truism (roughly)
since 1950!

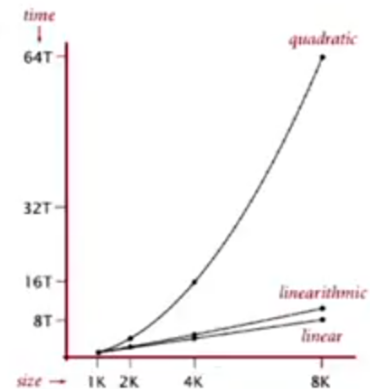


Ex. Huge problem for quick-find.

- 10^9 union commands on 10^9 objects.
- Quick-find takes more than 10^{18} operations.
- 30+ years of computer time!

Quadratic algorithms don't scale with technology.

- New computer may be 10x as fast.
- But, has 10x as much memory \Rightarrow want to solve a problem that is 10x as big.
- With quadratic algorithm, takes 10x as long!



Solution 2: Quick-Union [lazy approach]

数据结构:

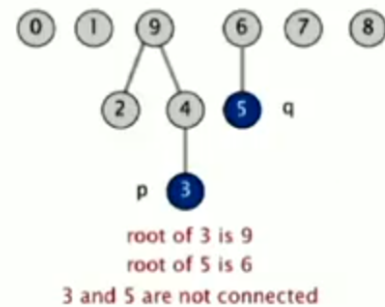
- 整型数组 $id[]$ (大小为N)
- 定义 $id[i]$ 的值是该节点的父节点的id (该算法将各个连通分量维护为一个树型结构)
- 确定根节点: $id[id[id[... id[i]...]]]$

Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

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

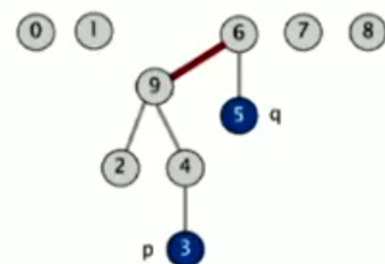


Find. Check if `p` and `q` have the same root.

Union. To merge components containing `p` and `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	6

↑
only one value changes



功能函数:

- Find: 判断`p`, `q`是否连通; 直接比较两者的root节点是否相同即可
- Union(`p`, `q`): `p`节点的root连接到`q`节点的root上

java 实现

```
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 (id[i] != i) i = id[i];
        return i;
    }

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

    public void union(int p, int q)
```

```

{
    int i = root(p);
    int j = root(q);
    id[i] = j;
}
}

```

python 实现

```

class QuickUnionUF:
    def __init__(self, N):
        self.id = list(range(N))

    def root(self, i):
        while self.id[i] != i:
            i = self.id[i]
        return i

    def connected(self, p, q):
        return self.root(p) == self.root(q)

    def union(self, p, q):
        i = self.root(p)
        j = self.root(q)
        self.id[i] = j

```

algorithm	initialize	union	find
Quickunion	N	N↑	N

总结:

对于快速合并算法，采取的优化策略是有针对的对数据结构进行更新。避免了在合并的时候进行多次无用的索引更新。

但是同样的，这个算法也不是完美的。他不适合需要大量操作的场景，在构建连通分量树的时候，我们并没有对树的结构进行优化，因此可能出现的情况是：树的结构大概率是冗杂的，他可能非常高瘦。这就导致在查找root索引的时候会浪费大量的时间。

Improvement 1 weighting

针对quick union方法中可能出现高树的情况，可以采取的 优化措施是更新带权树。

目标如下：

- 记录每个树的节点个数
- 将小树连接到大树上从而实现更平衡的树

这样生成的树虽然不是绝对平衡的，但是在相比于原来平均高度有着极大的缩减。

代码方面，需要更新的数据结构是size[]，用来记录对应树的节点个数。

running time分析

find: 和节点的深度成正比

union: 常数时间的复杂度

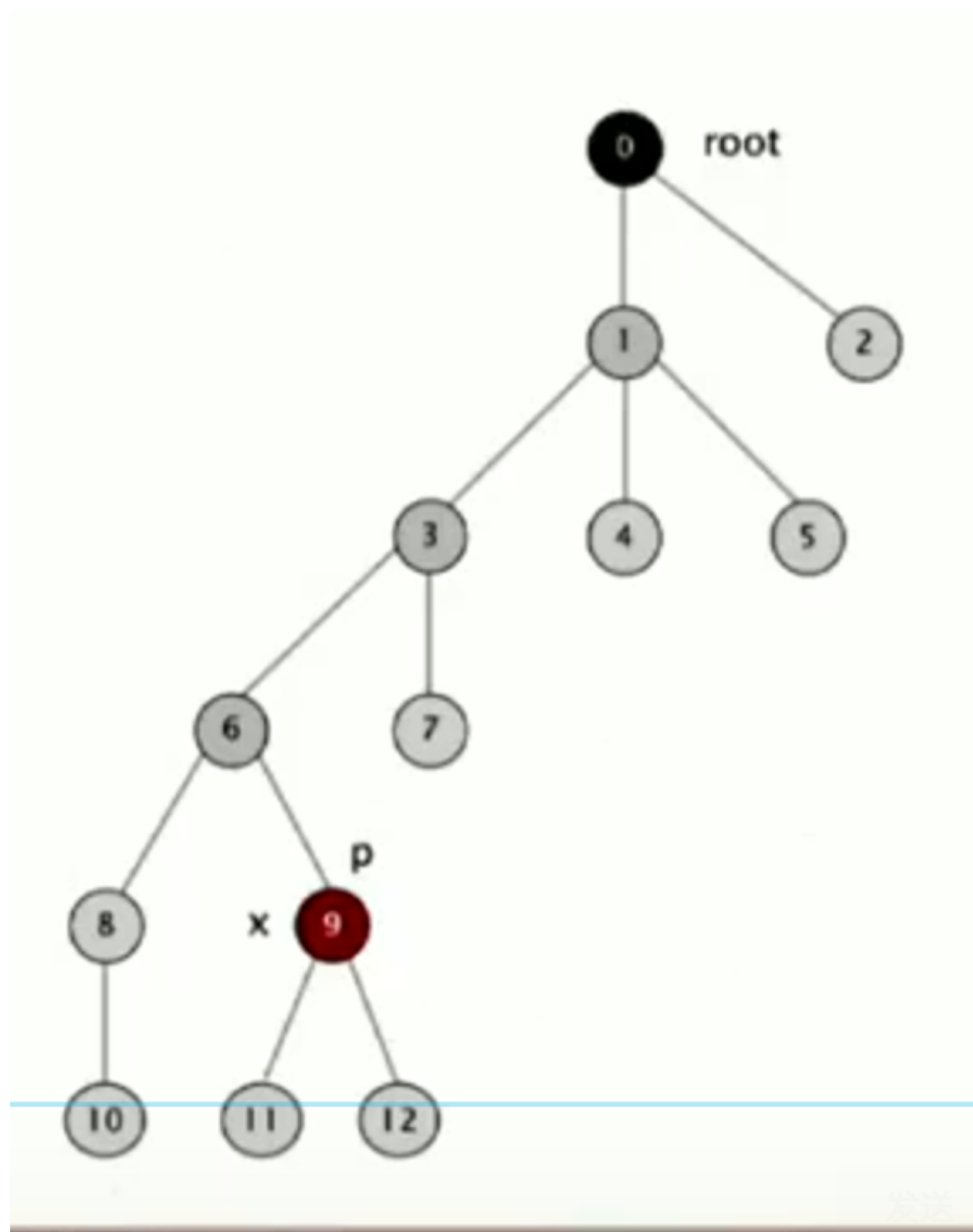
结论: 节点的深度至多是 $\log_2 N$ 级别的 (当两个树进行合并的时候, 总是小的树合并到大的树中。树中节点的个数至多会翻一倍, 因此当最后所有节点都加入到同一颗树中时, 对于 N 个节点来说, 需要合并 $\lg N$ 次)

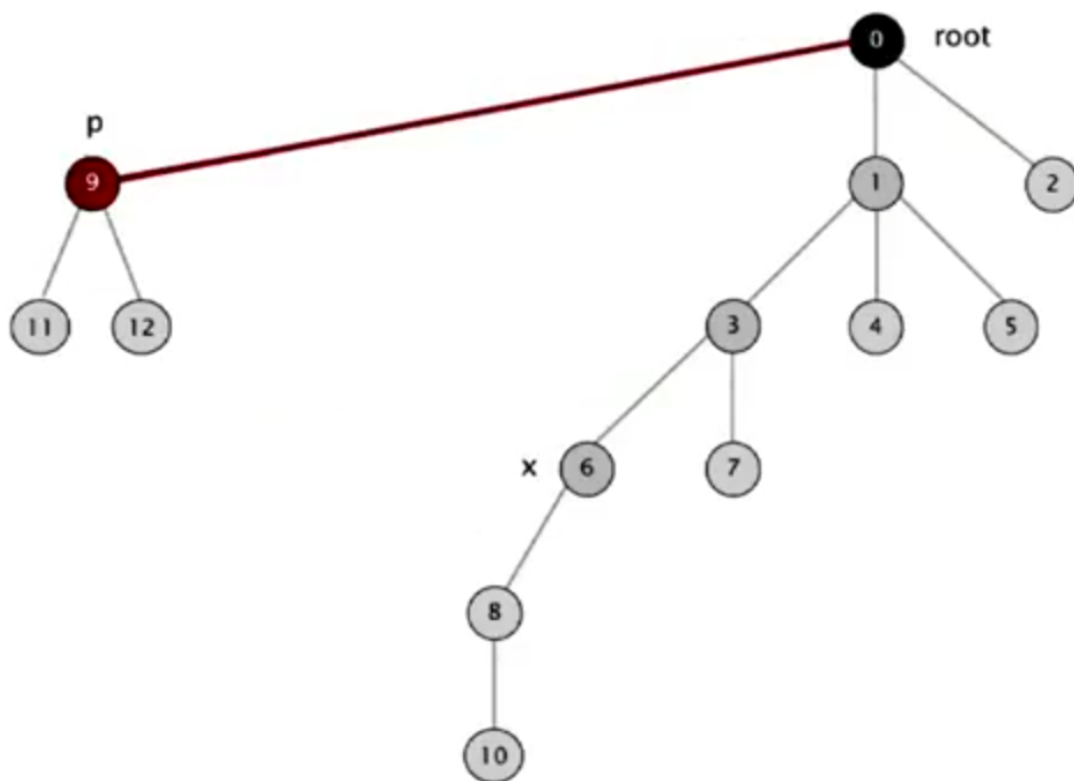
Algorithm	initialize	find	union
Improvement 1 weighted	N	$\lg N$	$\lg N$

Improvement 2 path compression

在带权树的基础上, 对树进行进一步的优化。在计算节点对应的root时, 算法需要从节点开始遍历。事实上, 可以对树进行铺平的操作, 即每个节点在计算root的时候就将遍历时的根节点进行更新。更新的代码如下

```
private int root(int i)
{
    while (i != id[i])
    {
        id[i] = id[id[i]];
        i = id[i];
    }
    return i;
}
```



对于N个对象，进行M次union find操作的时间复杂度 $\leq c(N + Mlg^*N)$ ，其中 lg^*N 是迭代对数函数。是指取多少次对数可以将N变为1。在现实生活中，我们普遍认为这个函数的值域是 ≤ 5 的。（ $lg^*2^{65536} = 5$ ）

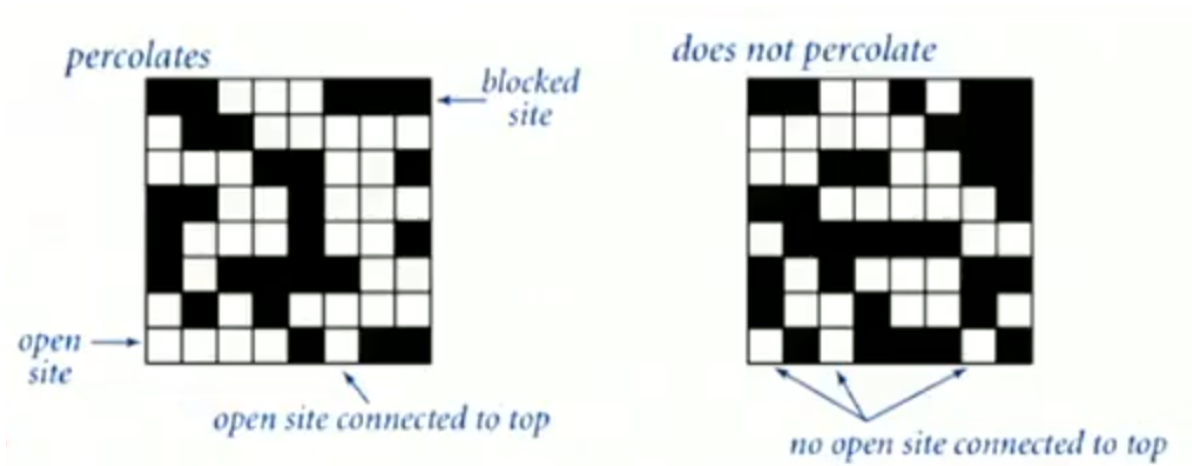
因此，压缩路径的带权快速合并算法可以被认为是**几乎线性时间的复杂度**。

并查集问题并不存在**线性时间复杂度的算法**（已证明）

Algorithm	worst-cost time
quick-find	MN
quick-union	MN
weighted QU	$N + MlogN$
QU + path compression	$N + MlogN$
weighted QU + path compression	$N + Mlg^*N$

Application

渗滤系统（上下是否连通）

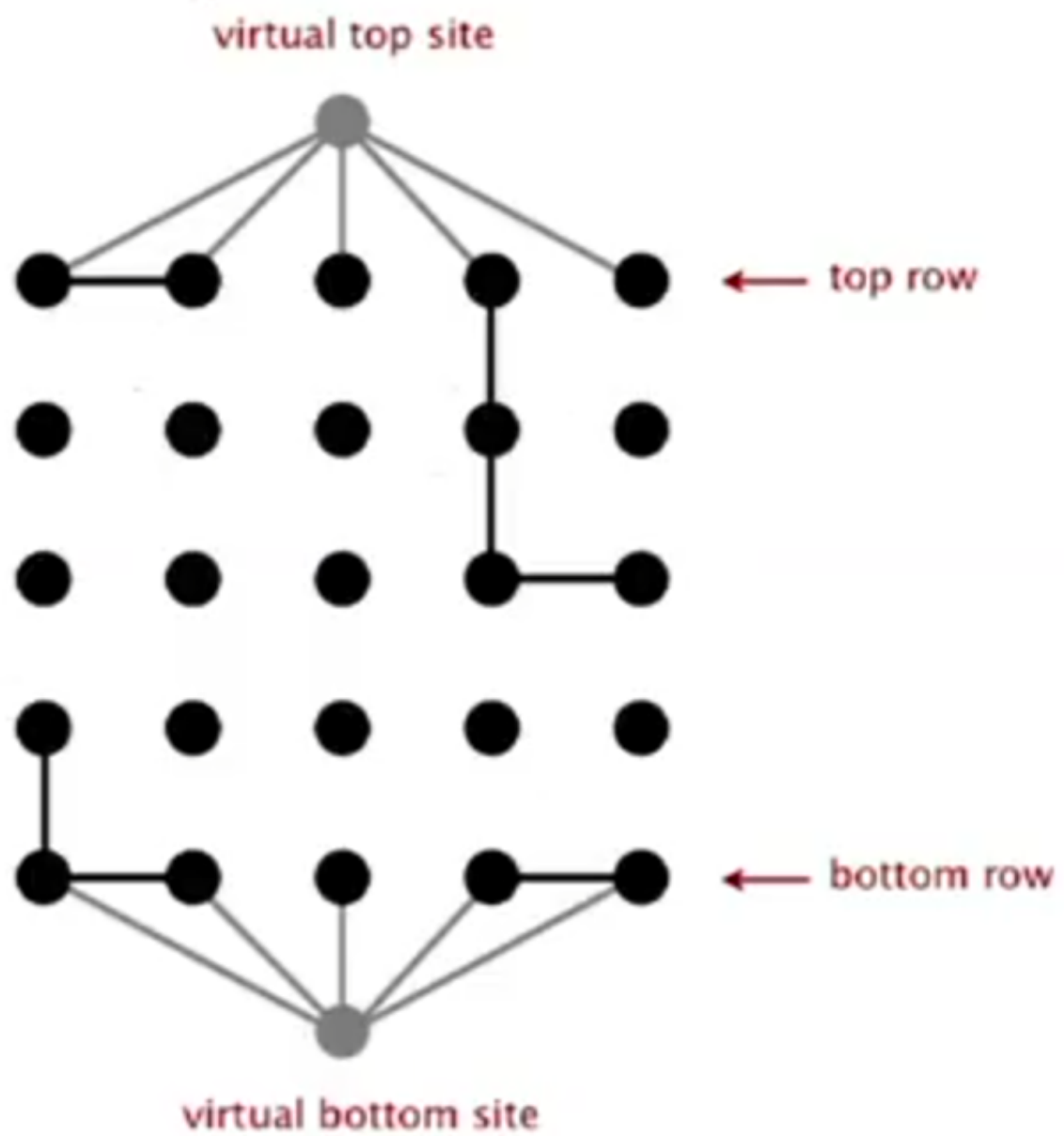


每个位（方格）的开放概率是 p ，该模型可以应用到很多实际的问题中（电力系统，网络结构，社交网络等等）

当系统中的位以某种概率进行开放的时候，确定的系统是否是渗滤是容易的，但是渗滤存在的阈值时难以用数学方法进行确定的。

我们可以采用蒙特卡罗仿真的手段对系统进行测试，通过随机开放位直到系统是渗滤的，经过不断重复实验最终找到阈值概率。

为了避免在验证上下 N 位是否连通时使用暴力查找，可以规定一个虚拟的连通位，如下：



这样可以在常数时间内确定上下是否连通。